



Community Experience Distilled

# Learning Web Component Development

Discover the potential of web components using PolymerJS,  
Mozilla Brick, Bosonic, and ReactJS

Sandeep Kumar Patel

[PACKT] open source PUBLISHING

# Learning Web Component Development

---

# Table of Contents

[Learning Web Component Development](#)

[Credits](#)

[About the Author](#)

[About the Reviewers](#)

[www.PacktPub.com](#)

[Support files, eBooks, discount offers, and more](#)

[Why subscribe?](#)

[Free access for Packt account holders](#)

[Preface](#)

[What this book covers](#)

[What you need for this book](#)

[Who this book is for](#)

[Conventions](#)

[Reader feedback](#)

[Customer support](#)

[Downloading the example code](#)

[Errata](#)

[Piracy](#)

[Questions](#)

[1. Introducing Web Components](#)

[What are web components?](#)

[Benefits and challenges of web components](#)

[The web component architecture](#)

[Template element](#)

[Template element detail](#)

[Template feature detection](#)

[Inert template](#)

[Activating a template](#)

[Cloning a node](#)

[Importing a node](#)

[HTML Import](#)

[HTML Import feature detection](#)

[Accessing the HTML Import document](#)

## HTML Import events

### Shadow DOM

#### Shadow DOM feature detection

#### Shadow tree

### Custom element

#### Custom element feature detection

#### Developing a custom element

##### Creating a new object

##### Defining object properties

##### Defining lifecycle methods

##### Registering a new element

##### Extending an element

#### Example of a custom element

### Node distribution

#### A content insertion point

#### A shadow insertion point

### Styling web components

### Building a digital clock component

#### Clock template

#### Clock element registration script

### Using the clock component

### X-Tag

#### X-Tag element lifecycle

#### X-Tag custom element development

#### Polymer

#### Mozilla Brick

#### ReactJS

#### Bosonic

### Summary

## 2. Introducing Polymer

### What is Polymer?

### Installing and configuring Polymer

#### Downloading ZIP file

#### Using GIT clone

#### Using Bower

### Architecture of PolymerJS

### Web components with polyfill

## [The Polymer library](#)

### [Elements](#)

#### [Core elements](#)

[The core-input element](#)

[The core-label element](#)

[The core-tooltip element](#)

#### [Paper elements](#)

[Material design](#)

[The paper-checkbox element](#)

[The paper-slider element](#)

[The paper-button element](#)

### [Polymer designer tool](#)

[Developing with the designer tool](#)

[Getting a GitHub token](#)

[Developing an e-mail subscription form](#)

### [Yeoman Polymer generator](#)

[The polymer-generator commands](#)

[The Polymer application generator](#)

[The Polymer element generator](#)

[The Polymer seed generator](#)

[The Polymer GitHub page generator](#)

### [Summary](#)

## [3. Developing Web Components Using Polymer](#)

### [PolymerJS ready event](#)

### [Polymer expressions](#)

[Polymer templating with auto-binding](#)

[Polymer template attributes](#)

[Filtering expression](#)

[Built-in filtering expression](#)

[The TokenList filter](#)

[The styleObject filter](#)

[Custom filtering expression](#)

[Global filtering expression](#)

### [Developing Polymer custom elements](#)

[Defining a custom element](#)

[Defining element attributes](#)

[Defining default attributes](#)

[Defining public properties and methods](#)

[Publishing properties](#)

[Defining a lifecycle method](#)

[Registering a custom element](#)

[Developing a sample custom element](#)

[Extending a custom element](#)

[Polymer methods](#)

[The Polymer mixin method](#)

[The Polymer import method](#)

[The Polymer waitingFor method](#)

[The Polymer forceReady method](#)

[Asynchronous task execution](#)

[Developing a digital clock](#)

[Working with Yeoman](#)

[Yeoman element generator](#)

[Yeoman seed generator](#)

[Yeoman GitHub page generator](#)

[Preparing for production using vulcanize](#)

[Vulcanize installation](#)

[Running vulcanize process](#)

[Summary](#)

## [4. Exploring Bosonic Tools for Web Component Development](#)

[What is Bosonic?](#)

[Browser support](#)

[Configuring Bosonic](#)

[Bosonic packages](#)

[Built-in elements](#)

[The b-sortable element](#)

[The b-toggle-button element](#)

[Developing custom component](#)

[Step 1 – creating the red-message element directory](#)

[Step 2 – changing the current directory to red-message](#)

[Step 3 – generating the skeleton for <red-message>](#)

[Step 4 – verifying the directory structure](#)

[Step 5 – defining code for the <red-message> element](#)

[Step 6 – modifying the index.html demo file](#)

[Step 7 – generating distribution files using Grunt](#)

[Step 8 – running the index.html file](#)

[Bosonic lifecycle](#)

[Example of lifecycle](#)

[Digital clock development](#)

[Summary](#)

## [5. Developing Web Components Using Mozilla Brick](#)

[What is the Brick library?](#)

[Mozilla Brick 1.0](#)

[Mozilla Brick 2.0](#)

[Installing Mozilla Brick](#)

[Configuring Mozilla Brick](#)

[Built-in components](#)

[The brick-calendar element](#)

[The brick-flipbox element](#)

[The brick-deck element](#)

[The brick-tabbar element](#)

[The brick-action element](#)

[The brick-menu element](#)

[The X-Tag library](#)

[Developing a digital clock using X-Tag](#)

[Summary](#)

## [6. Building Web Components with ReactJS](#)

[The reactive approach](#)

[The flux architecture](#)

[Flux key features](#)

[Installing ReactJS](#)

[Configuring ReactJS](#)

[Using ReactJS](#)

[What is JSX](#)

[Custom components with JSX](#)

[ReactJS inline style](#)

[ReactJS event handling](#)

[Useful non-DOM attributes](#)

[ReactJS component lifecycle](#)

[ReactJS initialization phase](#)

[ReactJS lifetime phase](#)

[ReactJS teardown phase](#)

[ReactJS lifecycle example](#)

[Stateful custom component](#)

[Precompiled JSX for production](#)

[JSX file watcher](#)

[Developing a digital clock using ReactJS](#)

[Step1 – defining the digital clock lifecycle script](#)

[Step2 – defining CSS styles for the digital clock](#)

[Debugging ReactJS](#)

[Summary](#)

## [A. Web Component References](#)

[Chapter 1](#)

[Chapter 2](#)

[Chapter 3](#)

[Chapter 4](#)

[Chapter 5](#)

[Chapter 6](#)

[Index](#)

# Learning Web Component Development

---

# Learning Web Component Development

Copyright © 2015 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: May 2015

Production reference: 1180515

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78439-364-9

[www.packtpub.com](http://www.packtpub.com)

# Credits

## Author

Sandeep Kumar Patel

## Reviewers

Zhelan Chen

Krzysztof Cislo

Lars Kappert

Sebastian Metzger

## Commissioning Editor

Dipika Gaonkar

## Acquisition Editor

Sam Wood

## Content Development Editor

Anand Singh

## Technical Editor

Prajakta Mhatre

## Copy Editors

Charlotte Carneiro

Sameen Siddiqui

## Project Coordinator

Akash Poojary

**Proofreaders**

Stephen Copestake

Safis Editing

**Indexer**

Rekha Nair

**Graphics**

Sheetal Aute

**Production Coordinator**

Melwyn D'sa

**Cover Work**

Melwyn D'sa

# About the Author

**Sandeep Kumar Patel** is a senior web developer and the founder of [www.tutorialsavvy.com](http://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*

I would like to thank the three most important people in my life—my parents, Dilip Kumar Patel and Sanjukta Patel, for their love and my little sister, Sangeeta Patel, for her support and motivation.

A special thanks to the team at Packt Publishing, without whom this book wouldn't have been possible.

# About the Reviewers

**Zhelan Chen** graduated from the University of Texas at Dallas with a major in computer science. She has worked as an IT staff member for several companies while developing Java EE, .NET applications, websites, and web services. She has been teaching computer courses part time at Dallas Community College for several years. She holds several Oracle Java EE certificates.

**Krzysztof Cislo** is a software developer and architect with 10 years of professional experience focused on web applications.

He started his programming journey by working with websites created in the PHP language, during which time he also tried to stay up to date with frontend languages, such as JavaScript, HTML, and CSS. After a while, he completely turned his focus to server-side technologies, such as the Java language, which became his main area of interest for a long time.

However, these days he focuses more on frontend-side technologies and is considering going back to the JavaScript world.

**Lars Kappert** (<http://webpro.nl>) is a Dutch frontend solution architect and lead developer. He specializes in architecture, solutions, performance, tooling, and the development of websites and applications. He works closely with a number of core web technologies, including HTML5, JavaScript, NodeJS, and CSS. He is an active open source developer on GitHub (<https://github.com/webpro>), and publishes articles for Medium (<https://medium.com/@webprolific>), Smashing Mag (<https://www.smashingmagazine.com/author/lars-kappert/>), and more. You can follow him on Twitter at @webprolific (<https://twitter.com/webprolific>).

**Sebastian Metzger** graduated from the University of Erlangen-Nuremberg in Germany with a diploma in information systems. He then worked for both big banking corporations and small start-ups, after which he founded his own software development and technology consulting company in 2014.

He started creating single-page web applications in 2011 using Java with the Google Web Toolkit (GWT). Now he has moved on to full stack native JavaScript development that facilitates NodeJS and AngularJS. Always looking into recent

developments, he created his first web components app using Google Polymer in the summer of 2014, which can be found at <http://foodtrack.de>.

Be sure to check out his company web page at <http://sebastianmetzger.com>, where he regularly blogs about the latest technologies and trends.

**www.PacktPub.com**

# **Support files, eBooks, discount offers, and more**

For support files and downloads related to your book, please visit  
[www.PacktPub.com](http://www.PacktPub.com).

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at  
[www.PacktPub.com](http://www.PacktPub.com) and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at <[service@packtpub.com](mailto:service@packtpub.com)> for more details.

At [www.PacktPub.com](http://www.PacktPub.com), you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<https://www2.packtpub.com/books/subscription/packtlib>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

## **Why subscribe?**

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

## **Free access for Packt account holders**

If you have an account with Packt at [www.PacktPub.com](http://www.PacktPub.com), you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

# 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.

# What you need for this book

The following list of tools and libraries are required for this book:

- WebStorm IDE.
- Latest Chrome browser.
- GIT
- npm
- Bower
- Yeoman
- Grunt

# **Who this book is for**

This book is for JavaScript developers wanting to learn and develop web component. This book is also helpful for those who want to learn different frameworks available in the market, by which a web component can be developed. Finally, the book is for everyone interested in a better understanding of web component, to develop their own custom components.

# Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "We can include other contexts through the use of the `include` directive."

A block of code is set as follows:

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

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
<body>
<div id="container"></div>
<template id="aTemplate">
    <h1>Template is activated using importNode method.</h1>
</template>
<script>
```

Any command-line input or output is written as follows:

```
npm install --save b-sortable
```

New terms and important words are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Now, press the **Delete SayHello Element** button which will remove the element from the DOM tree."

**Note**

Warnings or important notes appear in a box like this.

## Tip

Tips and tricks appear like this.

# Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail <[feedback@packtpub.com](mailto:feedback@packtpub.com)>, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at [www.packtpub.com/authors](http://www.packtpub.com/authors).

# Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

## 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.

## Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the book in the search field. The required information will appear under the **Errata** section.

## Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at <[copyright@packtpub.com](mailto:copyright@packtpub.com)> with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

## Questions

If you have a problem with any aspect of this book, you can contact us at <[questions@packtpub.com](mailto:questions@packtpub.com)>, and we will do our best to address the problem.

# Chapter 1. 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:

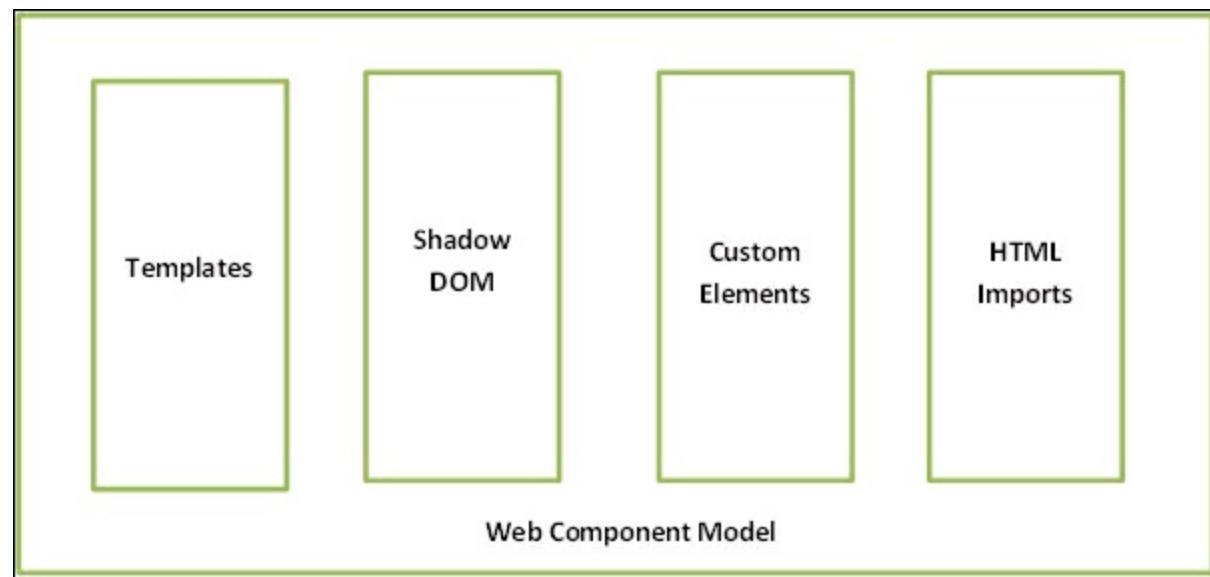
- A web component can be used in multiple applications. It provides interoperability between frameworks, developing the web component ecosystem. This makes it *reusable*.
- A web component has a template that can be used to put the entire markup separately, making it more *Maintainable*.
- As web components 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:

```
interface HTMLTemplateElement : HTMLElement {  
    readonly attribute DocumentFragment content;  
};
```

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.

### Note

To find out more about DocumentFargment, 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:

```
<!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.";
    } else {
        message.innerHTML = "Template element is not 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:



A screenshot of a browser window showing the developer tools' Elements tab. The address bar shows 'localhost:8080/Chapter1/html-tempates/tempateDemo1.html'. The main content area displays the text 'Template element is supported by the browser.' The Elements tab shows the DOM structure:

```
<!DOCTYPE html>
<html>
  <head lang="en">...</head>
  <body>
    <h1 id="message">Template element is supported by the browser.</h1>
    <script>...</script>
  </body>
</html>
```

The 'html' node is selected in the tree view.

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

## Tip

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.

Can I use... Support tables x caniuse.com/#feat=template

IE	Firefox	Chrome	Safari	Opera	iOS Safari*	Opera Mini*	Android Browser*	Chrome for Android
		31						
		33						
		35						
8		36	5.1				4.1	
9	31	37	7		7.1		4.3	
10	32	38	7.1		8		4.4	
11	33	39	8	26	8.1	8	37	39
TP	34	40		27				
	35	41		28				
	36	42						

## 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:

```
<!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" alt="Placeholder for profile image">
</template>
<button id="loadImage" type="button">Load Image</button>
<script>
    document.querySelector('#loadImage').addEventListener('click', function() {
        var messageElement = document.querySelector('#message');
        var templateElement = document.querySelector('#aTemplate');
        var imgElement = templateElement.content.querySelector('img');

        messageElement.innerHTML = '';
        messageElement.appendChild(imgElement);
    });
</script>
```

```
src="http://www.gravatar.com/avatar/c6e6c57a2173fcf2afdd5fe6786e92f.png">
<script>
    alert("This is a script.");
</script>
</template>
<script>
(function() {
    var imageElement = document.getElementById("profileImage"),
        messageElement = document.getElementById("message");
    messageElement.innerHTML = "IMG element "+imageElement;
})();
</script>
</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:

# IMG element null

```
Elements Network Sources Timeline Profiles Resources Audits Console
```

```
<!DOCTYPE html>
▼<html>
▶<head lang="en">...</head>
▼<body>
  <div id="message">IMG element null</div>
  ▼<template id="aTemplate">
    ▼#document-fragment
      
