Learning Web Component Development

Web components are an exciting new set of web standards used to provide reusable and powerful widgets by encapsulating and extending HTML and CSS. Web components are rapidly coming of age and are ready to make their debut in your browser.

Starting with an introduction to all the core concepts of web component specifications, you will be taken through building your own custom clock component. You will then get to grips with Shadow DOM, HTML import, and templating as you create a component using pure JavaScript. Following this, you’ll explore the core tools and libraries for web component development, including Polymer, Bosonic, Mozilla Brick, and ReactJS, and learn how to put them to work for practical development. This book will provide you with a detailed understanding of architecture, configuration, and selecting the right tool for you and your needs.

Who this book is written for

This book is the perfect reference for any web developer looking for an introduction to the new tools and techniques used to create web components.

What you will learn from this book

- Get hands-on experience with native JavaScript for web component creation
- Discover the core of Polymer and use it to craft your web components
- Use Mozilla Brick to customize and create web components
- Debug web components with the power of ReactJS
- Take a reactive approach to web component creation
- Master the Bosonic framework for practical web component design

Who this book is written for

This book is the perfect reference for any web developer looking for an introduction to the new tools and techniques used to create web components.
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Introducing Web Components'
- A synopsis of the book’s content
- More information on *Learning Web Component Development*
About the Author

Sandeep Kumar Patel is a senior web developer and the founder of www.tutorialsavvy.com, a programming blog that has been widely read since its inception in 2012. He has over 5 years of experience in object-oriented JavaScript and JSON-based web applications development. He is GATE-2005 Information Technology (IT) qualified and has a master's degree from VIT University, Vellore. You can get to know more about him by looking at his LinkedIn profile (http://www.linkedin.com/in/techblogger). He has received the DZone Most Valuable Blogger (MVB) award for technical publications related to web technologies. His article can be viewed at http://www.dzone.com/users/sandeepgiet. He has also received the Java Code Geeks (JCG) badge for a technical article published on the JGC website. His article can be viewed at http://www.javacodegeeks.com/author/sandeep-kumar-patel/.

His other books are listed as follows:

- *Instant GSON*
- *Responsive Web Design with AngularJS*
Preface

Welcome to *Learning Web Component Development*. If you want to learn and understand the W3C web component specification and develop a custom web component using Polymer, Bosonic, Mozilla Brick, and ReactJS, then this is the book for you. It offers a systematic approach to build a responsive web application. All the key features of web component specification that can help in building a web component are explained in this book, and are accompanied by the detailed code you will need.

**What this book covers**

*Chapter 1, Introducing Web Components*, will provide an introduction to web components. It includes a detailed explanation of the building blocks of web component.

*Chapter 2, Introducing Polymer*, is all about Google's Polymer library. It explains the architecture of this library. It also explores the core and paper elements.

*Chapter 3, Developing Web Components Using Polymer*, is all about custom web component development using PolymerJS. It provides a step-by-step guide to develop a custom component using this library.

*Chapter 4, Exploring Bosonic Tools for Web Component Development*, focuses on Bosonic tools. It explains how to use these tools to create a custom component.

*Chapter 5, Developing Web Components Using Mozilla Brick*, deals with the Mozilla Brick library. It includes a brief introduction to Brick library, and it also includes a coded example of the various components using Brick.

*Chapter 6, Building Web Components with ReactJS*, is all about ReactJS. It explains what a reactive approach is. It includes coded examples of creating web component using ReactJS.
Appendix, Web Component References, lists all of the online sites and forums on web component for further study.
Introducing Web Components

In this chapter, we will learn about the web component specification in detail. Web component is changing the web application development process. It comes with standard and technical features, such as templates, custom elements, Shadow DOM, and HTML Imports.

The main topics that we will cover in this chapter about web component specification are as follows:

- What are web components?
- Benefits and challenges of web components
- The web component architecture
- Template element
- HTML Import
- Shadow DOM
- Custom elements
- Building a digital clock component
- The X-Tag library
- web component libraries

What are web components?

Web components are a W3C specification to build a standalone component for web applications. It helps developers leverage the development process to build reusable and reliable widgets. A web application can be developed in various ways, such as page focus development and navigation-based development, where the developer writes the code based on the requirement. All of these approaches fulfil the present needs of the application, but may fail in the reusability perspective. This problem leads to component-based development.
Benefits and challenges of web components

There are many benefits of web components:

- A web component can be used in multiple applications. It provides interoperability between frameworks, developing the web component ecosystem. This makes it **reusable**.
- A web component has a template that can be used to put the entire markup separately, making it more **maintainable**.
- As web components are developed using HTML, CSS, and JavaScript, it can run on different browsers. This makes it **platform independent**.
- Shadow DOM provides **encapsulation mechanism** to style, script, and HTML markup. This encapsulation mechanism provides **private scope** and prevents the content of the component being affected by the external document.

Equally, some of the challenges for a web component include:

- **Implementation**: The W3C web component specification is very new to the browser technology and not completely implemented by the browsers.
- **Shared resource**: A web component has its own scoped resources. There may be cases where some of the resources between the components are common.
- **Performance**: Increase in the number of web components takes more time to get used inside the DOM.
- **Polyfill size**: The polyfill are a workaround for a feature that is not currently implemented by the browsers. These polyfill files have a large memory footprint.
- **SEO**: As the HTML markup present inside the template is inert, it creates problems in the search engine for the indexing of web pages.

The web component architecture

The W3C web component specification has four main building blocks for component development. Web component development is made possible by template, HTML Imports, Shadow DOM, and custom elements and decorators. However, decorators do not have a proper specification at present, which results in the four pillars of web component paradigm. The following diagram shows the building blocks of web component:
These four pieces of technology power a web component that can be reusable across the application. In the coming section, we will explore these features in detail and understand how they help us in web component development.

**Template element**
The HTML `<template>` element contains the HTML markup, style, and script, which can be used multiple times. The templating process is nothing new to a web developer. Handlebars, Mustache, and Dust are the templating libraries that are already present and heavily used for web application development. To streamline this process of template use, W3C web component specification has included the `<template>` element.

This template element is very new to web development, so it lacks features compared to the templating libraries such as Handlebars.js that are present in the market. In the near future, it will be equipped with new features, but, for now, let's explore the present template specification.

**Template element detail**
The HTML `<template>` element is an HTMLTemplateElement interface. The interface definition language (IDL) definition of the template element is listed in the following code:

```javascript
interface HTMLTemplateElement : HTMLElement {
    readonly attribute DocumentFragment content;
};
```
Introducing Web Components

The preceding code is written in IDL language. This IDL language is used by the W3C for writing specification. Browsers that support HTML Import must implement the aforementioned IDL. The details of the preceding code are listed here:

- **HTMLTemplateElement**: This is the template interface and extends the HTMLElement class.
- **content**: This is the only attribute of the HTML template element. It returns the content of the template and is read-only in nature.
- **DocumentFragment**: This is a return type of the content attribute. DocumentFragment is a lightweight version of the document and does not have a parent.