      <script>
        alert("This is a script.");
      </script>
    </template>
  ▶<script>...</script>
  </body>
</html>
```

html body template#aTemplate #document-fragment script

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:

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

The details of the preceding code syntax are listed here:

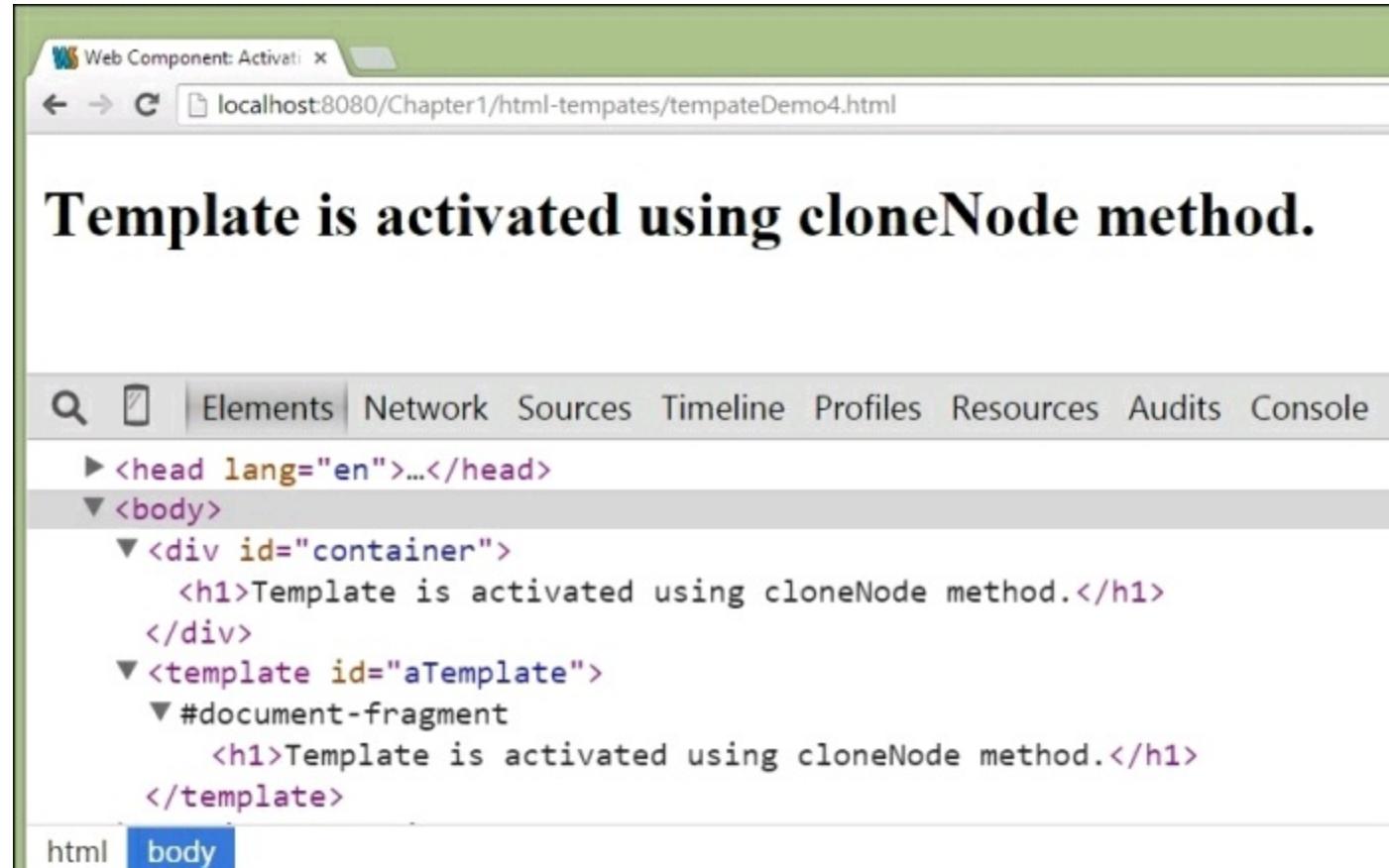
- This method can be applied on a node that needs to be cloned.
- The return type of this method is `Node`.
- The input parameter for this method is of the `Boolean` type and represents a type of cloning. There are 2 different types of cloning, listed as follows:
  - **Deep cloning:** In deep cloning, 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:

```
<!DOCTYPE html>
<html>
<head lang="en">
    <meta charset="UTF-8">
    <title>
        Web Component: Activating template using cloneNode method
    </title>
</head>
<body>
<div id="container"></div>
<template id="aTemplate">
    <h1>Template is activated using cloneNode method.</h1>
</template>
<script>
    var aTemplate = document.querySelector("#aTemplate"),
        container = document.getElementById("container"),
        templateContent = aTemplate.content,
        activeContent = templateContent.cloneNode(true);
    container.appendChild(activeContent);
</script>
</body>
```

```
</html>
```

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:



## Note

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:

```
<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:

```
<!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);
    container.appendChild(activeContent);
</script>
</body>
</html>
```

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:

Web Component: Activat... X

localhost:8080/Chapter1/html-templates/templateDemo2.html

# Template is activated using importNode method.

Elements Network Sources Timeline Profiles Resources Audits Console

```
<!DOCTYPE html>
▼ <html>
  ▶ <head lang="en">...</head>
  ▼ <body>
    ▼ <div id="container">
      <h1>Template is activated using importNode method.</h1>
    </div>
    ▼ <template id="aTemplate">
      ▼ #document-fragment
        <h1>Template is activated using importNode method.</h1>
      </template>
    ▶ <script>...</script>
```

html body

## Note

To find out more about the `importNode` method, visit:

<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 HTML `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:

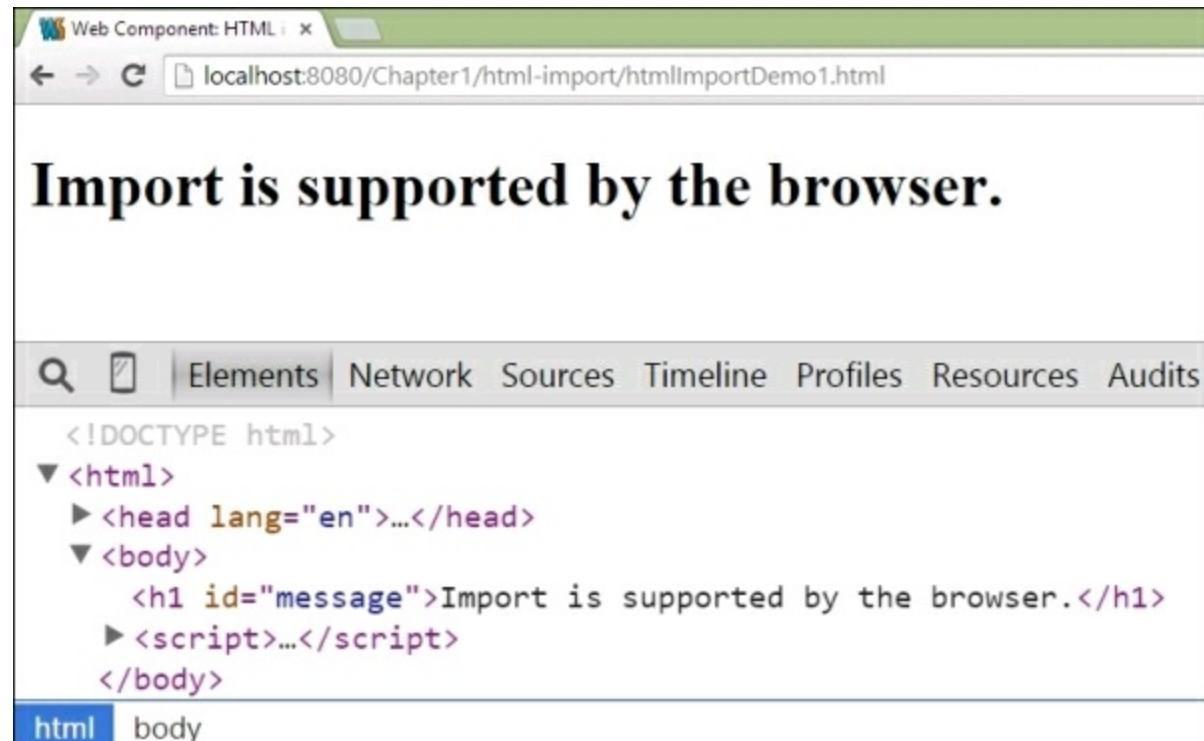
```
<!DOCTYPE html>  
<html>  
<head lang="en">
```

```

<meta charset="UTF-8">
<title>
    Web Component: HTML import support
</title>
</head>
<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>

```

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:



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

## Tip

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:



## 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:

```
<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:

```
<!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:

The screenshot shows a browser window with the URL `localhost:8080/Chapter1/html-import/htmlImportDemo3.html`. The main content area displays the text **This is from another HTML file document.**. Below the content, the browser's developer tools are open, specifically the Elements tab. The DOM tree is displayed, showing the following structure:

```
<html>
  <head>
    <link rel="import" href="message.html">
    <script>...</script>
  </head>
  <body>
    <h1>This is from another HTML file document.</h1>
  </body>
</html>
```

The `message.html` file imported via the `<link>` element contains its own `<html>`, `<head>`, and `<body>` structures, which are correctly nested within the imported file's root node. The browser's developer tools also show the `head` tab selected at the bottom.

## 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:

```
<!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:

localhost:8080/Chapter1/

localhost:8080/Chapter1/html-import/htmlImportDemo2.html

# This is from another HTML file document.

Elements Network Sources Timeline Profiles Resources Audits Console

```
▼<html>
  ▼<head lang="en">
    ▶<script async>...</script>
    ▼<link rel="import" href="message.html" onload="handleSuccess(event)" onerror="handleError(event)">
      ▶#document
    </link>
  </head>
  ▼<body>
    <h1>
      This is from another HTML file document.
    </h1>
  </body>
```

html body h1

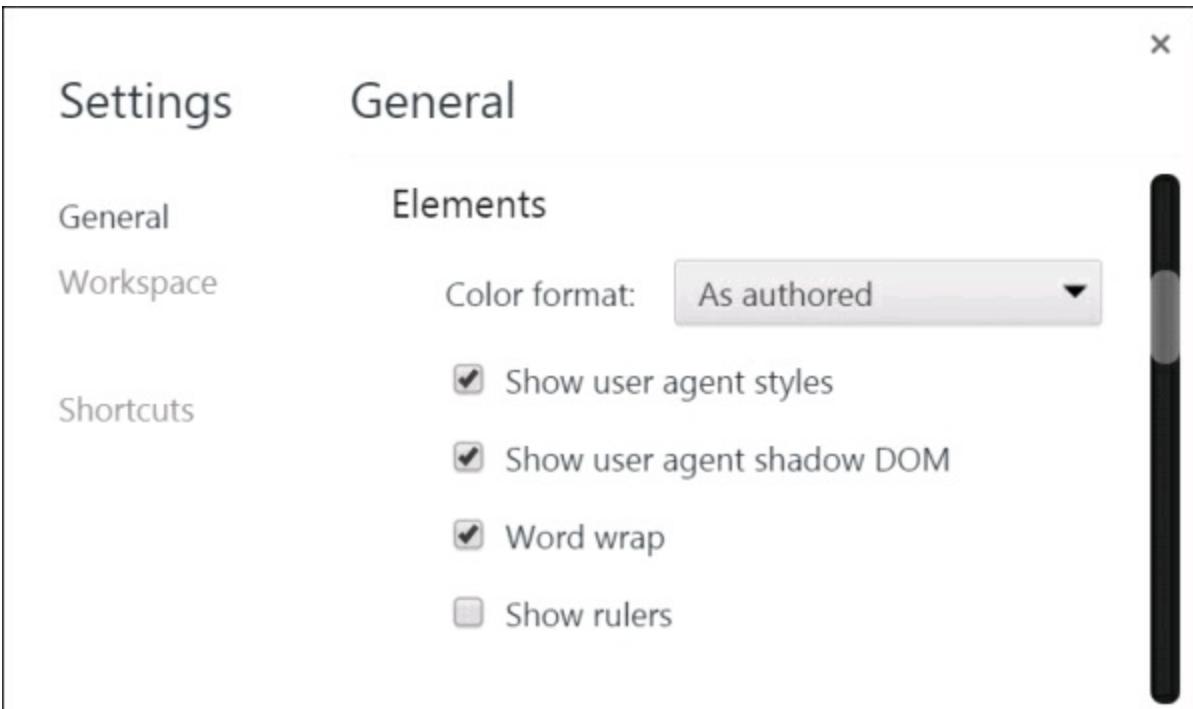
# Shadow DOM

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:

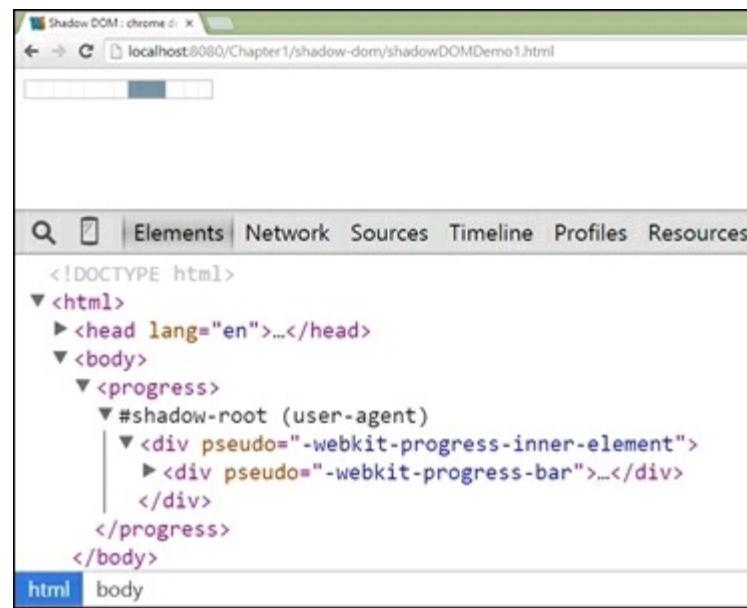
- **Style override:** The document stylesheet may change the style of the web component.
- **Script alteration:** The document JavaScript may alter some part of the web component.
- **ID overlap:** There may be a duplicate ID present in the document, which can lead to many erroneous situations.

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:



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:



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:

```
<!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:

# Shadow DOM is supported by the browser.

The screenshot shows the Chrome DevTools Elements tab with the following DOM structure:

```
<!DOCTYPE html>
▼ <html>
  ▶ <head lang="en">...</head>
  ▼ <body>
    <h1 id="message">Shadow DOM is supported by the browser.</h1>
    ▶ <script>...</script>
  </body>
```

The `html` tab is selected at the bottom left.

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:

The screenshot shows a compatibility matrix for various web browsers. The columns represent different browsers: IE, Firefox, Chrome, Safari, Opera, iOS Safari\*, Opera Mini\*, Android Browser\*, and Chrome for Android. The rows represent specific features or versions. The background color of each cell indicates the browser's support status: green for full support, red for partial support, and grey for no support.

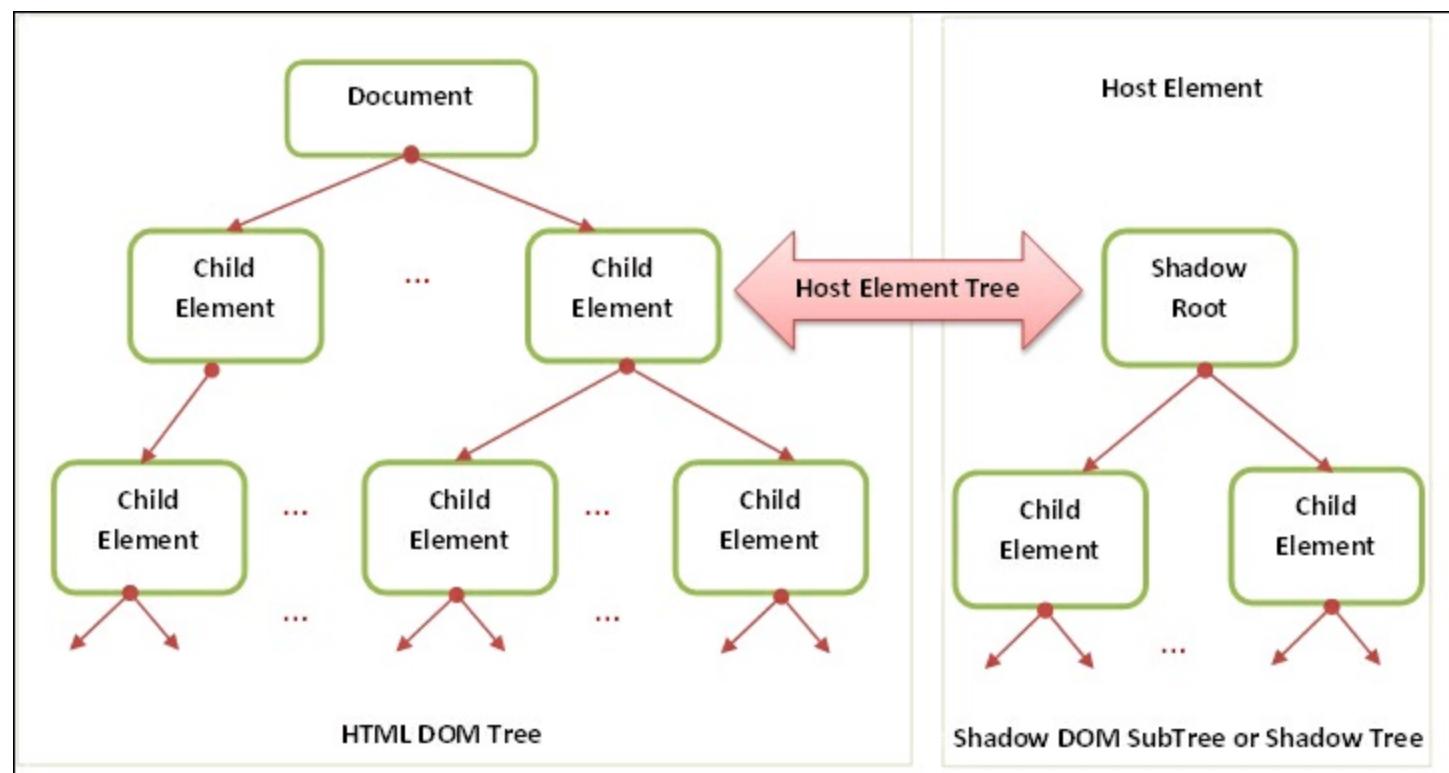
	IE	Firefox	Chrome	Safari	Opera	iOS Safari*	Opera Mini*	Android Browser*	Chrome for Android
1			31						
2			33						
3			35						
4	8		36	5.1				4.1	
5	9	31	37	7		7.1		4.3	
6	10	32	38	7.1		8		4.4	
7	11	33	39	8	26	8.1	8	37	39
8	TP	34	40			27			
9		35	41			28			
10		36	42						

## Shadow tree

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:



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 getElementById(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

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:

```
<!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>
    </template>
</body>
```

```

        }
    </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.getElementsByClassName("labelClass"));
        console.log("getElementsByTagName:
", shadowRoot2.getElementsByTagName("label"));
            console.log("getElementsByTagNameNS:
", shadowRoot2.getElementsByTagNameNS("*","label"));
                console.log("getSelection() Method:
", shadowRoot2.getSelection());
                    console.log("elementFromPoint:
", shadowRoot2.elementFromPoint(8,9));
//Shadow Root Properties
    console.log("activeElement: ", shadowRoot2.activeElement);
    console.log("host: ", shadowRoot2.host);
    console.log("olderShadowRoot: ", shadowRoot2.olderShadowRoot);
console.log("styleSheets: ", shadowRoot2.styleSheets);
    console.log("innerHTML: ", shadowRoot2.innerHTML);
})();
</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:

The screenshot shows a browser window with the title "Shadow Root: Method &". The address bar displays "localhost:8080/Chapter1/shadow-dom/shadowDOMDemo4.html". The main content area shows the text "Name" followed by a yellow-highlighted input field containing "Sandeep". Below this, the developer tools Elements tab is open, showing the DOM structure:

```
<!DOCTYPE html>
▼<html>
  ▶<head lang="en">...</head>
  ▼<body>
    ▼<div id="aShadowHost">
      ▼#shadow-root
        ▶<style>...</style>
        ▼<form>
          <label for="nameElement" class="labelClass">Name</label>
          ▶<input type="text" id="nameElement" placeholder="Enter your name" value="Sandeep" autofocus>
        </form>
      #shadow-root
    </div>
    ▶<template id="selectorTemplate">...</template>

```

The "script" tab is selected in the bottom navigation bar.

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

Elements Network Sources Timeline Profiles Resources Audits »   

**<top frame>** ▼  Preserve log

```

getElementById:                                         shadowDOMDemo4.html:30
▶ <input type="text" id="nameElement" placeholder="Enter your name" value="Sandeep" autofocus>

getElementsByClassName:                                shadowDOMDemo4.html:31
▶ [Label.LabelClass, item: function, namedItem: function]

getElementsByTagName:                                 shadowDOMDemo4.html:32
▶ [Label.LabelClass, item: function, namedItem: function]

getElementsByTagNameNS:                               shadowDOMDemo4.html:33
▶ [Label.LabelClass, item: function, namedItem: function]

getSelection() Method:                               shadowDOMDemo4.html:34
▶ Selection {type: "Range", extentOffset: 3, extentNode: form,
  baseOffset: 3, baseNode: form...}

elementFromPoint:                                    shadowDOMDemo4.html:35
<label for="nameElement" class="labelClass">Name</label>

```

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

Developer Tools - http://localhost:8080/Chapter1/shadow-dom/shadowDOMDemo4.html

Elements Network Sources Timeline Profiles Resources Audits »   

**<top frame>** ▼  Preserve log

```

activeElement:                                         shadowDOMDemo4.html:38
▶ <input type="text" id="nameElement" placeholder="Enter your name" value="Sandeep" autofocus>

host:   ▶ <div id="aShadowHost">...</div>                                shadowDOMDemo4.html:39
olderShadowRoot: #shadow-root                         shadowDOMDemo4.html:40
styleSheets:                                           shadowDOMDemo4.html:41
▶ StyleSheetList {0: CSSStyleSheet, Length: 1, item: function}

innerHTML:
  <style>
    :host input{ background: lightyellow;}
    :host .labelClass{ color:blue;}
  </style>
  <form>
    <label for="nameElement" class="labelClass">Name</label>

```

# 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:

```
<!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>
```

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:

The screenshot shows a browser window with the title "Web Component: custom". The address bar displays "localhost:8080/Chapter1/custom-elements/customElementDemo1.html". The main content area contains the text "Custom element is supported by the browser." Below this, the browser's developer tools are open, specifically the "Elements" tab. The DOM tree is visible, starting with the root <!DOCTYPE html>. It includes a <html> element, a <head> element with a lang attribute set to "en", and a <body> element. Inside the body, there is an <h1> element with an id attribute set to "message" containing the text "Custom element is supported by the browser.". There is also a <script> element. The bottom of the developer tools interface shows a navigation bar with tabs for "html", "body", and "h1#message", where "html" is currently selected.

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 tables x caniuse.com/#feat=custom-elements

IE	Firefox	Chrome	Safari	Opera	iOS Safari *	Opera Mini *	Android Browser *	Chrome for Android
		31						
		33						
		35						
8		36	5.1				4.1	
9	31	37	7		7.1		4.3	
10	32	38	7.1		8		4.4	
11	33	39	8	26	8.1	8	37	39
TP	34	40		27				
	35	41		28				
	36	42						

## 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.

## Note

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

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Object/create](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/create)

## 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:

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

## Note

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

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Object/defineProperty](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty)

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`:

```
var newObject = Object.create(HTMLElement.prototype);  
Object.defineProperties(newObject, {  
    title:{  
        writable: true  
    },  
    country:{  
        writable: false,  
        value: "India"  
    }  
});
```

## Note

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

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Object/defineProperties](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperties)

## 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 `attaributeChanged` 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:

```
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:

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

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:

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

## Note

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

<https://developer.mozilla.org/en-US/docs/Web/API/document.registerElement>

## Extending an element

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

```
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:

```
<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:

```
<!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>
```

## Tip

### 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.

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:

# Welcome to custom element 1

## Welcome to custom element 2

Elements Network Sources Timeline Profiles Resources Audits Console

```
<!DOCTYPE html>
▼ <html>
  ▶ <head lang="en">...</head>
  ▼ <body>
    <my-name title="Welcome to custom element 1">Welcome to custom element 1</my-name>
    <br>
    <my-name title="Welcome to custom element 2">Welcome to custom element 2</my-name>
  </body>
```

html head

# 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:

```
<!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>
            Fruits <content select="b.fruit"></content>.
        </h1>
        <h1>
```

```

        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>
        <b class="fruit">Banana </b>
        <b class="flower">Lotus </b>
        <b class="fruit">Grapes </b>
        <b class="flower">Jasmine </b>
    </selector-component>
</body>
</html>

```

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:

Web Component: content

localhost:8080/Chapter1/insertion-point/insertinPointDemo2.html

# Fruits Apple Orange Banana Grapes .

# Flowers Rose Lotus Jasmine .

Elements Network Sources Timeline Profiles Resources Audits Console

```
> <head lang="en">...</head>
<body>
  <selector-component>
    <#shadow-root>
      <style>...</style>
      <h1>...</h1>
      <h1>...</h1>
      <b class="fruit">Apple </b>
      <b class="flower">Rose </b>
      <b class="fruit">Orange </b>
      <b class="fruit">Banana </b>
      <b class="flower">Lotus </b>
      <b class="fruit">Grapes </b>
      <b class="flower">Jasmine </b>
  </selector-component>
```

html body selector-component #shadow-root h1

## 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:

```
<!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`.

## Shadow Root 2

Shadow Root 1 Button

Elements Network Sources Timeline Profiles Resources Audits

```
▼ <body>
  ▼ <div id="aShadowHost">
    ▼ #shadow-root
      ▼ <fieldset style="width: 62%;">
        <legend>Shadow Root 2</legend>
        <shadow></shadow>
      </fieldset>
    ▼ #shadow-root
      | <button>Shadow Root 1 Button</button>
    </div>
  ▶ <template id="shadow1Template">...</template>
  ▶ <template id="shadow2Template">...</template>
  ▶ <script>...</script>
</body>
```

html body div#aShadowHost #shadow-root fieldset

# 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:

```
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:

```
: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:

```
: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:

```
:host ::content b{  
    color: blue;
```

}

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.

```
<!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);
                shadow.appendChild(templateNodes);
            }
        })
    </script>
</head>
<body>
    <header-element>
        <h1>Hello World</h1>
    </header-element>
</body>
</html>
```

```

};

window.setTimeout(function() {
    document.registerElement("header-element", {
        prototype: objectPrototype
    });
}, 3000);
})();
</script>
</head>
<body>
    <header-element>
        <b>Web Component</b>
    </header-element>
</body>
</html>

```

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.

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.

The screenshot shows a browser window with the URL `localhost:8080/Chapter1/styling-components/header-demo.html`. The page title is "Registering Element...". Below the title, the browser's developer tools Elements tab is active, displaying the DOM tree. The tree structure is as follows:

```
<!DOCTYPE html>
▼ <html>
  ▶ <head lang="en">...</head>
  ▼ <body>
    ▼ <header-element>
      <b>Web Component</b>
      ::after
    </header-element>
  </body>

```

The "header-element" node is expanded, showing its children: a **Web Component** element and a ::after pseudo-element. The "body" node is also expanded. The "html" node is the root of the tree. The "body" node is currently selected. The status bar at the bottom of the developer tools shows "html" and "body".

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:

Web Component: Unresolved

localhost:8080/Chapter1/styling-components/header-demo.html

# HELLO WEB COMPONENT

Elements Network Sources Timeline Profiles Resources

```
<!DOCTYPE html>
▼ <html>
  ▶ <head lang="en">...</head>
  ▼ <body>
    ▼ <header-element>
      ▼ #shadow-root
        ▶ <style>...</style>
        ▼ <h1>
          "Hello "
          <content></content>
        </h1>
        <b>Web Component</b>
      </header-element>
    </body>
  </html>
```

# 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:

```
<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>
```

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:

```
<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);
        };
        var digitalClockElement = document.registerElement("digital-
clock", {
```

```
        prototype: objectPrototype
    );
})();
</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.

# 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:

```
<!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:



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:

```
<!DOCTYPE html>
<html>
  <head lang="en">
    <meta charset="UTF-8">
    <title>Web component : digital clock element</title>
    <link rel="import" href="clock-element.html">
    <#document
      <html>
        <head>...</head>
        <body></body>
      </html>
    </link>
  </head>
  <body>
    <digital-clock>
      <#shadow-root
        <style>...</style>
        <div class="clock">...</div>
      </digital-clock>
    </body>
  </html>
```

html body digital-clock #shadow-root

Console Search Emulation Rendering

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>.

## 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.
- **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:

```
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.

## Note

The X-Tag core library source code can be downloaded by visiting:

<https://github.com/x-tag/core>

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

```
xtag.register('<element-name>', {
    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
        }
    },
    accessors: {
<property name> : {
        attribute: {
            //type and value of the property
        }
    }
},
methods: {
<method name> : function() {
            //Code for the method
        }
},
events: {
    '<event type>:delegate(<element>)': function(e) {
        //Code for event handler
    }
}
});
```

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:

```
<!DOCTYPE html>
<html>
<head lang="en">
<meta charset="UTF-8">
<title>Web Component: xTag custom element support</title>
<script src="x-tag-components.js"></script>
<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:

Click Me

Make Red

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:

Click Me

Make Red

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:

Click Me

Make Red

Elements Network Sources Timeline

<top frame> ▾ Preserve log

click event is fired.

## Note

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.

## Note

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.

## Note

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.

## Note

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.

## Note

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.

# Chapter 2. Introducing Polymer

In this chapter, you will be introduced to the Polymer library. You will learn how to install the Polymer library to an application, which will be followed by a section on core element and paper elements. In the later section of this chapter, you will explore the Polymer designer tool.

## What is Polymer?

Polymer is a library developed by Google developers to support web component development. Basically, it is built on the guidelines by W3C web component specification. Before it is implemented by all browsers, it also adds some syntactic sugar on top of the web component standards. The goal of Polymer is an attempt at getting a test of web component specification completely implemented by a browser.

Polymer follows the element-based application development where we can build our own elements similar to an HTML element. Element-based application development increases reusability of the developed code across different applications.

# Installing and configuring Polymer

The Polymer library can be obtained in the following three different ways:

- Downloading ZIP file
- Using GIT clone
- Using Bower

## Downloading ZIP file

The Polymer library can be downloaded as a zipped file using the link at <https://www.polymer-project.org/0.5/docs/start/getting-the-code.html#using-zip>. The directory can be unzipped and the Polymer library can be extracted from it. The following screenshot shows the unzipped directory containing the Polymer library:

Name	Date modified	Type
core-component-page	12/19/2014 9:12 PM	File folder
NodeBind	12/19/2014 9:12 PM	File folder
observe-js	12/19/2014 9:12 PM	File folder
polymer	12/19/2014 9:12 PM	File folder
polymer-expressions	12/19/2014 9:12 PM	File folder
polymer-gestures	12/19/2014 9:12 PM	File folder
TemplateBinding	12/19/2014 9:12 PM	File folder
URL	12/19/2014 9:12 PM	File folder
webcomponentsjs	12/19/2014 9:12 PM	File folder

The Polymer library is present inside the `polymer` directory. The `polymer.html` is present inside the `polymer` directory. This `polymer.html` file can be included to application by using *HTML Import*. The following code shows the syntax of the Polymer file:

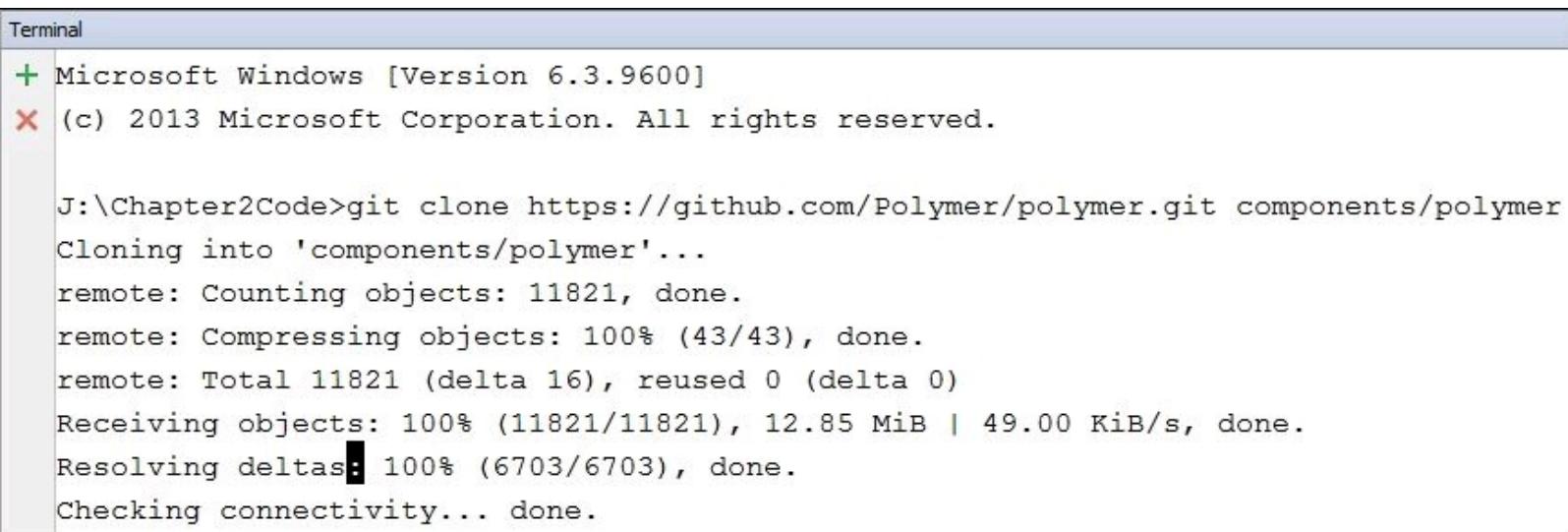
```
<link rel="import" href="bower_components/polymer/polymer.html">
```

# Using GIT clone

Polymer can also be cloned using the GIT tool. The command for cloning GIT in the project is as follows:

```
git clone https://github.com/Polymer/polymer.git components/polymer
```

The following screenshot shows the GIT cloning of the Polymer library inside the Window's command prompt:



```
Terminal
+ Microsoft Windows [Version 6.3.9600]
X (c) 2013 Microsoft Corporation. All rights reserved.

J:\Chapter2Code>git clone https://github.com/Polymer/polymer.git components/polymer
Cloning into 'components/polymer'...
remote: Counting objects: 11821, done.
remote: Compressing objects: 100% (43/43), done.
remote: Total 11821 (delta 16), reused 0 (delta 0)
Receiving objects: 100% (11821/11821), 12.85 MiB | 49.00 KiB/s, done.
Resolving deltas: 100% (6703/6703), done.
Checking connectivity... done.
```

Once the above GIT clone is completed, it creates a directory structure inside the current directory, as shown in the following screenshot. We can find the `polymer` directory that contains `polymer.html` and some other files inside the `components` folder.



```
Terminal
+ Microsoft Windows [Version 6.3.9600]
X (c) 2013 Microsoft Corporation. All rights reserved.

J:\Chapter2Code>git clone https://github.com/Polymer/polymer.git components/polymer
Cloning into 'components/polymer'...
remote: Counting objects: 11821, done.
remote: Compressing objects: 100% (43/43), done.
remote: Total 11821 (delta 16), reused 0 (delta 0)
Receiving objects: 100% (11821/11821), 12.85 MiB | 49.00 KiB/s, done.
Resolving deltas: 100% (6703/6703), done.
Checking connectivity... done.
```

# Using Bower

Bower is a package manager for web application development. A package manager is a collection of software tools that automate the process of installing, upgrading, configuring, and removing software packages for a computer's operating system in a consistent manner. To install Bower in the system, **node package manager (npm)** is required. More details about Bower installation are available at <http://bower.io/#install-bower>.