To find out more about DocumentFragment, use the following link: https://developer.mozilla.org/en/docs/Web/API/DocumentFragment

Template feature detection

The HTML `<template>` element is very new to web application development and not completely implemented by all browsers. Before implementing the template element, we need to check the browser support. The JavaScript code for template support in a browser is listed in the following code:

```html
<!DOCTYPE html>
<html>
<head lang="en">
    <meta charset="UTF-8">
    <title>Web Component: template support</title>
</head>
<body>
<h1 id="message"></h1>
<script>
    var isTemplateSupported = function () {
        var template = document.createElement("template");
        return 'content' in template;
    };
    var isSupported = isTemplateSupported(),
        message = document.getElementById("message");
    if (isSupported) {
        message.innerHTML = "Template element is supported by the browser.";
    }
</script>
</body>
</html>
```
In the preceding code, the `isTemplateSupported` method checks the `content` property present inside the template element. If the `content` attribute is present inside the template element, this method returns either `true` or `false`. If the template element is supported by the browser, the `h1` element will show the support message. The browser that is used to run the preceding code is Chrome 39 release. The output of the preceding code is shown in following screenshot:

```

```

The preceding screenshot shows that the browser used for development is supporting the HTML template element.

There is also a great online tool called **Can I Use** for checking support for the template element in the current browser. To check out the template support in the browser, use the following link: http://caniuse.com/#feat=template
The following screenshot shows the current status of the support for the template element in the browsers using the Can I Use online tool.

Inert template
The HTML content inside the template element is inert in nature until it is activated. The inertness of template content contributes to increasing the performance of the web application. The following code demonstrates the inertness of the template content:

```html
<!DOCTYPE html>
<html>
<head lang="en">
    <meta charset="UTF-8">
    <title>Web Component: A inert template content example.</title>
</head>
<body>
    <div id="message"></div>
    <template id="aTemplate">
        <img id="profileImage" src="http://www.gravatar.com/avatar/c6e6c57a2173fcbf2afdd5fe6786e92f.png">
        <script>
        </script>
    </template>
</body>
</html>
```
In the preceding code, a template contains an image element with the `src` attribute, pointing to a Gravatar profile image, and an inline JavaScript `alert` method. On page load, the `document.getElementById` method is looking for an HTML element with the `#profileImage` ID. The output of the preceding code is shown in the following screenshot:
The preceding screenshot shows that the script is not able to find the HTML element with the profileImage ID and renders null in the browser. From the preceding screenshot it is evident that the content of the template is inert in nature.

**Activating a template**

By default, the content of the `<template>` element is inert and are not part of the DOM. The two different ways that can be used to activate the nodes are as follows:

- Cloning a node
- Importing a node

**Cloning a node**

The `cloneNode` method can be used to duplicate a node. The syntax for the `cloneNode` method is listed as follows:

```html
<Node> <target node>.cloneNode(<Boolean parameter>)
```

The details of the preceding code syntax are listed here:

- This method can be applied on a node that needs to be cloned.
- The return type of this method is `Node`.
- The input parameter for this method is of the `Boolean` type and represents a type of cloning. There are 2 different types of cloning, listed as follows:
  - **Deep cloning**: In deep cloning, the children of the targeted node also get copied. To implement deep cloning, the `Boolean` input parameter to `cloneNode` method needs to be `true`.
  - **Shallow cloning**: In shallow cloning, only the targeted node is copied without the children. To implement shallow cloning the `Boolean` input parameter to `cloneNode` method needs to be `false`.

The following code shows the use of the `cloneNode` method to copy the content of a template, having the `h1` element with some text:

```html
<!DOCTYPE html>
<html>
<head lang="en">
  <meta charset="UTF-8">
  <title>
    Web Component: Activating template using cloneNode method
  </title>
</head>
```
In the preceding code, the template element has the `aTemplate` ID and is referenced using the `querySelector` method. The HTML markup content inside the template is then retrieved using a `content` property and saved in a `templateContent` variable. The `cloneNode` method is then used for deep cloning to get the activated node that is later appended to a `div` element. The following screenshot shows the output of the preceding code:
Introducing Web Components

To find out more about the `cloneNode` method visit: https://developer.mozilla.org/en-US/docs/Web/API/Node.cloneNode

Importing a node

The `importNode` method is another way of activating the template content. The syntax for the aforementioned method is listed in the following code:

```html
<Node> document.importNode(<target node>,<Boolean parameter>)
```

The details of the preceding code syntax are listed as follows:

- This method returns a copy of the node from an external document.
- This method takes two input parameters. The first parameter is the target node that needs to be copied. The second parameter is a Boolean flag and represents the way the target node is cloned. If the Boolean flag is `false`, the `importNode` method makes a shallow copy, and for a `true` value, it makes a deep copy.

The following code shows the use of the `importNode` method to copy the content of a template containing an `h1` element with some text:

```html
<!DOCTYPE html>
<html>
<head lang="en">
  <meta charset="UTF-8">
  <title>Web Component: Activating template using importNode method</title>
</head>
<body>
<div id="container"></div>
<template id="aTemplate">
  <h1>Template is activated using importNode method.</h1>
</template>
<script>
var aTemplate = document.querySelector("#aTemplate");
container = document.getElementById("container");
templateContent = aTemplate.content,
activeContent = document.importNode(templateContent,
true);
```
In the preceding code, the template element has the `aTemplate` ID and is referenced using the `querySelector` method. The HTML markup content inside the template is then retrieved using the `content` property and saved in the `templateContent` variable. The `importNode` method is then used for deep cloning to get the activated node that is later appended to a `div` element. The following screenshot shows the output of the preceding code:

![Template is activated using importNode method.](image)

To find out more about the `importNode` method, visit: [http://mdn.io/importNode](http://mdn.io/importNode)
**HTML Import**

The HTML Import is another important piece of technology of the W3C web component specification. It provides a way to include another HTML document present in a file with the current document. HTML Imports provide an alternate solution to the **<iframe>** element, and are also great for resource bundling. The syntax of the HTML Imports is listed as follows:

```
<link rel="import" href="fileName.html">
```

The details of the preceding syntax are listed here:

- The HTML file can be imported using the **<link>** tag and the **rel** attribute with import as the value.
- The **href** string points to the external HTML file that needs to be included in the current document.

The **import** element is implemented by the **HTMLElementLink** class. The IDL definition of HTML Import is listed in the following code:

```
partial interface LinkImport {
  readonly attribute Document? import;
}

HTMLLinkElement implements LinkImport;
```

The preceding code shows IDL for the HTML Import where the parent interface is **LinkImport** which has the **readonly attribute import**. The **HTMLLinkElement** class implements the **LinkImport** parent interface. The browser that supports HTML Import must implement the preceding IDL.