Assuming that Bower is installed in the system, we will go ahead and install the Polymer library. For the first time, Bower can be initialized to a web application project using the following command:

**Bower init**

When the preceding command is executed, it asks a set of questions regarding the web application configuration properties. The following screenshot shows the command prompt with the above command in effect:

```
Terminal
+ J:\Chapter2Code>bower init
? name: Chapter2Code
? version: 0.0.1
? description: PolymerJs library demonstration
? main file:
? what types of modules does this package expose?:
? keywords:
? authors: saan1984 <sandeep_giet@yahoo.com>
? license: MIT
? homepage:
? set currently installed components as dependencies?: Yes
```

After the execution of the preceding command, it creates a `bower.json` file inside the application directory. The following code shows the content of the `bower.json` file containing all the default settings of the application:

```
{
  "name": "Chapter2Code",
  "version": "0.0.1",
  "authors": [
    "saan1984 <sandeep_giet@yahoo.com>"
  ],
  "description": "Polymer.js library Demonstration",
  "license": "MIT",
  "ignore": [
    "**/.*", "node_modules", "bower_components", "test", "tests"
  ]
}
```

Once the `bower.json` file is created, now we can install the Polymer library. Use the following command to install the Polymer library:

**bower install -save Polymer/polymer**

The following screenshot shows the command prompt installing the Polymer library inside an application:

```
Terminal
+ J:\Chapter2Code>bower install Polymer/polymer
X bower polymer#*          not-cached git://github.com/Polymer/polymer.git#*
  bower polymer#*          resolve git://github.com/Polymer/polymer.git#*
  bower polymer#*          download https://github.com/Polymer/polymer/archive/0.5.2.tar.gz
  bower polymer#*          extract archive.tar.gz
  bower polymer#*          invalid-meta polymer is missing "main" entry in bower.json
  bower polymer#*          invalid-meta polymer is missing "ignore" entry in bower.json
  bower polymer#*          resolved git://github.com/Polymer/polymer.git#0.5.2
  bower core-component-page#^0.5.0  not-cached git://github.com/Polymer/core-component-page.git#^0.5.0
  bower core-component-page#^0.5.0  resolve git://github.com/Polymer/core-component-page.git#^0.5.0
  bower webcomponentsjs#^0.5.0   not-cached git://github.com/Polymer/webcomponentsjs.git#^0.5.0
  bower webcomponentsjs#^0.5.0   resolve git://github.com/Polymer/webcomponentsjs.git#^0.5.0
  bower webcomponentsjs#^0.5.0   download https://github.com/Polymer/webcomponentsjs/archive/0.5.2
```

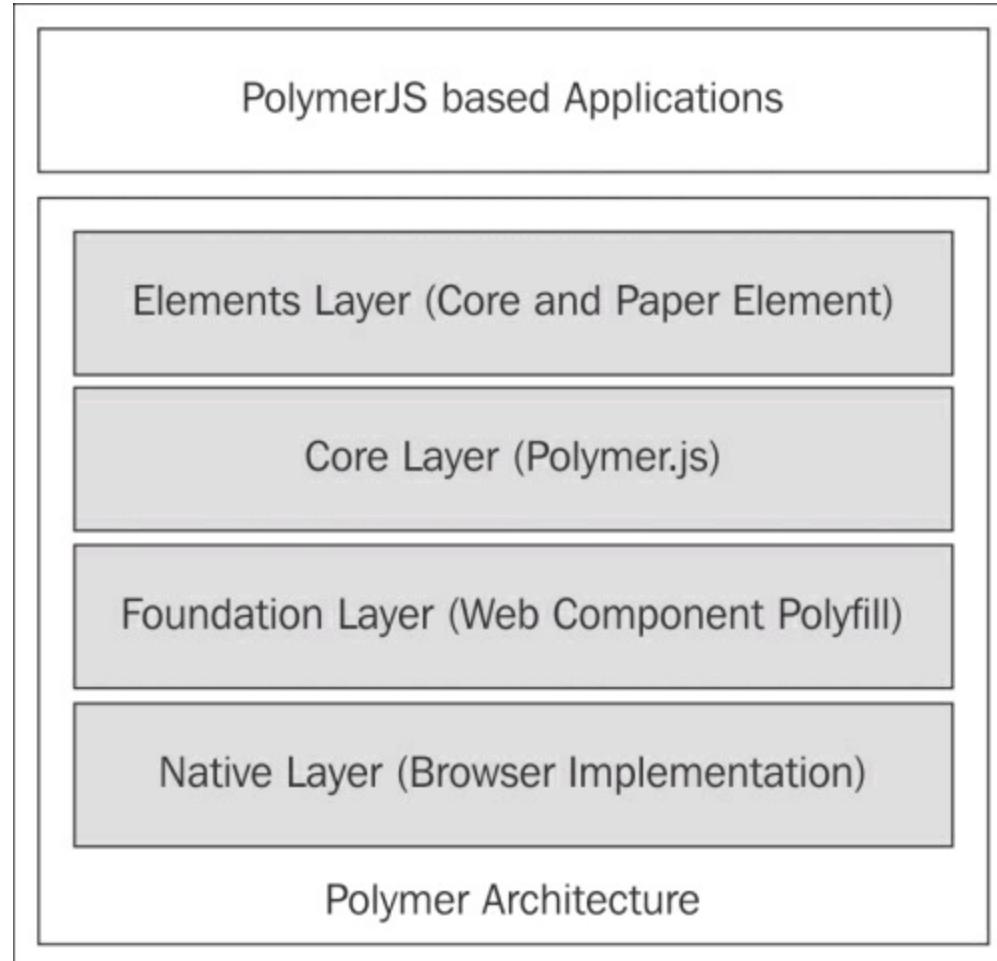
Once the Polymer library is installed, the `bower.json` file gets updated with the `dependencies` property. The following code shows the updated content of the `bower.json` file:

```
{
  "name": "Chapter2Code",
  "version": "0.0.1",
  "authors": ["saan1984 <sandeep_giet@yahoo.com>"],
  "description": "Polymer.js library Demonstration",
  "license": "MIT",
```

```
"ignore": [  
  "**/.*", "node_modules", "bower_components", "test", "tests"],  
"dependencies": {  
  "webcomponentsjs": "Polymer/webcomponentsjs#^0.5.0",  
  "core-component-page": "Polymer/core-component-page#^0.5.0"  
}  
}
```

# Architecture of PolymerJS

The Polymer library is built upon multiple layers of technology, with each layer serving a specific purpose. The following diagram shows the technology stack of the Polymer library:



The following list shows four important layers of Polymer:

- **Native layer:** This layer represents the current state of browser support and implementation for the web component specification.
- **Foundation layer:** This layer contains the polyfill libraries for the web component specification. A polyfill is a piece of code that simulates behavior when it is not available natively in a browser. To find out more about polyfill, refer to <http://en.wikipedia.org/wiki/Polyfill>.
- **Core layer:** This layer contains the Polymer library code. The Polymer library can be found in the `polymer.html` file.
- **Elements layer:** This layer contains the core and paper elements.

At the top of the Polymer architecture, another layer of applications are present. This layer represents the applications developed using the Polymer library. So, we can conclude that the Polymer technology is built on the following three different building blocks:

- Web components with polyfill
- The Polymer library
- Elements

## Web components with polyfill

Web component specification is new to browsers. It is not completely implemented by browsers, so to bridge this gap, Polymer comes with a polyfill in a `webcomponent.js` file.

In the early release of the Polymer library, polyfill was named as `platform.js`. The `webcomponent.js` script provides the polyfill for the following technologies:

- **Shadow DOM**: This provides a private scope to separate the content from presentation. You can refer to [Chapter 1, Introducing Web Components](#) to revise the Shadow DOM concept.
- **HTML Imports**: This includes the external HTML document in the current page. You can refer to [Chapter 1, Introducing Web Components](#) to revise the HTML Import concept.
- **Custom elements**: A new element can be created using custom element. You can refer to [Chapter 1, Introducing Web Components](#) to revise the custom element concept.
- **WeakMap**: The WeakMap object is a collection of key/value pairs in which the keys are used for getting the associated value. To find out more about WeakMap objects, refer to [https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global\\_Objects/WeakMap](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/WeakMap)
- **Mutation observer**: This provides a way to execute a callback method by listening to the DOM changes. To find out more about mutation observer, refer to <https://developer.mozilla.org/en/docs/Web/API/MutationObserver>.

During the installation of Polymer using Bower, the web component polyfill gets downloaded automatically to the project directory. To install the web component polyfill as an independent download, use the following Bower command:

```
bower install webcomponentsjs
```

We can also install the web component polyfill using npm. To install the web component polyfill using npm, use the following command:

```
npm install webcomponents.js
```

Once the above command is executed successfully, a `webcomponentsjs` directory is added to the project component directory. This directory contains the following polyfill files:

- `ShadowDOM.js`: This JavaScript file contains the polyfill code for Shadow DOM
- `ShadowDOM.min.js`: This file is a minified version of `ShadowDOM.js`.
- `HTMLImports.js`: This JavaScript file contains the polyfill code for HTML Imports
- `HTMLImports.min.js`: This file is a minified version of `HTMLImports.js`
- `CustomElements.js`: This JavaScript file contains the polyfill code for custom elements
- `CustomElements.min.js`: This file is a minified version of `CustomElements.js`
- `webcomponents.js`: This JavaScript file contains the polyfill code for the entire web component specification
- `webcomponents.min.js`: This file is a minified version of `webcomponents.js`
- `webcomponents-lite.js`: This is a lighter version of `webcomponent.js` containing polyfill code for HTML imports and custom elements
- `webcomponents-lite.min.js`: This file is a minified version of `webcomponents-lite.js`

To find out more about the web component polyfill, refer to  
<http://webcomponents.org/polyfills>.

## The Polymer library

The Polymer library is the core file of this framework and is built upon the web component polyfill. When installing the Polymer library using the Bower tool, a `polymer` directory gets installed to the project folder. This `polymer` directory mainly contains the following files:

- `polymer.js`: This is the core JavaScript file, which has the code for all the 'magical features' provided by the Polymer library. We will explore these magical features in the coming section.

- `polymer-min.js`: This is a minified version of the `polymer.js` file.
- `layout.html`: This file contains CSS attributes for creating layout positions for the elements present inside the page.
- `polymer.html`: This file contains combined code by including links to `polymer.js` and `layout.html` files, and can be included to a current page using HTML Import.

## Elements

In the Polymer world, everything is an element. The Polymer team come with a set of elements to build web application. These elements can be divided into the following two categories:

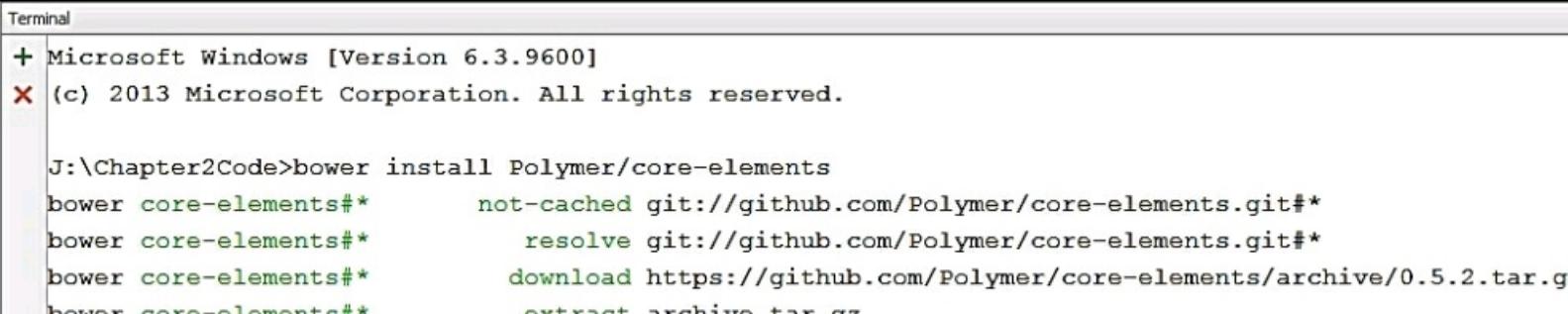
- Core elements
- Paper elements

### Core elements

Polymer core elements are a set of general purpose UI and non-UI elements for building web pages. They include icons, layouts, toolbars, AJAX, signaling, and storage elements. There are many core elements present inside the Polymer library. We can directly download the entire set of core elements as a single ZIP from <https://bowerarchiver.appspot.com/archive?core-elements=Polymer/core-elements>. Also, we can include the entire core element set using the following Bower command:

```
bower install Polymer/core-elements
```

The following screenshot shows the installation of core elements in the command prompt using Bower:



The screenshot shows a Windows Command Prompt window titled "Terminal". The command `bower install Polymer/core-elements` is being run. The output shows the progress of the installation, including cloning from GitHub, resolving dependencies, downloading the archive, and extracting it. The terminal also displays system information like the Windows version and copyright notice.

```

Terminal
+ Microsoft Windows [Version 6.3.9600]
x (c) 2013 Microsoft Corporation. All rights reserved.

J:\Chapter2Code>bower install Polymer/core-elements
bower core-elements#*           not-cached git://github.com/Polymer/core-elements.git#*
bower core-elements#*           resolve git://github.com/Polymer/core-elements.git#*
bower core-elements#*           download https://github.com/Polymer/core-elements/archive/0.5.2.tar.gz
bower core-elements#*           extract archive.tar.gz

```

Once the preceding command is successfully executed, we can include the set of core elements to our web page by HTML Imports. The following code shows the HTML Import of core elements inside the web page:

```
<link rel="import" href="components/core-elements/core-elements.html">
```

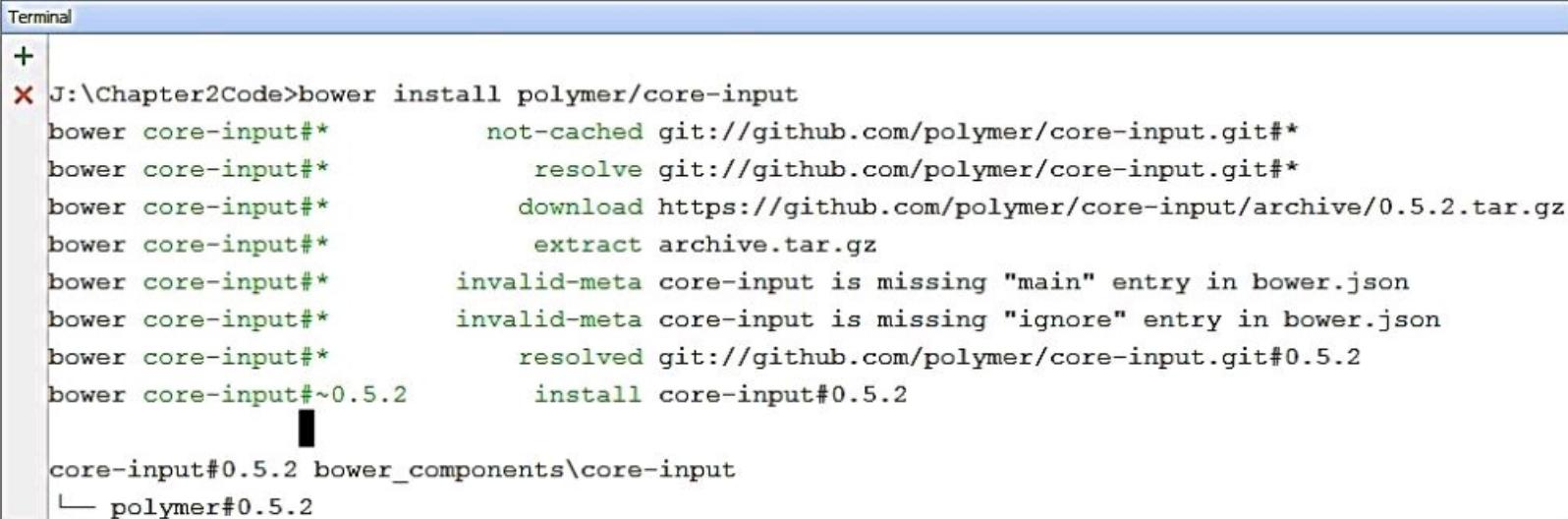
We can find the complete list of core elements at <https://www.polymer-project.org/docs/elements/core-elements.html>. However, we will explore few of them.

## The core-input element

The `core-input` element is a single line input field, which extends the native HTML input element. To install the `core-input` field as a standalone component, we can use the following Bower command:

```
bower install Polymer/core-input
```

The following screenshot shows the command prompt with the `core-input` field installation in progress:



A screenshot of a terminal window titled "Terminal". The command `bower install polymer/core-input` is being run. The output shows the progress of the installation, including cloning from GitHub, resolving dependencies, and extracting the archive. It also highlights errors where the component's bower.json files are missing "main" and "ignore" entries. Finally, it shows the component being installed under the path `core-input#0.5.2`.

```
+ J:\Chapter2Code>bower install polymer/core-input
bower core-input#*           not-cached git://github.com/polymer/core-input.git#*
bower core-input#*             resolve git://github.com/polymer/core-input.git#*
bower core-input#*             download https://github.com/polymer/core-input/archive/0.5.2.tar.gz
bower core-input#*               extract archive.tar.gz
bower core-input#*             invalid-meta core-input is missing "main" entry in bower.json
bower core-input#*             invalid-meta core-input is missing "ignore" entry in bower.json
bower core-input#*               resolved git://github.com/polymer/core-input.git#0.5.2
bower core-input#*               install core-input#0.5.2
core-input#0.5.2 bower_components\core-input
└── polymer#0.5.2
```

After the installation of the `core-input` element, it can be imported using HTML Import. The code for including `core-input` to the current page is as follows:

```
<link rel="import" href="bower_components/core-input/core-input.html">
```

A normal input element can be converted to a core input by using the `is` attribute. The code for the `core-input` element is as follows:

```
<input is="core-input">
```

The `core-input` element has the following properties and methods:

- `value`: This attribute is inherited from the `<input>` element and represents the current value
- `committedValue`: This attribute contains the final value when the user hits the *Enter* key or blurs the input after changing the value
- `commit`: This method is used to transfer the existing content inside the `value` attribute to the `committedValue` attribute
- `preventInvalidInput`: This is a Boolean property and takes true value to prevent the invalid input

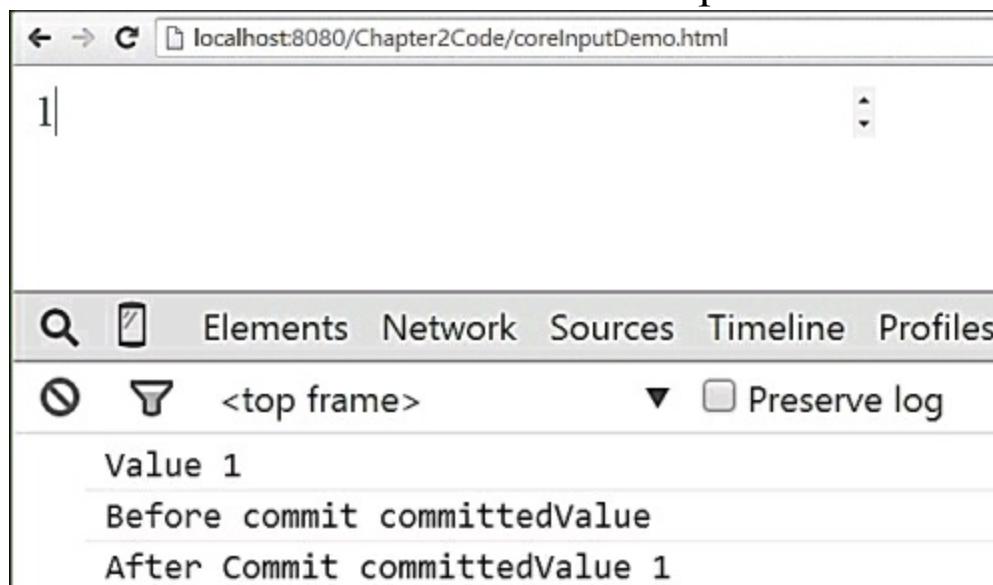
Now, let's check out an example of using the `core-input` value and the preceding properties/methods. The following code shows the use of `core-input` element:

```
<!DOCTYPE html>
<html>
<head lang="en">
<meta charset="UTF-8">
<title>Core Input Demo</title>
<script src="bower_components/webcomponentsjs/webcomponents.min.js">
</script>
<link rel="import" href="bower_components/polymer/polymer.html">
<link rel="import" href="bower_components/core-input/core-input.html">
</head>
<body>
  <section>
    <input is="core-input" id="coreInput1"
           type="number" placeholder="Enter a Number"
           preventInvalidInput onchange="changeHandler()">
  </section>
<script>
  var changeHandler = function() {
    var coreInput1 = document.getElementById("coreInput1");
    console.log("Value " + coreInput1.value);
    console.log("Before commit committedValue
"+coreInput1.committedValue);
    //Committing the value
    coreInput1.commit();
    console.log("After Commit committedValue
"+coreInput1.committedValue);
  }
</script>
```

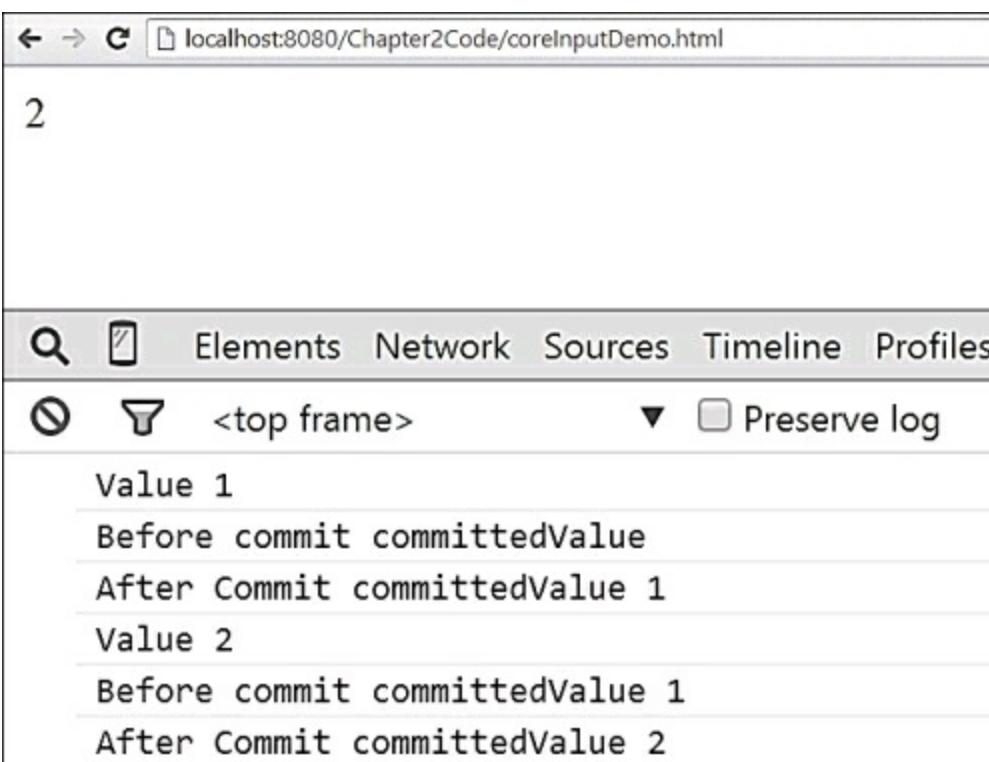
```
</body>  
</html>
```

The details of the preceding code are as follows:

- A `core-input` element with the `coreInput1` ID is included in the code. The `coreInput1` element is of type `number`.
- This `coreInput1` element has the `preventInvalidInput` attribute set to `true`, which helps in preventing any invalid input from the user.
- A `changeHandler` method is attached to the `change` event of the `coreInput1` element and gets called when any change is detected in the `core` element.
- Let's enter a valid number `1` into the `core` element and then click on the body of the page. This user action will trigger the `changeHandler` method to be executed. The following screenshot shows the output of the preceding code, where the `1` is entered to the `core` input element:



- Let's enter another number `2`, which again triggers the `changeHandler` method to be executed. The following screenshot shows the output of the preceding code, where `2` is entered to the `core` input element:



To find out more about the Polymer core-input element, refer to <https://www.polymer-project.org/docs/elements/core-elements.html#core-input>.

## The core-label element

The `core-label` element provides a feature of the `<label>` element to target a specific element. To install `core-label` as a standalone component, we can use the following Bower command:

```
bower install Polymer/core-label
```

The following screenshot shows the command prompt with the `core-label` element installation in progress:

## Terminal

```
+ J:\Chapter2Code>bower install polymer/core-label
X bower not-cached      git://github.com/polymer/core-label.git#
 *
 bower resolve          git://github.com/polymer/core-label.git#
 *
 bower download         https://github.com/polymer/core-label/ar-
 chive/0.5.2.tar.gz
 bower extract          core-label#* archive.tar.gz
 bower invalid-meta    core-label is missing "main" entry in bower.json
 bower resolved         git://github.com/polymer/core-label.git#0.5.2
 bower install          core-label#0.5.2

core-label#0.5.2 bower_components\core-label
└── polymer#0.5.2
```

After the installation of the `core-label` element, it can be imported using the HTML Import. The code for including the core input to the current page is as follows:

```
<link rel="import" href="bower_components/core-label/core-label.html">
```

The code for the `core-label` element is as follows:

```
<core-label></core-label>
```

The `core-label` element has the `for` property. The `for` attribute works like a query selector for targeting elements.

Now, let's check out an example of using the `core-label` element. The following code shows the different usages of the `core-label` element:

```
<!DOCTYPE html>
<html>
<head lang="en">
  <meta charset="UTF-8">
  <title>Core Label Demo</title>
  <script src="bower_components/webcomponentsjs/webcomponents.min.js">
```

```

</script>
<link rel="import" href="bower_components/polymer/polymer.html">
<link rel="import" href="bower_components/core-input/core-
input.html">
<link rel="import" href="bower_components/core-label/core-
label.html">
</head>
<body>
<section>
  <core-label for="#coreInput1">Student Name :</core-label>
  <input is="core-input" id="coreInput1"
         type="text" placeholder="Enter Your Name">
  <br/>
  <core-label>
    Student Subject :
    <input is="core-input" type="text"
          placeholder="Enter Your Subject">
  </core-label>
</section>
</body>
</html>

```

In the preceding code, the `core-label` element has been used to target core input elements. This targeting has been done in two different ways:

- **Using for attribute:** The **Student Name** text is used as a label to target the `coreInput1` element
- **Wrapping target element:** The **Student Subject** text is used as a label to the `core-input` element by wrapping it up inside a `core-label` element

The following screenshot shows the output of the preceding code, giving two different usages of the `core-label` element:

Student Name : Enter Your Name  
Student Subject : Enter Your Subject

The screenshot shows the Chrome DevTools Elements tab with the DOM tree for the file 'coreLabelDemo.html'. The tree is as follows:

```
<!DOCTYPE html>
<html lang="en">
  <head>...</head>
  <body>
    <section>
      <core-label for="#coreInput1" id="core-label-0">Student Name :</core-label>
      <input is="core-input" id="coreInput1" type="text" placeholder="Enter Your Name" aria-label="Enter Your Name" aria-labelledby="core-label-0">
      <br>
      <core-label id="core-label-1">...</core-label>
    </section>
  </body>
```

The 'html' tab is selected in the bottom navigation bar.

## The core-tooltip element

The `core-tooltip` element provides the feature of showing supportive information in a tooltip by targeting a specific element. To install `core-tooltip` as a standalone component, we can use the following Bower command:

```
bower install Polymer/core-tooltip
```

The following screenshot shows the command prompt with the `core-tooltip` element installation in progress:

```
Terminal
+ J:\Chapter2Code>bower install polymer/core-tooltip
X bower not-cached  git://github.com/polymer/core-tooltip.git#*
bower resolve      git://github.com/polymer/core-tooltip.git#*
bower download     https://github.com/polymer/core-tooltip/archive/0.5.2.tar.gz

bower extract      core-tooltip#* archive.tar.gz
bower invalid-meta core-tooltip is missing "main" entry in bower.json
bower invalid-meta core-tooltip is missing "ignore" entry in bower.json
bower resolved     git://github.com/polymer/core-tooltip.git#0.5.2
bower cached       git://github.com/Polymer/core-focusable.git#0.5.2
bower validate     0.5.2 against git://github.com/Polymer/core-focusable.git#^0
.5.0
bower cached       git://github.com/Polymer/core-icon-button.git#0.5.2
bower validate     0.5.2 against git://github.com/Polymer/core-icon-button.git#
^0.5.0
core-tooltip#0.5.2 bower_components\core-tooltip
|__ core-focusable#0.5.2
|__ core-icon-button#0.5.2
|__ core-resizable#0.5.2
|__ paper-fab#0.5.2
└__ polymer#0.5.2

core-icon-button#0.5.2 bower_components\core-icon-button
└__ core-icons#0.5.2

core-resizable#0.5.2 bower_components\core-resizable
```

After the installation of the `core-tooltip` element, it can be imported using the HTML Import. The code for including the `core-tooltip` to the current page is as follows:

```
<link rel="import" href="bower_components/core-tooltip/core-
tooltip.html">
```

The code for the `core-tooltip` element is as follows:

```
<core-tooltip></core-tooltip>
```

The `core-tooltip` element has the following properties:

- `Label`: This attribute takes a string to display as a tooltip for the current target element
- `Position`: This attribute takes the `right`, `left`, `top`, and `bottom` as values to align the tooltip around the target element
- `Show`: This attribute takes the Boolean value and for the `true` value makes the tooltip appear by default
- `tip/tipAttribute`: The `tip` attribute specifies the HTML content for a rich tooltip and customizes this attribute with the `tipAttribute` attribute

- `noArrow`: This attribute takes the Boolean value, and for `true` input, the arrow of the tooltip will not be shown

Now, let's check out an example of using the `core-tooltip` element. The following code shows the different usages of the `core-tooltip` element:

```
<!DOCTYPE html>
<html>
<head lang="en">
  <meta charset="UTF-8">
  <title>Core ToolTip Demo</title>
  <script src="bower_components/webcomponentsjs/webcomponents.min.js">
  </script>
  <link rel="import" href="bower_components/polymer/polymer.html">
    <link rel="import" href="bower_components/core-tooltip/core-
tooltip.html">
</head>
<body>
  <section>
    <core-tooltip position="right" show>
      <h2>Polymer.js Icon Tooltip</h2>
      <div tip>
        
      </div>
    </core-tooltip>
    <br/>
    <core-tooltip label="Polymer.js core-tooltip element"
position="right" noarrow>
      
    </core-tooltip>
    <br/>
  </section>
</body>
</html>
```

The following screenshot shows the output of the preceding code, demonstrating the use of the `core-tooltip` element and its different properties:

# PolymerJS Icon Tooltip



PolymerJS core-tooltip element

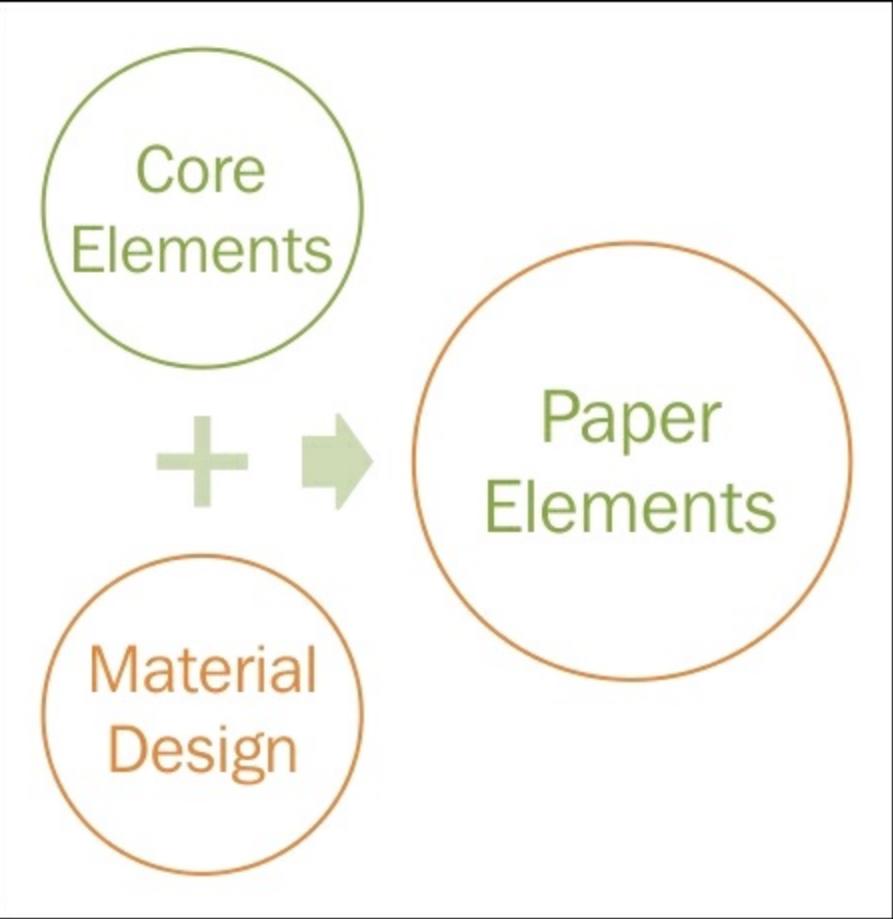
Elements Network Sources Timeline Profiles Resources Audits Console

```
<!DOCTYPE html>
▼<html>
  ►<head lang="en">...</head>
  ▼<body>
    ▼<section>
      ►<core-tooltip position="right" show role="tooltip" tabindex="0">...
      ...</core-tooltip>
      <br>
      ►<core-tooltip label="PolymerJS core-tooltip element" position="right" noarrow role="tooltip" tabindex="0" focused>...</core-tooltip>
      <br>
    </section>
  </body>
```

html body

## Paper elements

Paper elements are built on top of the Polymer core element. These elements are more rich in UI compared to the core elements. Basically, paper elements are built on the guidelines of *Google Material Design*. The following diagram shows a graphical representation of the relationship between these elements:



## Tip

To find out more about *Google Material Design*, refer to <http://www.google.com/design/spec/material-design/introduction.html>.