**HTML Import feature detection**

The HTML Import is new to the browser and may not be supported by all browsers. To check the support of the HTML Import in the browser, we need to check for the **import** property that is present inside a **<link>** element. The code to check the HTML import support is as follows:

```
<!DOCTYPE html>
<html>
<head lang="en">
  <meta charset="UTF-8">
  <title>
    Web Component: HTML import support
  </title>
</head>
```

---

[12]
The preceding code has a `isImportSupported` function, which returns the Boolean value for HTML `import` support in the current browser. The function creates a `<link>` element and then checks the existence of an `import` attribute using the `in` operator. The following screenshot shows the output of the preceding code:

```
<body>
<h1 id="message"></h1>
<script>
  var isImportSupported = function () {
    var link = document.createElement("link");
    return 'import' in link;
  }
  var isSupported = isImportSupported(),
    message = document.getElementById("message");
  if (isSupported) {
    message.innerHTML = "Import is supported by the browser.";
  } else {
    message.innerHTML = "Import is not supported by the browser.";
  }
</script>
</body>
</html>
```

Import is supported by the browser.
Introducing Web Components

The preceding screenshot shows that the import is supported by the current browser as the `isImportSupported` method returns true.

The Can I Use tool can also be utilized for checking support for the HTML Import in the current browser. To check out the template support in the browser, use the following link:

http://caniuse.com/#feat=imports

The following screenshot shows the current status of support for the HTML Import in browsers using the Can I Use online tool:

![Can I Use Support Table](image)

Accessing the HTML Import document

The HTML Import includes the external document to the current page. We can access the external document content using the `import` property of the link element. In this section, we will learn how to use the `import` property to refer to the external document. The `message.html` file is an external HTML file document that needs to be imported. The content of the `message.html` file is as follows:

```html
<h1>
    This is from another HTML file document.
</h1>
```
The following code shows the HTML document where the message.html file is loaded and referenced by the import property:

```html
<!DOCTYPE html>
<html>
<head lang="en">
    <link rel="import" href="message.html">
</head>
<body>
<script>
    (function()
    {
        var externalDocument =
            document.querySelector('link[rel="import"]').import;
        headerElement = externalDocument.querySelector('h1')
            document.body.appendChild(headerElement.cloneNode(true));
    })();
</script>
</body>
</html>
```

The details of the preceding code are listed here:

- In the header section, the `<link>` element is importing the HTML document present inside the message.html file.
- In the body section, an inline `<script>` element using the `document.querySelector` method is referencing the link elements having the `rel` attribute with the `import` value. Once the link element is located, the content of this external document is copied using the `import` property to the `externalDocument` variable.
- The header `h1` element inside the external document is then located using a `querySelector` method and saved to the `headerElement` variable.
- The header element is then deep copied using the `cloneNode` method and appended to the `body` element of the current document.
The following screenshot shows the output of the preceding code:

![Screenshot](image.png)

This is from another HTML file document.

### HTML Import events

The HTML `<link>` element with the `import` attribute supports two event handlers. These two events are listed as follows:

- **load**: This event is fired when the external HTML file is imported successfully onto the current page. A JavaScript function can be attached to the `onload` attribute, which can be executed on a successful load of the external HTML file.

- **error**: This event is fired when the external HTML file is not loaded or found (HTTP code 404 not found). A JavaScript function can be attached to the `onerror` attribute, which can be executed on error of importing the external HTML file.
The following code shows the use of these two event types while importing the message.html file to the current page:

```html
<!DOCTYPE html>
<html>
<head lang="en">
  <script async>
    function handleSuccess(e) {
      // import load Successful
      var targetLink = e.target,
          externalDocument = targetLink.import;
      headerElement = externalDocument.querySelector('h1'),
      clonedHeaderElement = headerElement.cloneNode(true);
      document.body.appendChild(clonedHeaderElement);
    }
    function handleError(e) {
      // Error in load
      alert("error in import");
    }
  </script>
  <link rel="import" href="message.html"
        onload="handleSuccess(event)"
        onerror="handleError(event)"/>
</head>
<body>
</body>
</html>
```

The details of the preceding code are listed here:

- **handleSuccess**: This method is attached to the onload attribute which is executed on the successful load of message.html in the current document. The handleSuccess method imports the document present inside the message.html file, then it finds the h1 element, and makes a deep copy of it. The cloned h1 element then gets appended to the body element.

- **handleError**: This method is attached to the onerror attribute of the <link> element. This method will be executed if the message.html file is not found.
As the message.html file is imported successfully, the `handleSuccess` method gets executed and header element `h1` is rendered in the browser. The following screenshot shows the output of the preceding code:

![Screenshot](image)

**This is from another HTML file document.**

### Shadow DOM

Before the web component specification, there were many issues of building web applications using HTML, CSS, and JavaScript. Some of the issues are listed as follows:

- **Style override**: The document stylesheet may change the style of the web component.
- **Script alteration**: The document JavaScript may alter some part of the web component.
- **ID overlap**: There may be a duplicate ID present in the document, which can lead to many erroneous situations.

From the aforementioned issue list, there is clearly a problem with **scoping**. Shadow DOM is another important piece of web component specification that solves the scoping problem by the encapsulation mechanism. Shadow DOM provides a way of packaging the HTML, CSS, and JavaScript for a web component.
Most of the HTML5 elements, such as the progress bar, are implemented as Shadow DOM by the Chrome browser. We can inspect this Shadow DOM through the Chrome developer console. By default, the Chrome developer console will not show Shadow DOM. We need to enable the **Show user agent shadow DOM** checkbox present inside the settings of the developer console. The following screenshot shows the Chrome developer console setting to enable Shadow DOM inspection:

![Settings](image1)

After enabling the Shadow DOM inspection setting, we can inspect the `<progress>` HTML5 element. The following screenshot shows the Chrome developer inspection of the progress bar element containing Shadow DOM node:

![Elements](image2)
Introducing Web Components

In the preceding screenshot, we can see a new element #shadow-root. This node is the Shadow DOM of the progress bar element. As the progress bar is built in the browser element; we can see the user-agent text in parenthesis.