There are many paper elements present inside the Polymer library. We can directly download the entire set of paper element as a single ZIP file from <https://bowerarchiver.appspot.com/archive?paper-elements=Polymer/paper-elements>. Also, we can include the entire paper element set using the following Bower command:

```
bower install Polymer/paper-elements
```

The following screenshot shows the installation of paper elements in the command prompt using Bower:

## Terminal

```
+ J:\Chapter2Code>bower install Polymer/paper-elements
X bower cached      git://github.com/Polymer/paper-elements.git#
  0.5.2
  bower validate      0.5.2 against git://github.com/Polymer/paper-
  -elements.git#*
  bower cached      git://github.com/Polymer/paper-fab.git#0.5.2

  bower validate      0.5.2 against git://github.com/Polymer/paper-
  -fab.git#^0.5.0
  bower cached      git://github.com/Polymer/paper-shadow.git#0.
  5.2
  bower validate      0.5.2 against git://github.com/Polymer/paper-
  -shadow.git#^0.5.0
```

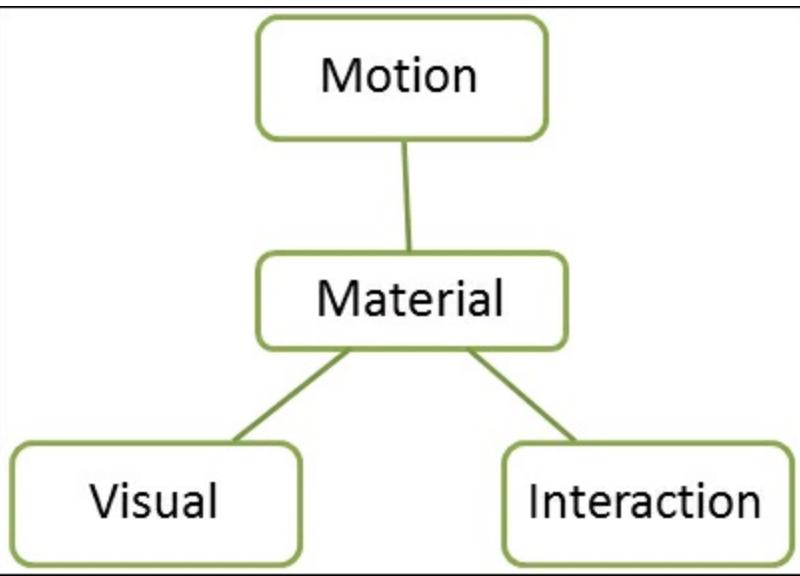
Once the preceding command is successfully executed, we can include the paper elements set to our web page by HTML Imports. The following code shows the HTML Import of the paper element inside the web page:

```
<link rel="import" href="components/paper-elements/paper- elements.html">
```

We can find a complete list of the paper elements at <https://www.polymer-project.org/docs/elements/paper-elements.html>. However, we will explore a few of them here.

## Material design

This is a common design guideline for developing user interfaces across different devices. These guidelines are influenced with real-world material, such as paper and ink. The following diagram shows three building blocks—**Visual**, **Motion**, and **Interaction**:



To find out more about *Material Design* specification, refer to <http://www.google.com/design/spec/material-design/introduction.html>.

## The paper-checkbox element

The paper-checkbox element provides the feature of a normal checkbox element, but with better visualization through animation. To install paper-checkbox as a standalone component, we can use the following Bower command:

```
bower install Polymer/paper-checkbox
```

The following screenshot shows the command prompt with the paper-checkbox element installation in progress:

```
Terminal
+
X J:\Chapter2Code>bower install Polymer/paper-checkbox
bower cached      git://github.com/Polymer/paper-checkbox.git#0.5.2
bower validate    0.5.2 against git://github.com/Polymer/paper-checkbox.git#*
bower install     paper-checkbox#0.5.2

paper-checkbox#0.5.2 bower_components\paper-checkbox
├── core-label#0.5.2
├── font-roboto#0.5.2
└── paper-radio-button#0.5.2

J:\Chapter2Code>
```

After the installation of the `paper-checkbox` element, it can be imported using the HTML Import. The code for including `paper-checkbox` to the current page is as follows:

```
<link rel="import" href="bower_components/paper-checkbox/paper-checkbox.html">
```

The code for the `paper-checkbox` element is as follows:

```
<paper-checkbox></paper-checkbox>
```

The `paper-checkbox` element has the following events that can be attached to a callback method:

- `change`: This event is fired when the checked state changes due to user interaction
- `core-change`: This event is fired when the checked state changes

Now, let's check out an example of using the `paper-checkbox` element. The following code shows the different usages of the `paper-checkbox` element:

```
<!DOCTYPE html>
<html>
<head lang="en">
  <meta charset="UTF-8">
  <title>Paper CheckBox Demo</title>
  <meta name="viewport" content="width=device-width, minimum-scale=1.0, initial-scale=1, user-scalable=yes">
  <script src="bower_components/webcomponentsjs/webcomponents.min.js">
  </script>
  <link rel="import" href="bower_components/polymer/polymer.html">
  <link rel="import" href="bower_components/paper-checkbox/paper-checkbox.html">
</head>
<body unresolved>
  <section>
    <core-label horizontal layout center>
      <paper-checkbox for id="mathCheckbox"></paper-checkbox>
      <h1>Mathematics</h1>
    </core-label>
    <core-label horizontal layout center>
      <paper-checkbox for checked id="compCheckbox"></paper-
```

```
checkbox>
  <h1>Computer</h1>
  </core-label>
</section>
<script>
  (function() {
    var mathCheckbox = document.getElementById("mathCheckbox"),
        compCheckbox = document.getElementById("compCheckbox");
    //checked state changes due to user interaction
    mathCheckbox.addEventListener("change", function (e) {
      console.log("change event fired");
    }),
    //when the checked state changes by any mean
    compCheckbox.addEventListener("core-change", function (e) {
      console.log("core-change event fired");
    });
  })();
</script>
</body>
</html>
```

The following screenshot shows the output of the preceding code, demonstrating the use of paper-checkbox:

# □Mathematics

## ✓Computer

Elements | Network Sources Timeline Profiles »

```
<!DOCTYPE html>
▼<html>
►<head lang="en">...</head>
▼<body>
▼<section>
▼<core-label horizontal layout center>
►<paper-checkbox for id="mathCheckbox" role="checkbox" tabindex="0" aria-checked="false">
...</paper-checkbox>
<h1>Mathematics</h1>
</core-label>
►<core-label horizontal layout center>...</core-label>
</section>
```

html body

## The paper-slider element

The `paper-slider` element provides the feature of a range element, but with better visualization with animation. To install `paper-slider` as a standalone component, we can use the following Bower command:

```
bower install Polymer/paper-slider
```

The following screenshot shows the command prompt with the `paper-slider` element installation in progress:

```
+  
x J:\Chapter2Code>bower install Polymer/paper-slider  
bower cached      git://github.com/Polymer/paper-slider.git#0.5.2  
bower validate    0.5.2 against git://github.com/Polymer/paper-slider.git#*  
bower install     paper-slider#0.5.2  
  
paper-slider#0.5.2 bower_components\paper-slider  
|   core-ally-keys#0.5.2  
|   font-roboto#0.5.2  
|   paper-input#0.5.2  
|   paper-progress#0.5.2  
  
J:\Chapter2Code>
```

After the installation of the `paper-slider` element, it can be included using the HTML Import. The code for including the paper slider to the current page is as follows:

```
<link rel="import" href="bower_components/paper-slider/paper-slider.html">
```

The code for the `paper-slider` element is as follows:

```
<paper-slider></paper-slider>
```

The `paper-slider` element has the following properties that can be used to customize the paper element:

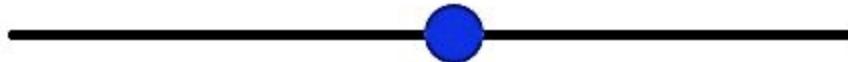
- **Min, Max, and Value:** These are the properties inherited from the core range element and represent the lowest, highest, and current value of the slider respectively. To find out about core range element properties, refer to <https://www.polymer-project.org/docs/elements/core-elements.html#core-range>.
- **Snaps:** This is based on the step value, where the slider arranges tick marks evenly with the slider's thumb snap.
- **Pin:** This property shows a visual pin, with a numeric value label shown when the slider thumb is pressed.

There are a few more properties present inside the `paper-slider` element that can be found at <https://www.polymer-project.org/docs/elements/paper-elements.html#paper-slider>.

Now, let's check out an example of using the `paper-slider` element. The following code shows the different usages of the `paper-slider` element:

```
<!DOCTYPE html>
<html>
<head lang="en">
  <meta charset="UTF-8">
  <title>Paper Slider Demo</title>
  <script src="bower_components/webcomponentsjs/webcomponents.min.js">
  </script>
  <link rel="import" href="bower_components/polymer/polymer.html">
  <link rel="import" href="bower_components/paper-slider/paper-
slider.html">
</head>
<body>
  <section>
    <paper-slider min="10" max="200"
                  value="110" pin="true"
                  snaps="true">
    </paper-slider>
  </section>
</body>
</html>
```

The following screenshot shows the output of the preceding code, demonstrating the use of `paper-slider` properties:



Elements Network Sources Timeline Profiles Resources

```
▼ <html>
  ► <head lang="en">...</head>
  ▼ <body>
    ▼ <section>
      ▼ <paper-slider min="10" max="200" value="110" pin=
        "true" snaps="true" role="slider" tabindex="0" aria-
        valuemin="10" aria-valuemax="200" aria-valuenow="110">
        ► #shadow-root
        </paper-slider>
      </section>
    </body>
```

html body

## The paper-button element

The `paper-button` element provides the feature of a range element, but with better visualization through animation. To install `paper-button` as a standalone component, we can use the following Bower command:

```
bower install Polymer/paper-button
```

The following screenshot shows the command prompt with the `paper-button` element installation in progress:

```
Terminal + J:\Chapter2Code>bower install Polymer/paper-button
X bower cached      git://github.com/Polymer/paper-button.git#0.5.2
bower validate      0.5.2 against git://github.com/Polymer/paper-button.git#^0.5
.0
bower cached      git://github.com/Polymer/paper-button.git#0.5.2
bower validate      0.5.2 against git://github.com/Polymer/paper-button.git#*
bower install      paper-button#0.5.2

paper-button#0.5.2 bower_components\paper-button
└── core-focusable#0.5.2
└── core-icon#0.5.2
└── paper-ripple#0.5.2
└── paper-shadow#0.5.2
└── polymer#0.5.2
```

After the installation of the `paper-button` element, it can be included using the HTML Import. The code for including `paper-button` to the current page is as follows:

```
<link rel="import" href="bower_components/paper-button/paper-button.html">
```

The code for the `paper-button` element is as follows:

```
<paper-button></paper-button>
```

The `paper-button` element has the following properties that can be used to customize the element:

- `raised`: This attribute creates a shadow effect for the paper button
- `recenteringTouch`: The ripple effect produced by the button press can be customized using this attribute
- `fill`: This constrains the ripple effect produced by the button press to a circle within the button.

Now, let's check out an example of using the `paper-button` element. The following code shows the different usages of the `paper-button` element:

```
<!DOCTYPE html>
<html>
<head lang="en">
  <meta charset="UTF-8">
```

```

<title>Paper Button Demo</title>
<script src="bower_components/webcomponentsjs/webcomponents.min.js">
</script>
<link rel="import" href="bower_components/polymer/polymer.html">
<link rel="import" href="bower_components/paper-button/paper-
button.html">
</head>
<body>
  <section>
    <paper-button>Button</paper-button>
    <paper-button raised>Button1</paper-button>
    <paper-button nolink>Button2</paper-button>
    <paper-button fill="false">Button3</paper-button>
    <paper-button recenteringTouch="true">Button4</paper- button>
  </section>
</body>
</html>

```

The following screenshot shows the output of the preceding code, demonstrating the use of paper-button properties:

The screenshot shows a web browser window with the URL `localhost:8080/Chapter2Code/paperButtonDemo.html`. The page displays five buttons labeled `BUTTON`, `BUTTON1`, `BUTTON2`, `BUTTON3`, and `BUTTON4`. The `BUTTON1` button is highlighted with a white background and a thin gray border, indicating it is the active or selected button.

The browser's developer tools are open, specifically the Elements tab of the DevTools interface. The code pane shows the following HTML structure:

```

<!DOCTYPE html>
<html>
  <head lang="en">...</head>
  <body>
    <section>
      <paper-button role="button" tabindex="0">...</paper-button>
      <paper-button raised role="button" tabindex="0">...</paper-button>
      <paper-button noink role="button" tabindex="0">...</paper-button>
      <paper-button fill="false" role="button" tabindex="0">...</paper-button>
      <paper-button recenteringtouch="true" role="button" tabindex="0">...</paper-button>
    </section>
  </body>

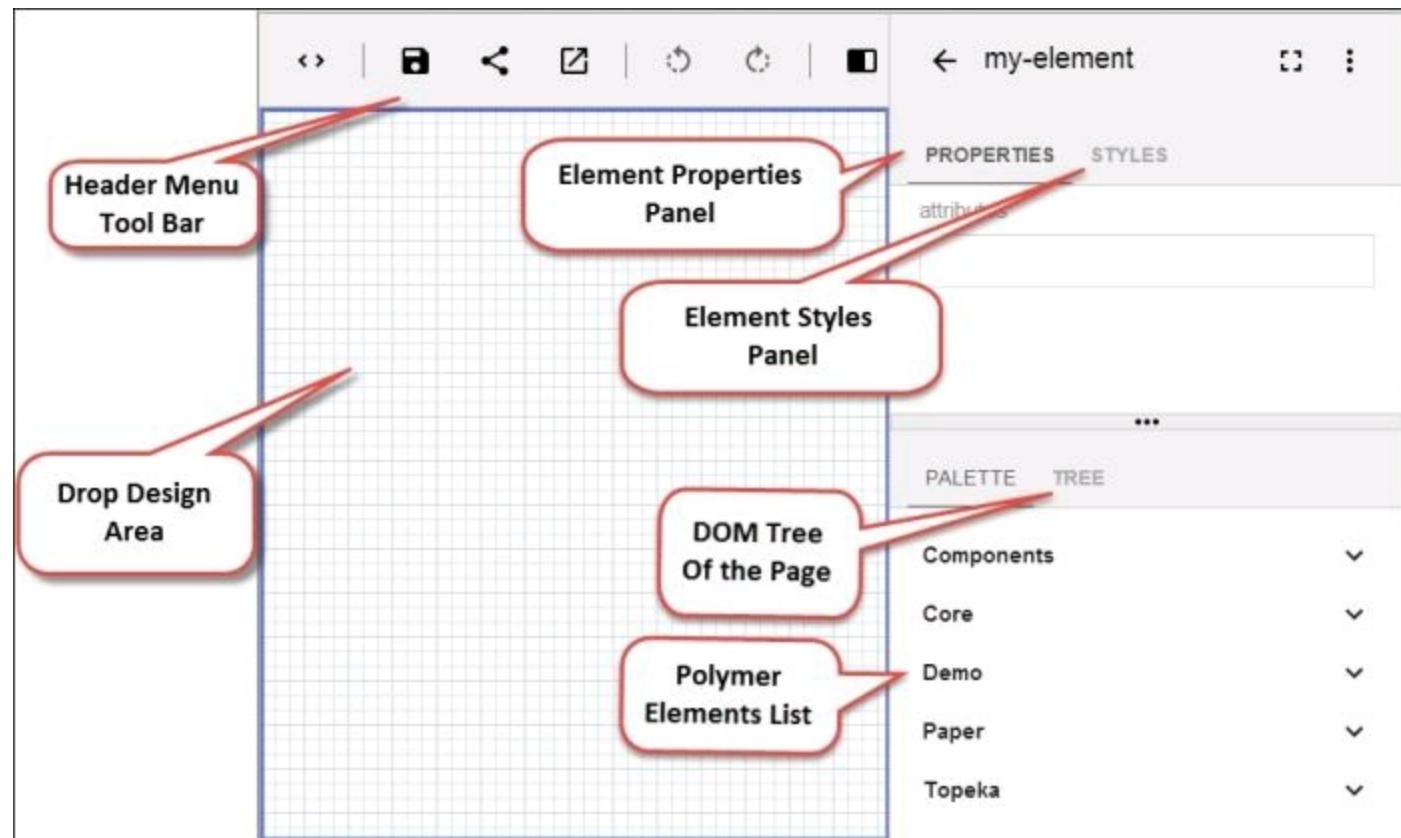
```

The `Elements` tab is currently selected in the DevTools toolbar. The code pane shows the entire HTML document structure, including the DOCTYPE, head, body, section, and paper-button elements. The browser's address bar at the top also shows the URL `localhost:8080/Chapter2Code/paperButtonDemo.html`.

# Polymer designer tool

The Polymer designer tool is an online tool for creating Polymer elements visually. It provides a drag and drop environment for a developer to create web application pages. You can find this tool at <https://www.polymer-project.org/tools/designer/>. The source code for the designer tool can be found at <https://github.com/polymer/designer>.

The following screenshot shows the home page of the designer tool:



The designer tool has the following three different main sections:

- **Header menu toolbar:** This toolbar is present at the top of the designer tool. It contains the following seven different options for developers:
  - **Toggle code/design**—Using this option, we can toggle the view between design and code.
  - **Save**—Using this option, we can save the progress of the development.
  - **Share Gist**—Using this option, we can share the code with GitHub gist.
  - **Launch preview**—Using this option, we can preview the developed page

in a new window.

- **Undo**—Using this option, we can restore the last changed state.
  - **Redo**—Using this option, we can restore the next changed state.
  - **Toggle full width**—Using this option, we can make the design area to full width window.
- **Main design area:** This is the drop target of the element. In this area, we can drop any element that needs to be present in the page. The design area is filled with horizontal and vertical grid lines.
  - **Right menu bar:** This bar is used to show the details of the currently selected element in the design. It is again divided into two subsections:
    - **Properties and styles:** This subsection shows all the properties and style attributes of the currently selected element in the design area. We can change the value of any attribute or style for the selected element.
    - **Palette and trees:** The palette view contains the entire set of draggable elements that can be dropped into the designer area. The tree section shows the current document tree of the page in parent–child relationship.

## Developing with the designer tool

In this section, we will develop an e-mail subscription form using the Polymer designer tool. We have to follow these steps to work with the designer tool:

- Getting a GitHub token
- Developing the e-mail subscription form
- Previewing the design

### Getting a GitHub token

We need to get a GitHub token to save the work of the designer tool as a gist. When you click on the **Save** option, it asks for a GitHub token. You can get a new GitHub token from <https://github.com/settings/tokens/new>. The following screenshot shows the form for creating a GitHub token:

## Token description

What's this token for?

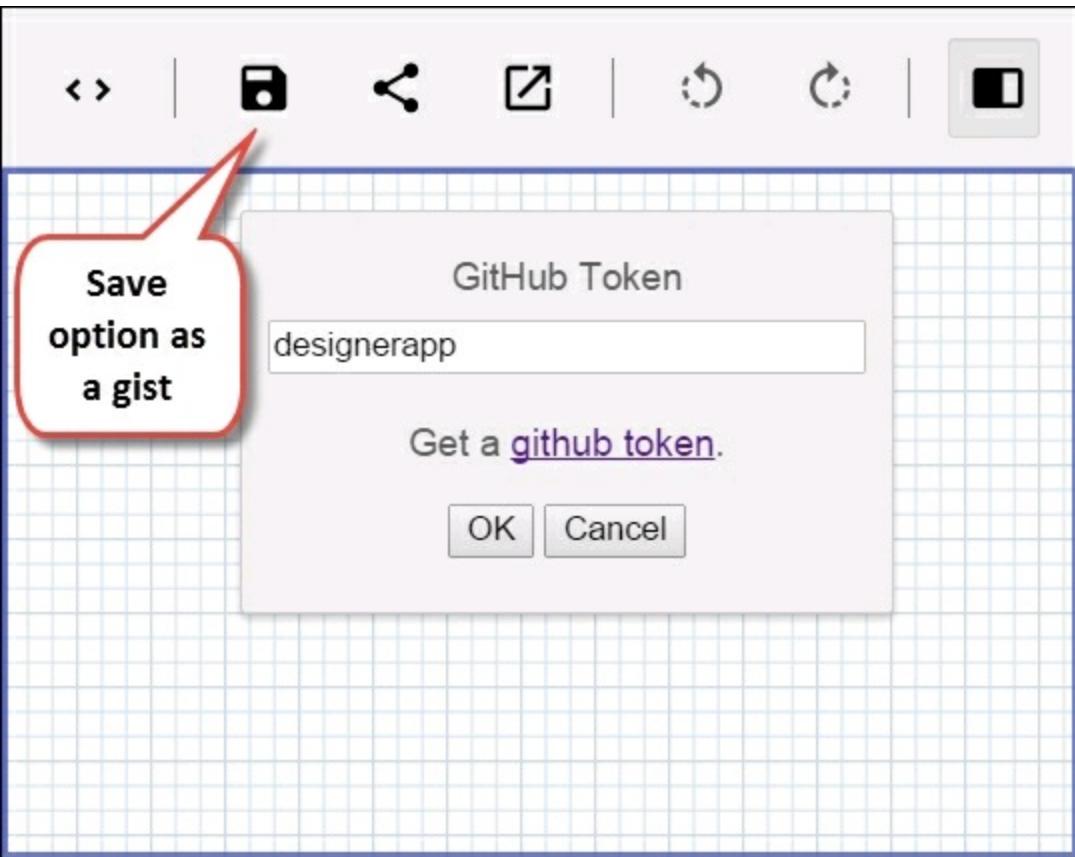
## Select scopes

Scopes *limit* access for personal tokens. Read more about OAuth scopes.

- |  |   |  |
|--|---|--|
| <input checked="" type="checkbox"/> <b>repo</b> ⓘ        | <input type="checkbox"/> <b>repo:status</b> ⓘ     | <input type="checkbox"/> <b>repo_deployment</b> ⓘ  |
| <input checked="" type="checkbox"/> <b>public_repo</b> ⓘ | <input type="checkbox"/> <b>delete_repo</b> ⓘ     | <input checked="" type="checkbox"/> <b>user</b> ⓘ  |
| <input type="checkbox"/> <b>user:email</b> ⓘ             | <input type="checkbox"/> <b>user:follow</b> ⓘ     | <input type="checkbox"/> <b>admin:org</b> ⓘ        |
| <input type="checkbox"/> <b>write:org</b> ⓘ              | <input type="checkbox"/> <b>read:org</b> ⓘ        | <input type="checkbox"/> <b>admin:public_key</b> ⓘ |
| <input type="checkbox"/> <b>write:public_key</b> ⓘ       | <input type="checkbox"/> <b>read:public_key</b> ⓘ | <input type="checkbox"/> <b>admin:repo_hook</b> ⓘ  |
| <input type="checkbox"/> <b>write:repo_hook</b> ⓘ        | <input type="checkbox"/> <b>read:repo_hook</b> ⓘ  | <input type="checkbox"/> <b>admin:org_hook</b> ⓘ   |
| <input checked="" type="checkbox"/> <b>gist</b> ⓘ        | <input type="checkbox"/> <b>notifications</b> ⓘ   |  |

ⓘ Personal access tokens function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to authenticate to the API over Basic Authentication.

The GitHub token created for this example is designerapp. Now we can use this token in the designer tool. The following screenshot shows the popup window asking for a GitHub token when the **Save** option is clicked:



In the popup window, we can input the created GitHub token and we are ready for the next step—developing an e-mail subscription form.

## Developing an e-mail subscription form

To develop an e-mail subscription form, we have dragged the following items from the **PALETTE** section to the designer drop area:

- The core-card element
- The core-item element
- The paper-input element
- The paper-button element

After dropping and aligning these elements inside the card, we have made some changes to the styles and attribute of these elements. The following screenshot shows the style and attribute section for paper-button only. Similarly, we can change other elements:

The screenshot shows the Polymer DevTools element inspector for a `<paper-button>` element. The left pane displays the `PROPERTIES` tab, listing attributes like `raised`, `recenteringTouch`, `fill`, `active`, `focused`, `pressed`, `disabled`, `toggle`, and `textContent`. The right pane displays the `STYLES` tab, which contains several style properties with their current values:

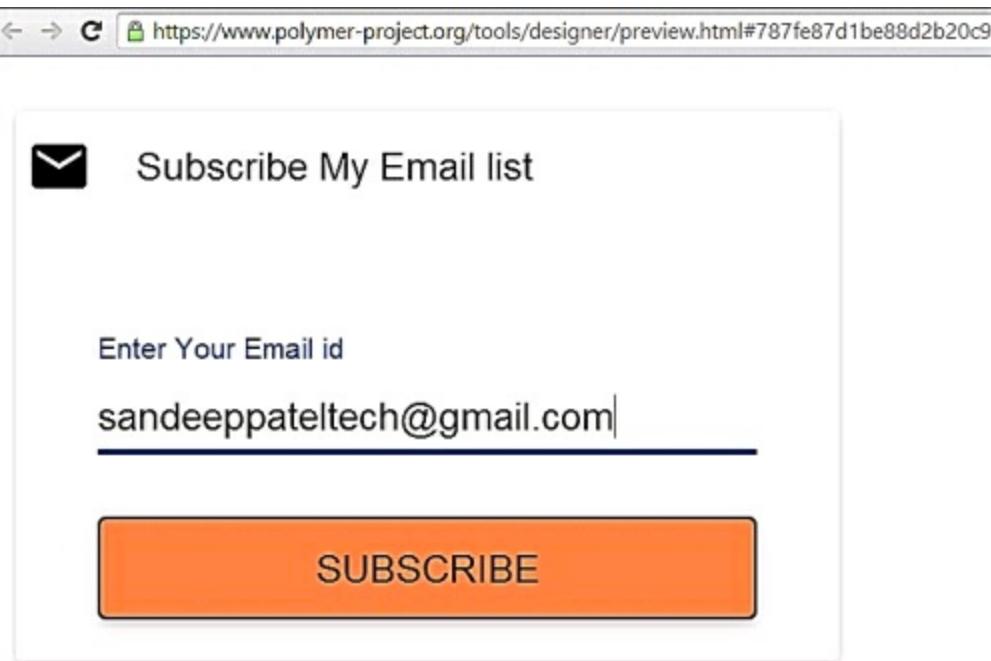
Property	Value
layout	reverse
justify	wrap
bg color	orange
border	1px solid
opacity	0.5
padding	10px
margin	5% 10%
position	left
left	0
top	0
display	block
width	100px
height	40px

A `Subscribe` button is also visible in the `textContent` field.

After the changes are made to styles and attributes, we can see the generated code by using the **Toggle Code/Design** option. The generated code is shown in the following screenshot:

```
1 <link rel="import" href="../core-item/core-item.html">
2 <link rel="import" href="../paper-input/paper-input.html">
3 <link rel="import" href="../paper-button/paper-button.html">
4 <polymer-element name="my-element">
5   <template>
6     <style></style>
57   <core-card id="core_card" layout vertical>
58     <core-item id="core_item" icon="mail"
59       label="Subscribe My Email list"
60       horizontal center layout></core-item>
61     <paper-input label="Enter Your Email id"
62       floatinglabel id="paper_input"
63       center>
64     </paper-input>
65     <paper-button raised id="paper_button">
66       Subscribe
67     </paper-button>
68   </core-card>
69 </template>
70 <script>
71   Polymer({});
```

The preview of the design can be seen by using the **Launch Preview** option. The following screenshot shows the live preview of the developed design in a new window:



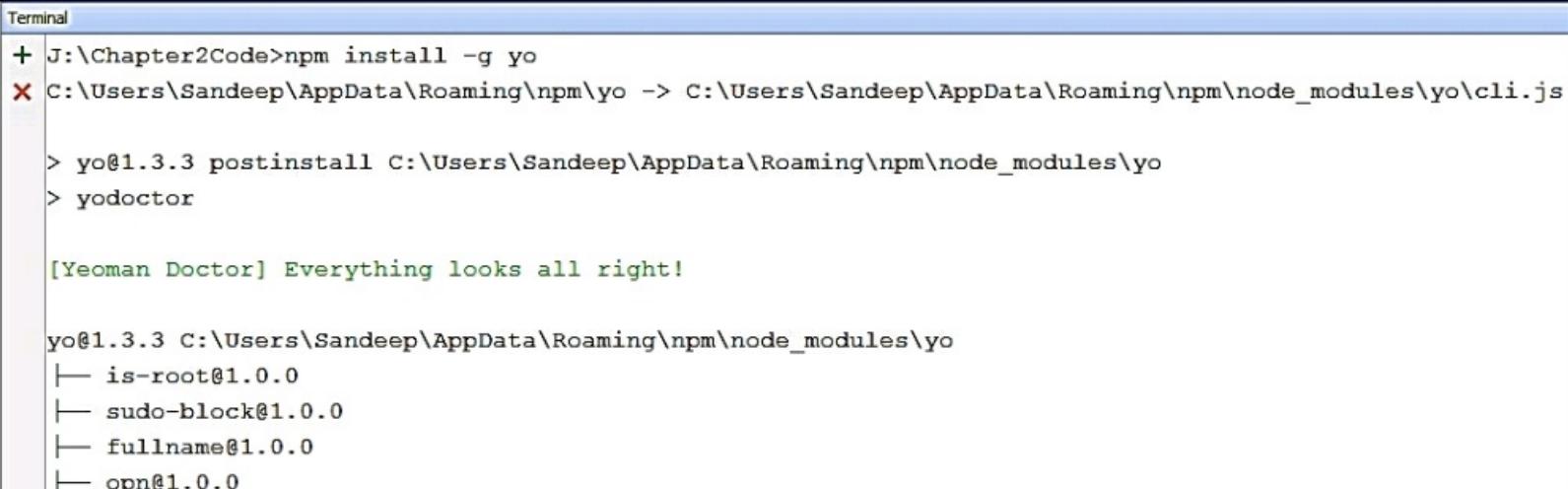
# Yeoman Polymer generator

The Yeoman tool provides the generator ecosystem for web developers in order to reduce the configuration work and get focused on web application logic. Yeoman helps in increasing productivity of a developer by getting a quick kick start to the project and implementing best practices and required boilerplate code to the project.

Yeoman can be installed using npm. The following code shows the command for installing Yeoman in the system:

```
npm install -g yo
```

The following screenshot shows the command prompt with the Yeoman installation in progress:



A screenshot of a terminal window titled "Terminal". The command "npm install -g yo" is being run. The output shows the package being copied from the local directory to the user's AppData folder, followed by postinstall and yodoctor steps, and a confirmation message from the Yeoman Doctor. Finally, the package structure is listed, showing dependencies like is-root, sudo-block, fullname, and opn.

```
+ J:\Chapter2Code>npm install -g yo
x C:\Users\Sandeep\AppData\Roaming\npm\yo -> c:\Users\Sandeep\AppData\Roaming\npm\node_modules\yo\cli.js

> yo@1.3.3 postinstall C:\Users\Sandeep\AppData\Roaming\npm\node_modules\yo
> yodoctor

[Yeoman Doctor] Everything looks all right!

yo@1.3.3 C:\Users\Sandeep\AppData\Roaming\npm\node_modules\yo
└── is-root@1.0.0
└── sudo-block@1.0.0
└── fullname@1.0.0
└── opn@1.0.0
```

Yeoman provides different generators for different types of applications. Each generator has a specific work flow for accomplishing a part or complete configuration of an application. These generators run with the `yo` command. Once Yeoman is installed in the system, the Polymer generator can be installed using the following command:

```
npm install -g generator-polymer
```

The following screenshot shows the command prompt with the polymer-generator installation in progress:

```
Terminal
+ J:\Chapter2Code>npm install -g generator-polymer
generator-mocha@0.1.6 C:\Users\Sandeep\AppData\Roaming\npm\node_modules\generator-mocha
└── yeoman-generator@0.17.7 (dargs@2.1.0, diff@1.2.0, class-extend@0.1.1, rimraf@2.2.8, underscore@1.0.0, mime@1.2.11, text-table@0.2.0, debug@1.0.4, async@0.9.0, nopt@3.0.1, isbinaryfile@2.1.0, shelljs@0.3.0, chalk@0.5.1, mkdirp@0.5.0, run-async@0.1.0, lodash@2.4.1, iconv-lite@2.0.2, request@2.51.0, file-utils@0.2.1, gruntfile-editor@0.2.0, cheerio@0.17.0, download@1.0.7)

generator-polymer@0.6.1 C:\Users\Sandeep\AppData\Roaming\npm\node_modules\generator-polymer
├── rimraf@2.2.8
├── underscore.string@2.4.0
├── ncp@1.0.1
└── lodash@2.4.1
```

## The polymer-generator commands

The polymer-generator provides the following utilities to support the Polymer-based application development:

- Polymer application generator
- Polymer element generator
- Polymer seed generator
- Polymer GitHub page generator

### The Polymer application generator

A new Polymer application can be created using the Polymer application generator. The steps for creating a new Polymer application are as follows:

1. We need to create a new project directory .The following command can be used to create a new project directory:

```
J:\>mkdir PolymerAppDemo
```

Change the directory to `PolymerAppDemo` using the following command:

```
J:\>cd PolymerAppDemo
```

2. After changing the directory, we can run the command to generate a Polymer application setup. The setup process asks a set of questions, which need to be answered by the developer. Once these questions are answered, it creates a Polymer application with all the required library and dependencies installed. The command for generating the Polymer application is as follows:

J:\PolymerAppDemo>yo polymer

The following screenshot shows the command prompt with the Polymer application-generator in progress:

```
Terminal
+ J:\PolymerAppDemo>yo polymer

      _-----.
     |       |
     |---(o)--|   | Out of the box I include |
     '-----'   |   HTML5 Boilerplate and   |
      ( _ 'U`_ )   |       Polymer       |
      /__A__\   '-----'
      | ~ |
      | .--. |
      ' .-' ' Y '


? Would you like to include core-elements? Yes
? Would you like to include paper-elements? No
? Would you like to use SASS/SCSS for element styles? (Y/n) n
```

## The Polymer element generator

The Polymer element generator can be used to create a boilerplate for developing a custom element. We will learn about this generator for creating custom elements in the next chapter.

If you want to find out more about the Polymer element generator, refer to <https://github.com/yeoman/generator-polymer#element-alias-el>.

## The Polymer seed generator

This generator can be used for creating a custom standalone element, which is intended to be shared with others using Bower. You will learn more about this generator in the next chapter.

If you want to find out more about the Polymer seed generator at this point, refer to <https://github.com/yeoman/generator-polymer#seedc>.

## The Polymer GitHub page generator

This generator can be used for creating a GitHub page for the standalone seed element, to share its details with the developer community. We will learn about this GitHub page generator in the next chapter.

If you want to find out more about Polymer GitHub page generator at this point, refer to <https://github.com/yeoman/generator-polymer#gh>.

# Summary

In this chapter, you learned about the installation of the Polymer library. We have explored the architecture of Polymer library followed by a section on core and paper elements. In the final section of this chapter, you were introduced to the Polymer designer tool. In the next chapter, you will learn to develop a custom element using the Polymer library.