**Shadow DOM feature detection**

The Shadow DOM support for a browser can be checked by enabling the createShadowRoot property inside an element. The following code demonstrates a way of detecting the support of the Shadow DOM in the current browser:

```html
<!DOCTYPE html>
<html>
<head lang="en">
  <meta charset="UTF-8">
  <title>
    Web Component: Shadow DOM Feature detection
  </title>
</head>
<body>
  <h1 id="message"></h1>
  <script>
    var isShadowDOMSupported = function () {
      return "createShadowRoot" in document.body;
    };
    var isSupported = isShadowDOMSupported(),
        message = document.getElementById("message");
    if (isSupported) {
      message.innerHTML = "Shadow DOM is supported by the browser.";
    } else {
      message.innerHTML = "Shadow DOM is not supported by the browser.";
    }
  </script>
</body>
</html>
```

In the preceding code, the isShadowDOMSupported method checks the support of the Shadow DOM in the current browser by checking the existence of the createShadowRoot property in the document.body element. The following screenshot shows the output of the preceding code in the current browser:
The preceding screenshot shows that the Shadow DOM is supported by the current browser, as the `isShadowDOMSupport` method returns `true`. We can also check the support of the Shadow DOM using the Can I Use online tool. The following screenshot shows the status of Shadow DOM support in a different browser:
Shadow DOM brings the ability to include a subtree of DOM elements inside a document on the rendering time. The nodes inside DOM are organized as a tree structure. A node inside the DOM tree can have its own Shadow DOM tree. This makes the DOM a tree of trees. We can classify the DOM tree into three different types:

- **Document tree**: This represents the normal DOM tree whose root node is a document.
- **Shadow tree**: This represents the internal DOM subtree formed using HTML elements present inside shadow host. The root node of this tree is called shadow root.
- **Composed tree**: This represents the more expanded version of document tree, which includes the Shadow DOM trees too and is used by the browser for rendering.

The DOM element that has one or more than one Shadow DOM subtrees is called as **host element** or **shadow host**. The following diagram shows a sample DOM tree:

![](image)

In the preceding diagram, we find out that the node present inside the DOM element represents another subtree, which makes the DOM a tree of trees. A browser which supports Shadow DOM implementation should follow the IDL definition for declaring the shadow root element. The IDL of a shadow root element is listed in the following code:
interface ShadowRoot : DocumentFragment {
    HTMLElement getElementsByById(DOMString elementId);
    NodeList getElementsByClassName(DOMString className);
    NodeList getElementsByTagName(DOMString tagName);
    NodeList getElementsByTagNameNS(DOMString? namespace, DOMString localName);
    Selection? getSelection();
    Element? elementFromPoint(double x, double y);
    readonly attribute Element? activeElement;
    readonly attribute Element host;
    readonly attribute ShadowRoot? olderShadowRoot;
    attribute DOMString innerHTML;
    readonly attribute StyleSheetList styleSheets;
};

The details of the preceding IDL are listed here:

- **getElementById**: This method finds the element present inside the Shadow DOM tree with the given ID
- **getElementsByClassName**: This method finds the element present inside the Shadow DOM tree with the given class name
- **getElementsByTagName**: This method finds the element present inside the Shadow DOM tree with the given tag name
- **getElementsByTagNameNS**: This method finds the element present inside the Shadow DOM tree with the given namespace and tag name
- **getSelection**: This method returns the selection object for currently selected element inside the Shadow DOM tree
- **elementFromPoint**: This method returns the element with the given x and y coordinates
- **activeElement**: This property returns currently focused element inside the Shadow DOM tree
- **host**: This property returns the shadow host element
- **olderShadowRoot**: If the element has multiple shadow trees then this property returns the shadow root which was created earlier
- **innerHTML**: This property returns the HTML content of the shadow root as a string
- **styleSheets**: This property returns the list of stylesheet objects if the shadow tree contains the `<style>` element
Introducing Web Components

Now, let's check out an example which demonstrates the use of these properties and the methods of a shadow root. The example code is listed as follows:

```html
<!DOCTYPE html>
<html>
<head lang="en">
    <meta charset="UTF-8">
    <title>Shadow Root: Method & Properties example</title>
</head>
<body>
    <div id="aShadowHost"></div>
    <template id="selectorTemplate">
        <style>
            :host input{
                background: lightyellow;
            }
            :host .labelClass{
                color: blue;
            }
        </style>
        <form>
            <label for="nameElement" class="labelClass">Name</label>
            <input type="text" id="nameElement" placeholder="Enter your name" value="Sandeep" autofocus>
        </form>
    </template>
    <script>
        (function(){
            var aShadowHost = document.getElementById("aShadowHost"),
                shadowRoot1 = aShadowHost.createShadowRoot(),
                shadowRoot2 = aShadowHost.createShadowRoot(),
                templateContent = document.querySelector('#selectorTemplate').content,
                templateNodes = document.importNode(templateContent, true);
            shadowRoot1.innerText = "inside shadowRoot1";
            shadowRoot2.appendChild(templateNodes);
            shadowRoot2.getElementById("nameElement").select();
            //Shadow Root Methods
            console.log("getElementById: ",shadowRoot2.getElementById("nameElement"));
            console.log("getElementsByClassName: ",shadowRoot2.getElementsByTagName("label"));
        }
    )
    </script>
</body>
</html>
```
In the preceding code, the two Shadow DOM subtrees `shadowRoot1` and `shadowRoot2` are present for the host element. The `shadowRoot1` subtree is created first and `shadowRoot2` is created later. Hence, the `shadowRoot1` subtree is an older shadow root. The `shadowRoot2` subtree contains the HTML markup from a template with the `selectorTemplate` ID. The `shadowRoot2` subtree has a `<form>` element containing a `<label>` and `<input>` element. It also contains some CSS styles inside the `<style>` element. The output of the preceding code is presented in the following screenshot:
Introducing Web Components

The following screenshot shows the console log messages, which demonstrate the use of the preceding methods for the shadow tree:

```
console.log(getElementById('nameElement')); // Returns the element with id 'nameElement'
console.log(getElementsByClassName('label', 'labelClass')); // Returns elements with class 'labelClass'
console.log(getElementsByTagName('label', 'labelClass')); // Returns elements with class 'labelClass'
console.log(getElementsByTagNames('label', 'labelClass')); // Returns elements with class 'labelClass'
console.log(getElementsByTagNamesNS('label', 'xml', 'labelClass')); // Returns elements with class 'labelClass'
console.log(getSelection().toString()); // Returns selection string
console.log(elementFromPoint(x, y)); // Returns element at point (x, y)
```

The following screenshot shows the console log messages that demonstrate the use of the preceding properties for the shadow tree:

```
const host = document.querySelector('#aShadowHost'); // Get host element
const shadowRoot = host.shadowRoot; // Get shadow root
const styleSheet = shadowRoot.styleSheets[0]; // Get style sheet

console.log(styleSheet.href); // Returns style sheet URL
console.log(styleSheet.cssRules[0].selectorText); // Returns style rule selector
```
Custom element

Web component specifications come with the power to create a new element for DOM. A custom element can have its own properties and methods. The reasons for creating a custom element are less code from the developer's point of view, creating a more semantic tag library, reducing the number of div tags, and so on. Once a web component is developed, it can be used by any application.

Custom element feature detection

A new element can be registered to DOM using the registerElement method. We can detect the support of the custom element in the current browser by checking the presence of the registerElement function inside document. The following JavaScript code shows a method to detect the support for custom element:

```html
<!DOCTYPE html>
<html>
<head lang="en">
    <meta charset="UTF-8">
    <title>Web Component: custom element support</title>
</head>
<body>
<h1 id="message"></h1>
<script>
    var isCustomElementSupported = function () {
        return 'registerElement' in document;
    };
    var isSupported = isCustomElementSupported(),
        message = document.getElementById("message");
    if (isSupported) {
        message.innerHTML = "Custom element is supported by the browser.;"
    } else {
        message.innerHTML = "Custom element is not supported by the browser.;"
    }
</script>
</body>
</html>
```
Introducing Web Components

In the preceding code, the `isCustomElementSupported` method has the code to check the custom element support. It uses the `in` operator to check whether the `registerElement` function is present inside the document object. If the custom element is supported, the method returns true and the success message gets rendered in the browser. The following screenshot shows the output of the preceding code in the browser:

![Custom element is supported by the browser.](image)

We can also use the Can I Use online tool to check the support for custom elements. The following screenshot shows the current status of the browser for custom element support:

![Can I Use support table](image)
Developing a custom element

In this section, we will develop a custom element and understand each step in detail. The steps involved in developing a custom element are listed here:

- Creating a new object
- Defining object properties
- Defining lifecycle methods
- Registering a new element
- Extending an element

Creating a new object

A new object can be created using the Object.create method. The syntax of this method is listed here:

```
Object.create(<target prototype> [, propertiesObject]);
```

The Object.create method takes two parameters. The first parameter is the target prototype of the newly created object. The second parameter contains the properties of the newly created object. The second parameter is optional. The following code defines a new object:

```
var objectPrototype = Object.create(HTMLElement.prototype);
```

In the preceding code, a new object is created that has the HTMLElement.prototype parameter and is saved in the objectPrototype variable.

To find out more about the Object.create method, use the following link:


Defining object properties

We can define the property of an object using two different methods defineProperty and defineProperties. The defineProperty method is used to create a single property, and the defineProperties method for multiple properties. The syntax of these methods is listed here:

```
Object.defineProperty(<targetObject>, <PropertyName>,
<propertySettings>);
Object.defineProperties(<targetObject>, <properties>);
```
The details of the preceding syntax are listed as follows:

- **targetObject**: This represents the target object for which the property needs to be defined.
- **propertyName**: This represents the key of the property.
- **propertySettings**: This represents all the configuration options for a property. The possible settings options are listed here:
  - **configurable**: This takes a Boolean value. For a `true` value, the type of property can be changed or deleted. For a `false` value, the property type cannot be changed and deleted.
  - **enumerable**: This takes a Boolean value. For a `true` value, the property will be enumerated as its own property.
  - **value**: This takes any JavaScript value. It represents the value associated with the property.
  - **writable**: This takes a Boolean value. For a `true` value, the associated value of the property can be updated using assignment operator.
  - **get**: This takes a function. It returns the value of the property.
  - **set**: This takes a function. It sets the input value to the property.

The following code shows an example of defining a single property named `title` for `newObject` that is writable:

```javascript
var newObject = Object.create(HTMLElement.prototype);
Object.defineProperty(newObject, 'title', {
  writable : true
});
```

To find out more about the `Object.defineProperty` method, use the following link:


The following code shows an example of defining multiple properties like `title` and `country` for the `newObject` variable. The `title` property is writable, and the `country` property is not writable and has a fixed value `India`:

```javascript
var newObject = Object.create(HTMLElement.prototype);
Object.defineProperties(newObject, {
  title: {
    writable: true
  }
});
```

```javascript
```
To find out more about the `Object.defineProperties` method, use the following link:

### Defining lifecycle methods

An object in JavaScript goes through different states during its lifecycle. The different states of an object lifecycle are listed here:

- **created**: An object is in the `created` state when it is initialized. The event handler for this state is the `createdCallback` method.
- **attached**: An object is in the `attached` state when it is inserted to the DOM. The event handler for this state is the `attachedCallback` method.
- **detached**: An object is in the `detached` state when it is removed from the DOM. The event handler for this state is the `detachedCallback` method.
- **attributeChanged**: An object is in the `attributeChanged` state when one of its property's values is updated. The event handler for this state is the `attributeChangedCallback` method.

The following code shows an example where an object is created using the `Object.create` method, and a callback method is attached for the created state:

```javascript
var objectPrototype = Object.create(HTMLElement.prototype);
objectPrototype.createdCallback = function(){
    console.log("Instance is created");
};
```

### Registering a new element

A new element can be registered to the DOM using the `document.registerElement` method. The syntax of this method is listed here:

```javascript
var constructor = document.registerElement(<tag-name>, settings);
```
Introducing Web Components

The details of the preceding syntax are listed as follows:

- **tag-name**: This represents the name of the custom element. The name must be separated with a hyphen.
- **settings**: This takes the configuration option for the custom element.
- **constructor**: The `registerElement` method returns the constructor of new element.

The following code shows an example of registering a new element named `welcome-message` to the DOM. The prototype of the `welcome-message` element is `objectPrototype`, which is created using the `Object.create` method:

```javascript
var objectPrototype = Object.create(HTMLElement.prototype),
    welcomeElement = document.registerElement("welcome-message",{
        prototype: objectPrototype
    });
```

To find out more about the `document.registerElement` method, use the following link:

---

### Extending an element

An element can inherit a native or another custom element. The `extend` property is used to inherit another element. The following code shows an example of extending an `<i>` element:

```javascript
var objectPrototype = Object.create(HTMLElement.prototype),
    italicElement = document.registerElement("italic-message",{
        prototype: objectPrototype,
        extends: 'i'
    });
```

The `is` operator is used to define the type of an HTML element. The following code shows if an element is of the italic type:

```html
<welcome-message is="i">
    Hello world
</welcome-message>
```
Example of a custom element

In this section, we will create a simple custom element named `<my-message>`.

Code for the `<my-message>` element is as follows:

```html
<!DOCTYPE html>
<html>
<head lang="en">
    <meta charset="UTF-8">
    <title>Web Component: custom element example</title>
    <script>
        var objectPrototype = Object.create(HTMLElement.prototype);
        Object.defineProperty(objectPrototype, 'title', {
            writable: true
        });
        objectPrototype.createdCallback = function() {
            this.innerText = this.title;
        };
        var myNameElement = document.registerElement("my-name", {
            prototype: objectPrototype
        });
    </script>
</head>
<body>
    <my-name title="Welcome to custom element 1"></my-name>
    <br>
    <my-name title="Welcome to custom element 2"></my-name>
</body>
</html>
```

Downloading the example code

You can download the example code files from your account at http://www.packtpub.com for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.
Introducing Web Components

In the preceding code, a custom my-name element is defined using the registerElement method. It has the title attribute, which has been defined using the Object.defineProperty method. A createdCallback method is added, which takes the input string of the title property and inserts it using the innerText property. The following screenshot shows the output of the preceding code:

![Screenshot showing the output of the code]

Welcome to custom element 1
Welcome to custom element 2

Node distribution

The composed tree takes part in rendering the DOM inside the browser. The Shadow DOM subtree of the nodes gets arranged for display. The arrangements of the nodes are done by a distribution mechanism with the help of specific insertion points. These insertion points are of two types:

- Content insertion point
- Shadow insertion point

A content insertion point

A content insertion point is a placeholder for child nodes of the shadow host distribution. It works like a marker, which reprojects the child nodes of the shadow host. A content insertion point can be defined using the <content> element. The <content> element has a select attribute through which we can filter out the reprojection.
The following code gives an example of the use of the `<content>` element with the `select` attribute:

```html
<!DOCTYPE html>
<html>
  <head lang="en">
    <meta charset="UTF-8">
    <title>Web Component: content insertion point with select attribute example</title>
    <template id="selectorTemplate">
      <style>
        :host b{
          margin: 0px 10px;
        }
        :host ::content b.fruit{
          color:green;
        }
        :host ::content b.flower{
          color:orange;
        }
      </style>
      <h1 class="fruit">Fruits <content select="b.fruit"></content>.</h1>
      <h1 class="flower">Flowers <content select="b.flower"></content>.</h1>
    </template>
    <script>
      var objectPrototype = Object.create(HTMLElement.prototype);
      objectPrototype.createdCallback = function(){
        var shadow = this.createShadowRoot(),
        templateContent = document.querySelector('#selectorTemplate').content,
        templateNodes = document.importNode(templateContent, true);
        shadow.appendChild(templateNodes);
      };
      var myNameElement = document.registerElement("selector-component",{
        prototype: objectPrototype
      });
    </script>
  </head>
  <body>
    <selector-component>
      <b class="fruit">Apple</b> <b class="flower">Rose</b> <b class="fruit">Orange</b>
    </selector-component>
  </body>
</html>
```
Introducing Web Components

A detailed explanation of the preceding code is listed here:

- A custom element named `selector-component` is created, which has a list of fruits and flowers.
- The HTML template of the custom element has two `<content>` elements. One content element filters out all the flowers using the `select` attribute with the `b.flower` value, and the other `<content>` element filters out all the fruits using the `select` attribute with the `b.fruit` value.

The following screenshot shows the output of the preceding code of filtering fruit and flower in a separate group:
A shadow insertion point

Shadow insertion points are placeholders for other shadow trees. This insertion point reprojects the elements of other shadow trees. A shadow insertion point can be created using the `<shadow>` element. The following code gives an example of the use of the shadow insertion point:

```html
<!DOCTYPE html>
<html>
<head lang="en">
  <meta charset="UTF-8">
  <title>Web Component: shadow insertion point example</title>
</head>
<body>
  <div id="aShadowHost"></div>
  <template id="shadow1Template">
    <button>Shadow Root 1 Button</button>
  </template>
  <template id="shadow2Template">
    <fieldset>
      <legend>Shadow Root 2</legend>
      <shadow></shadow>
    </fieldset>
  </template>
  <script>
    // Old shadow root
    var aShadowHost = document.getElementById("aShadowHost"),
    aShadowRoot1 = aShadowHost.createShadowRoot();
    templateContent = document.querySelector('#shadow1Template').content,
    templateNodes = document.importNode(templateContent, true);
    aShadowRoot1.appendChild(templateNodes);
    // New shadow root with insertion point for older shadow root
    var aShadowRoot2 = aShadowHost.createShadowRoot();
    templateContent = document.querySelector('#shadow2Template').content,
    templateNodes = document.importNode(templateContent, true);
    aShadowRoot2.appendChild(templateNodes);
  </script>
</body>
</html>
```
The details of the preceding code are listed here:

- There are two shadow roots, `shadowRoot1(old)` and `shadowRoot2(new)`, created for the `<div>` element with the `aShadowHost` ID.

- The `shadow1Template` is the HTML template for `shadowRoot1`, and `shadow2Template` is the HTML template for `shadow2Root`.

- The `shadow1Template` contains a `<button>` element, and `shadow2Template` contains a `<fieldset>` and `<legend>` element. The `<fieldset>` element also has a `<shadow>` insertion point.

- During rendering of the page, the shadow insertion point will take the older shadow root content and insert it in the shadow insertion point.

The following screenshot shows the output of the preceding code, where the older shadow root elements are reprojected and rendered inside the `<fieldset>` element, which belongs to the younger shadow root, that is, `shadowRoot1`. 
Styling web components

The way we styled the HTML DOM elements earlier needs to be changed with the emergence of the web component specification. In this section, we will explore some of the key areas that need more focus while authoring CSS. We need to know some new pseudo element selectors for styling the web component. These pseudo selectors are listed here:

- **Unresolved pseudo selector**: When a custom element is loaded and registered with the DOM, the browser picks the matched element and upgrades it based on the defined lifecycle. During this upgradation process, the elements are exposed to the browser and appear as unstyled for a few moments. We can avoid the flash of unstyled content using the :unresolved pseudo class. An example of unresolved pseudo selector for the `<header-message>` custom element are listed here:

  ```css
  header-message:unresolved:after {
    content: 'Registering Element...';
    color: red;
  }
  ```

- **Host pseudo selector**: The custom element itself can be referred using the :host pseudo selector to apply the style attribute. An example of the host selector is listed in the following code:

  ```css
  :host{
    text-transform: uppercase;
  }
  ```

- **Shadow pseudo selector**: The Shadow DOM subtree of the custom element can be referred using the ::shadow pseudo selector to apply the style attributes. An example of shadow selector is listed here:

  ```css
  :host ::shadow h1{
    color: orange;
  }
  ```

- **Content pseudo selector**: The content of the older insertion point element can be referred using the ::content pseudo selector to apply the style attributes. An example of content selector is listed in the following code:

  ```css
  :host ::content b{
    color: blue;
  }
  ```
Introducing Web Components

Let's check out a simple example to demonstrate the aforementioned pseudo selectors. The following code creates a custom element named `<header-element>`. To show the use of the :unresolved pseudo selector, we delayed registering the custom element for 3 seconds using the `window.setTimeout` method.

```html
<!DOCTYPE html>
<html>
<head lang="en">
  <meta charset="UTF-8">
  <title>Web Component: Unresolved pseudo selector</title>
  <style>
    header-element:unresolved{
      visibility: hidden;
    }
    header-element:unresolved:after {
      content: 'Registering Element...';
      color: red;
      visibility: visible;
    }
  </style>
  <template id="headerTemplate">
    <style>
      :host {
        text-transform: uppercase;
      }
      :host::shadow h1{
        color:orange;
      }
      :host ::content b{
        font-style: italic;
        color:blue;
      }
    </style>
    <h1>Hello <content></content></h1>
  </template>
  <script>
    (function(){
      var objectPrototype = Object.create(HTMLElement.prototype);
      objectPrototype.createdCallback=function(){
        var shadow = this.createShadowRoot(),
            templateContent =
            document.querySelector('#headerTemplate').content,
            templateNodes =
            document.importNode(templateContent, true);
    
```
The details of the preceding code are listed here:

- The registration process of the custom element is delayed on purpose for 3 seconds. During this time, the element becomes `HTMLUnknownElement`. We used the `:unresolved` pseudo selector to show a Registering Element... message during this time in the color red.
- Once the element is registered, the custom element becomes resolved (`HTMLElement`). In the `createdCallback` lifecycle method, we created a shadow root appended as a child.
- The template of `<header-element>` is present inside the `<template>` element with the `headerTemplate` ID. The template is then activated using the `document.importNode` method, which are added as children of the preceding shadow root.
- The host DOM tree is referred using the `:host` pseudo selector, which has a `style` attribute in order to transform the text into capital letters.
- The Shadow DOM tree is referred using the `::shadow` pseudo selector, which has a `style` attribute to change the text color to orange.
- The template also has the `<content>` element, which selects the original children of `<header-element>` and puts it into this location. In our example, the children are wrapped around the `<b>` tag. We referred this `<b>` element using the content selector to apply the `style` attribute so as to make the text color blue and the text type italic.
Introducing Web Components

The following screenshot shows the output of the preceding code with the :unresolved pseudo selector style in effect for the first 3 seconds. We can see the message in red.

Once the element is registered to the DOM, the lifecycle method gets executed and `<header-element>` gets upgraded with its Shadow DOM. The following screenshot shows the final output of the preceding code:
Building a digital clock component
In this section, we will build a simple digital clock element. The motive behind building a custom component is to implement the template, HTML Imports, Shadow DOM, and custom element to a real-time example. The definition of the digital clock component is present in the clock-element.html file, and the use of the digital clock component is present in the clock-demo.html file. The clock-element.html file has two sections. These are listed as follows:

- Clock template
- Clock element registration script

Clock template
The digital clock template contains the HTML markup and the CSS styles for rendering in the browser on activation. The HTML template code and the CSS styles for the clock component are listed in the following code:

```html
<template id="clockTemplate">
  <style>
    :host::shadow .clock {
      display: inline-flex;
      justify-content: space-around;
      background: white;
      font-size: 8rem;
      box-shadow: 2px 2px 4px -1px grey;
      border: 1px solid green;
      font-family: Helvetica, sans-serif;
      width: 100%;
    }
    :host::shadow .clock .hour,
    :host::shadow .clock .minute,
    :host::shadow .clock .second {
      color: orange;
      padding: 1.5rem;
      text-shadow: 0px 2px black;
    }
  </style>
  <div class="clock">
    <div class="hour">HH</div>
    <div class="minute">MM</div>
    <div class="second">SS</div>
  </div>
</template>
```

[43]
A detailed explanation of the preceding code is listed here:

- The content of the clock element is present inside the `<template>` element. The ID of the template element is `clockTemplate`.
- This template contains two section styles and HTML markup.
- All the CSS style classes are wrapped around the `<style>` element. The host clock element is targeted using the `:host` pseudo selector, and its shadow tree children are targeted using the `::shadow` pseudo attribute and the styles are applied.
- The HTML markup for the clock element is wrapped around the `div` element. The parent `div` element has the `.clock` class. The parent `div` element has the three children `div` element representing hours, minutes, and seconds.

**Clock element registration script**

The clock component registration script is present in the `clock-element.html` file and is wrapped around a self-invoking anonymous function. The JavaScript code to create and register a clock component is listed in the following code:

```javascript
<script>
    (function() {
        var selfDocument = document.currentScript.ownerDocument,
            objectPrototype = Object.create(HTMLElement.prototype);
        objectPrototype.createdCallback = function() {
            var shadow = this.createShadowRoot(),
                templateContent = selfDocument.querySelector('#clockTemplate').content,
                templateNodes = document.importNode(templateContent, true),
                hourElement = null,
                minuteElement = null,
                secondElement = null;
            shadow.appendChild(templateNodes);
            hourElement = shadow.querySelector('.hour'),
            minuteElement = shadow.querySelector('.minute'),
            secondElement = shadow.querySelector('.second');
            window.setInterval(function() {
                var date = new Date();
                hourElement.innerText = date.getHours();
                minuteElement.innerText = date.getMinutes();
                secondElement.innerText = date.getSeconds();
            }, 1000);
        }
    })();
</script>
```
The details of the preceding code are listed here:

- The script for registering the clock element is embedded inside a self-calling function, which saves the reference to the current owner document to selfDocument variable using document.currentScript.ownerDocument.
- A new object is created using the Object.create method. The prototype of this new object is HTMLElement.prototype. The reference of this new object is saved in the objectPrototype variable.
- The createdCallback lifecycle method of the host element is overloaded with the following steps:
  - A new shadowRoot object is created for the host element using the createShadowRoot method. Reference to this shadowRoot is then saved to the shadow variable.
  - The template content of the clock element is then retrieved using the selfDocument reference variable.
  - The inert content of the clock template is then activated using the document.importNode method.
  - The activated template contents are then added as children to the host's shadow root.
  - Using window.setInterval(), a block of code is called every 1 second. The purpose of this code block is to get the hours, minutes, and seconds of the current time and update the DOM repeatedly every second.
- The clock element is then registered with the DOM using the document.registerElement method. After registering, the clock component is now ready for use.
Introducing Web Components

Using the clock component
In the previous section, we developed the clock component that is present inside the clock-element.html file. In this section, we will import the clock element and use it in the markup to render in the browser. The code to use clock component is present in the clock-demo.html file and is listed here:

```html
<!DOCTYPE html>
<html>
<head lang="en">
  <meta charset="UTF-8">
  <title>Web Component : digital clock element</title>
  <link rel="import" href="clock-element.html">
</head>
<body>
  <digital-clock></digital-clock>
</body>
</html>
```

In the preceding code, the clock component is imported using the `link` element with the `rel` attribute, which has the `import` value. The digital clock component can be implemented using the `<digital-clock> </digital-clock>` custom element. The output of the preceding code is shown in the following screenshot:

![Digital Clock Screenshot]

The preceding screenshot shows the digital clock component. The numbers in the screenshot are showing `hours` (HH), `minutes` (MM), and `seconds` (SS). The following screenshot shows the developer console of the clock component:
The details of the preceding screenshot are listed here:

- The clock element is imported to the current page and has its own #document root.
- The digital clock element has its Shadow DOM tree, which is rendered as a clock.

**X-Tag**

The X-Tag is a small JavaScript library for web component development by Mozilla. This library is built on the web component polyfill from Polymer team. The Mozilla Bricks framework is built on top of the X-Tag library. We can download the X-Tag library using [http://www.x-tags.org/download](http://www.x-tags.org/download).

**X-Tag element lifecycle**

Every X-Tag element has a lifecycle. An element state is decided based on the event that is fired during state transition. An element during its lifecycle goes through the following states (event fired):

- **created**: This event is fired by the element when it is initially created.
- **inserted**: This event is fired by the element when it is inserted into the DOM for first time.
Introducing Web Components

- **removed**: This event is fired by the element when it is removed from the DOM.
- **attributeChanged**: This event is fired when any of the property values of the element is changed.

The lifecycle of the element can be defined inside the `lifecycle` attribute. The following code shows the syntax of the `lifecycle` attribute:

```javascript
lifecycle:
  created: function() {
    // code for created state
  },
  inserted: function() {
    // code for inserted state
  },
  removed: function() {
    // code for removed state
  },
  attributeChanged: function() {
    // code for attributeChanged state
  }
```

**X-Tag custom element development**

A custom X-Tag element can be created using the `xtag.register` method. The X-Tag core library code is present inside the `x-tag-components.js` file.

The `xtag.register` method has the following syntax:

```javascript
xtag.register('<element-name>', {
  lifecycle: {
    created: function() {
      // code for created state
    },
    inserted: function() {
      // code for inserted state
    },
    removed: function() {
      // code for removed state
    }
  }
```

[The X-Tag core library source code can be downloaded by visiting: https://github.com/x-tag/core](https://github.com/x-tag/core)
The details of the preceding syntax are listed here:

- **lifecycle**: This property can have code for all states during the lifecycle of the element. Therefore, we can define its logic for the custom elements by implementing the `created`, `inserted`, `removed`, and `attributeChanged` state.
- **methods**: This property can have all the methods that need to be exposed as a public API that is to be consumed externally.
- **events**: This property can have all the element's event binding listeners that need to be fired based on the user action of the custom element.
- **accessors**: This property can have all the attributes that need the getter and setters methods.

Now, it is time to create a custom component using this X-Tag library. The code for creating an X-Tag base custom element is as follows:

```html
<!DOCTYPE html>
<html>
<head lang="en">
  <meta charset="UTF-8">
</head>
<body>
  Your custom element here
</body>
</html>
```
Introducing Web Components

<title>Web Component: xTag custom element support</title>
<script src="x-tag-components.js"></script>

(function() {
  xtag.register('italic-string', {
    lifecycle: {
      created: function() {
        this.innerHTML = '<i style="color:' + this.textColor + '">' + this.innerHTML + '</i>';
      },
    },
    accessors: {
      textColor: {
        attribute: {object: this.textColor}
      },
    },
    methods: {
      changeToRed: function() {
        var italicElement = this.querySelector("i");
        italicElement.style.color = "red";
      },
    },
    events: {
      'click:delegate(i)': function(e) {
        console.log("click event is fired.");
      },
    }
  });
}());
</script>
</head>
<body>
<italic-string id="iStringComponent" textColor="blue">
  Click Me
</italic-string>
<br>
<button onclick="doColorRed()">Make Red</button>

<script>
  var doColorRed = function() {
    var italicStringElement = document.getElementById("iStringComponent");
    italicStringElement.changeToRed();
  }
</script>
</body>
</html>
The details of the preceding code are listed here:

- A custom X-Tag-based element named `italic-string` is created by the `xtag.register` method.
- This custom element takes the `innerHTML` content and wraps it with a `<i>` element, which gives it an italic style font.
- This custom element has a `textColor` property name, where a color string can be given. The value of the `textColor` property is then applied to the `style` property of the `<i>` element.
- The `textColor` property is created using the `accessors` property. This `accessors` property takes the attributes that need to be configured to the `italic-string` element.
- An event listener is created using the `events` property. In the preceding code a `click` event type listener is attached to the `<i>` element. When the `<i>` element is clicked on, it shows a message in the console.
- A method can be defined using the `methods` property. There is a method callback `changedToRed()` that can be accessed as an API. This callback method has used `document.getElementById()` to locate the X-Tag custom element with the `iStringComponent` (the `italic-string` component). It then finds and changes the `color` style attribute of the `<i>` element to Red. A button's `onclick` method is attached with a `doColorRed` JavaScript function, which in turn calls the `changeToRed` method.

The output of the preceding code looks like the following screenshot. It has the **Click Me** text and a **Make Red** button rendered in the browser:
Introducing Web Components

When user clicks on the **Make Red** button, the **Click Me** text will change to red in color. The following screenshot shows the **Click Me** text changed to red:

![Screenshot of Click Me text changed to red](image)

If the user clicks on the **Click Me** text, then the event handler attached with it gets executed and prints the message. The following screenshot shows the console log message when the user clicks on the X-Tag element:

![Screenshot of console log message](image)

To know more about X-Tag library use the following link:

http://www.x-tags.org/docs

Web component specification is not completely implemented by the browsers. However, there are many libraries with polyfill support for web components that exist. In this section, we will list the libraries, and get a quick introduction to them. Some of the most popular libraries are listed here:
Polymer
Polymer is the web component library from Google Inc. This library allows a web developer to compose CSS, HTML, and JavaScript to build rich, powerful, and reusable web component. In Chapter 2, Introducing Polymer and Chapter 3, Developing Web Components Using Polymer, we will learn more about this library.

To find out more about Polymer library use the following link:
https://www.polymer-project.org

Mozilla Brick
Mozilla Brick is another web component library from Mozilla. It has a collection of reusable UI components to be used in web application. The current version of this library is 2.0. In Chapter 5, Developing Web Components Using Mozilla Brick, we will learn more about this library.

To find out more about Mozilla Brick library use the following link:
http://brick.readme.io/v2.0

ReactJS
The ReactJS is a library for web component development from Facebook. This library takes a different approach to build the web application. In Chapter 6, Building Web Components with ReactJS, we will learn more about the ReactJS library.

To find out more about ReactJS library, use the following link:
http://facebook.github.io/react

Bosonic
Bosonic is another library for web component development. It uses some of the PolymerJS polyfill in the core. In Chapter 4, Exploring Bosonic Tools for Web Component Development, we will explore more details about Bosonic.

To find out more about the Bosonic library, use the following link:
http://bosonic.github.io/index.html
Summary
In this chapter, we learned about the web component specification. We also explored the building blocks of web components such as Shadow DOM, custom element, HTML Imports, and templates. In the next chapter, we will learn about the PolymerJS library in detail.
Where to buy this book