# Chapter 3. Developing Web Components Using Polymer

In the previous chapter, you learned how to install and configure the Polymer library in a web page. Now it is time to explore a few more useful features and learn to develop a custom element.

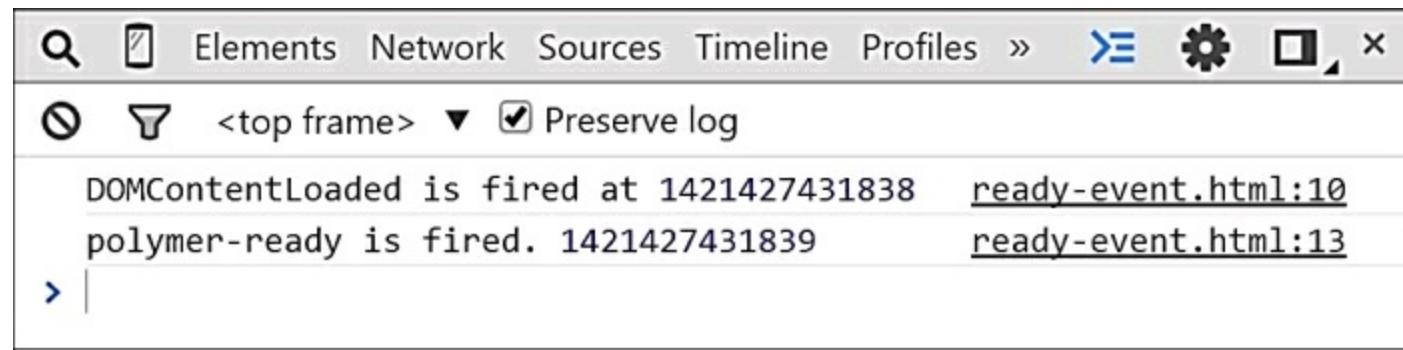
## PolymerJS ready event

The Polymer library can be included to a page by importing a `polymer.html` file. Polymer checks each and every element definition that registers it asynchronously. If we try to render the element before this, it will create the **Flash of Unstyled Content (FOUC)** problem. To resolve this issue, we need to catch the `polymer-ready` event. We can hide the page elements before Polymer is ready, and once Polymer is ready, we can make them visible. The following code shows the use of the `polymer-ready` event:

```
<!DOCTYPE html>
<html>
<head>
    <link rel="import" href="../bower_components/polymer/polymer.html">
    <title>Polymer ready event listener example</title>
</head>
<body>
<script>
    document.addEventListener("DOMContentLoaded", function(event) {
        console.log("DOMContentLoaded is fired at", new Date().getTime());
    });
    window.addEventListener('polymer-ready', function(e) {
        console.log("polymer-ready is fired.", new Date().getTime());
    });
</script>
</body>
</html>
```

In the preceding code, we have two callback methods listening to `DOMContentLoaded` and `polymer-ready` events. In the callback code, the console logs the time when the event is fired. The following screenshot shows the output of the preceding code, showing the time when `DOMContentLoaded` and `polymer-event`

are fired:



From the preceding screenshot, it is evident that once DOM is loaded, Polymer starts registering the element, and once it is completed, it fires the `polymer-ready` event.

# Polymer expressions

The separation of concern during development is an age-old topic of discussion. The goal of separation concern is to separate the rendering logic from the markup. It reduces the code maintenance cost and increases productivity. Polymer provides expression features that can be used inline with HTML code, while the computation logic is present in another JavaScript file. The syntax for a Polymer expression is as follows:

```
{ {Expression} }
```

In the preceding syntax, the Polymer expression is wrapped around two curly braces. There are a few points to note down about Polymer expression:

- Polymer expression is used as inline with HTML code for simple text value
- Polymer expression should not be used for HTML value
- The `Eval` method cannot be used inside the expression

Polymer supports many operations for working with expression. These operations are as follows:

- **Identifiers and paths:** This represents the object and the properties present inside the current page scope. Normally, paths are represented using dot (.) operator. The following code shows an example of identifier and paths:

```
{ {student.name} }  
{ {student.subject.mark} }
```

- **Array access:** This represents the members present inside an array object. Normally, array access can be represented using a square ([]) bracket. The following code shows an example of array access:

```
{ {student[0].name} }
```

- **Logical not operator:** This represents the Boolean operator to negate the truth value. The following code shows an example of not operator:

```
{ {!flag} }
```

- **Unary operators:** This represents the unary operators like + and - to work with single operand. The following code shows an example of unary operator:

```
{ {-number} }
```

- **Binary operators:** This represents the binary operators to work with two operands. The following code shows an example of binary operator:
 

```
{ {number1*number2} }
```
- **Comparators:** This represents <, >, <=, >=, ==, !=, ===, !==, and returns a Boolean value. The following code shows an example of comparator:
 

```
{ {number1 < number2} }
```
- **Logical comparators:** This represents logical operators like AND (&&) and OR(||), and returns a Boolean value. The following code shows an example of logical comparator:
 

```
{ {variable1 && variable2} }
```

- **Ternary operator:** This represents ternary operators like the if condition. The following code shows an example of ternary operator:
 

```
{ {number1 > number2 ? "number1 is greater" : "number2 is greater"} }
```

- **Parenthesis:** This helps in grouping the expression to be evaluated. The following code shows an example of parenthesis:
 

```
{ { (number1 + number2)*number3} }
```

- **Literal values:** This represents literal identifiers like null and undefined. The following code shows an example of literals:
 

```
{ {null} }
```

- **Array and object initializers:** This represents the anonymous array and object declaration. The following code shows an example of array initializer:
 

```
{ { ["Apple", "Orange"] } }
```

- **Function:** A function can be called using Polymer expression. The following code shows an example of a function:
 

```
{ {someFunction()} }
```

## Note

To find out more about Polymer expression, refer to <https://www.polymer-project.org/docs/polymer/expressions.html>.

We can also use filters on the Polymer expression. Before jumping into filter

expression, let's learn about the Polymer template with auto-binding. The reason for looking at auto-binding first is that the examples in the coming section are independent and run in the current page scope.

## Polymer templating with auto-binding

In [Chapter 1, Introducing Web Components](#), we learned about the `<template>` element defined in the W3C web component specification. Polymer adds some additional features, such as expressions and template binding. This means we can use expressions inside a template, which can be rendered to the browser.

The Polymer templating works differently than the traditional templating. The DOM model of a template instance remains in the browser as long as the respective data is in use. If there are any changes to the data, then Polymer compares and modifies the specific part of DOM. It results in the smallest changes into the DOM tree.

Polymer templating can be used on a page level by using the auto-binding feature. As we know, the `<template>` content is inert in nature until it is activated. Polymer template comes with a solution called auto-binding, where the template gets activated by considering the template, and the data model is present on the template itself. A template can be used as auto-bound by using the `is` attribute auto-binding value. The syntax for the auto-binding a template is as follows:

```
<template is="auto-binding">  
</template>
```

Let's check out a simple example of auto-binding a template. The code for the usage of the auto-binding feature is as follows:

```
<!DOCTYPE html>  
<html>  
<head>  
    <script  
src="..../bower_components/webcomponentsjs/webcomponents.min.js"></script>  
    <link rel="import" href="..../bower_components/polymer/polymer.html">  
    <title>Polymer template auto binding example</title>  
</head>  
<body>  
    <template is="auto-binding">  
        <h1>This is an auto binding example</h1>  
    </template>
```

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

In the preceding code, the template has an auto-binding property and some content within the `h1` element. If we run this code in a browser, the `<template>` element gets activated due to the auto-binding feature, and then the content is rendered. The following screenshot shows the output and HTML inspection of the preceding code:

The screenshot shows a browser window with the URL `localhost:8080/Chapter3Code/polymer-template/auto-binding.html`. The page title is "This is an auto binding example". Below the page content, the browser's developer tools are open, specifically the "Elements" tab. The DOM tree is displayed, showing the following structure:

```
<!DOCTYPE html>
▼ <html>
  ▶ <head>...</head>
  ▼ <body>
    ▼ <template is="auto-binding" bind>
      ▼ #document-fragment
        <h1>This is an auto binding example</h1>
      </template>
      <h1>This is an auto binding example</h1>
    </body>
```

The "Elements" tab is selected, and the node `#document-fragment` under the `<template>` node is highlighted. The status bar at the bottom shows the selected elements: `html body template #document-fragment h1`.

In the preceding screenshot, we can see that the HTML instance markup generated by the template is appended just after it. This is a unique feature of a template with auto-binding. The `<template>` element has the `bind` property attached to itself as the template is auto-bound.

## Note

To find out more about the Polymer template auto-binding feature, refer to <https://www.polymer-project.org/docs/polymer/databinding-advanced.html#autobinding>.

# Polymer template attributes

Polymer provides additional features to the `<template>` element to make it more useful. Polymer derives its templating mechanism from the `TemplateBinding` library. You can find more details about this library at <https://github.com/polymer/TemplateBinding>. It provides many useful attributes to help in data binding. These attributes are as follows:

- `bind`: This attribute can be used to bind a template to a JavaScript object.
- `repeat`: This attribute can be used to iterate a template by binding the template to a list.
- `if`: This attribute can be used to check a logical condition by comparing a property of the bound JavaScript object.
- `ref`: This attribute can be used to include another template in the current template. The `ref` attribute takes the ID of other templates to include it to the current template.

Let's check out an example to help understand the use of these attributes in a template element. The following code shows the use of these attributes:

```
<!DOCTYPE html>
<html>
<head>
    <script
src="../bower_components/webcomponentsjs/webcomponents.min.js"></script>
    <link rel="import" href="../bower_components/polymer/polymer.html">
    <title>Polymer template attributes example</title>
</head>
<body>

<template id="template1" is="auto-binding"
          if="{{myData.myFlag}}">
    <h1>The value of myFlag is {{myData.myFlag}}</h1>
</template>
<template id="template2" is="auto-binding"
          repeat="{{studentName in students}}">
    <h1>{{studentName}}</h1>
</template>
<template id="template3" is="auto-binding"
          repeat="{{sub in subjects}}">
    <template bind="{{sub}}>
        <h1>Subject name is {{name}} and country is {{country}}</h1>
    </template>
</template>
```

```

<template id="template4">
  <h1>Hello world !!!</h1>
</template>
<template id="template5" is="auto-binding"
          ref="template4">
</template>

<script>
  (function () {
    var template1 = document.querySelector("#template1"),
        template2 = document.querySelector("#template2"),
        template3 = document.querySelector("#template3");
    template1.myData = {
      "myFlag": true
    };
    template2.students = ["Sandeep", "Sangeeta", "Surabhi",
"Sumanta"];
    template3.subjects = [{"name": "Computer", "country": "India"}]
  })();
</script>
</body>
</html>

```

The details of the preceding code are as follows:

- The template element with the `template1` ID has the `myData` model, which has the `myFlag` property with true Boolean value. The template has the `if` attribute, which checks the `myFlag` value and displays the HTML content of `template1`.
- The template element with the `template2` ID has the `students` model containing an array of names. The `template2` element has the `repeat` attribute to iterate the array of names to render in the browser.
- The template element with the `template3` ID is bound with the `subjects` model containing objects having `name` and `country` property. The `template3` ID contains another nested template, which is bound to the `sub` instance of `subject` to display `name` and `country` values inside it.
- The template element with the `template5` ID has the `ref` attribute referring to `template4`. The `template4` ID has a text message wrapped inside the `h1` element. On activation of the `template5` element, the contents of `template4` are included and rendered in the browser.

The output of the preceding code is listed in the following screenshot, which shows the use of the mentioned attributes:

**The value of myFlag is true**

Sandeep

Sangeeta

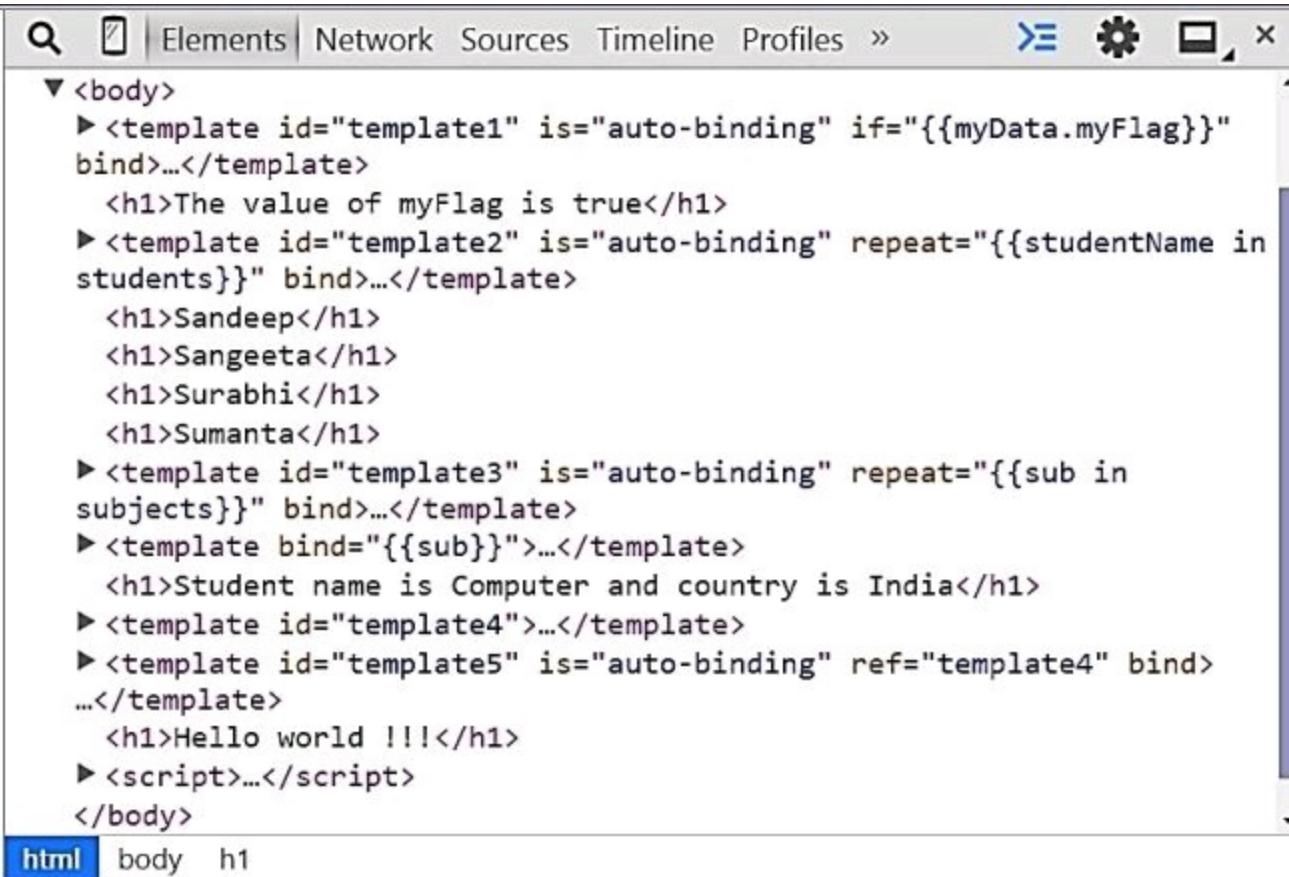
Surabhi

Sumanta

**Subject name is Computer and country is India**

**Hello world !!!**

The following screenshot shows the Chrome developer console, showing template attributes in action, and the generated HTML instances are inserted just after the template:



The screenshot shows the Chrome Developer Console with the "Elements" tab selected. The DOM tree is expanded to show the structure of the generated HTML. At the top level, there is a <body> element. Inside it, there are several <template> elements with various attributes like "is='auto-binding'", "if='{{myData.myFlag}}'", "repeat='{{studentName in students}}'", and "repeat='{{sub in subjects}}'". These templates are expanded to show their rendered content: "The value of myFlag is true", four student names (Sandeep, Sangeeta, Surabhi, Sumanta), a subject name ("Computer"), and a greeting ("Hello world !!!"). There is also a <script> node present. The bottom of the screenshot shows the status bar with tabs for "html", "body", and "h1".

```

▼ <body>
  ► <template id="template1" is="auto-binding" if="{{myData.myFlag}}"
  bind>...</template>
    <h1>The value of myFlag is true</h1>
  ► <template id="template2" is="auto-binding" repeat="{{studentName in
  students}}" bind>...</template>
    <h1>Sandeep</h1>
    <h1>Sangeeta</h1>
    <h1>Surabhi</h1>
    <h1>Sumanta</h1>
  ► <template id="template3" is="auto-binding" repeat="{{sub in
  subjects}}" bind>...</template>
  ► <template bind="{{sub}}>...</template>
    <h1>Student name is Computer and country is India</h1>
  ► <template id="template4">...</template>
  ► <template id="template5" is="auto-binding" ref="template4" bind>
  ...</template>
    <h1>Hello world !!!</h1>
  ► <script>...</script>
</body>

```

html body h1

# Filtering expression

Polymer provides filter support to work with the expression. Filters are useful to modify the output of an expression. The syntax of using filter with an expression is as follows:

```
{ {Expression | filterName} }
```

In the previous code, an expression and filter is combined using a bar (|) symbol. The filter function takes the value of the given expression and modifies it based on the filter logic, and then renders it in the browser.

## Built-in filtering expression

Polymer has two built-in filters to work with the expression. These two predefined filters are `tokenList` and `styleObject`. Let's explore these built-in filters in more detail.

### The TokenList filter

The `tokenList` filter can be used for adding and removing a string based on the supplied object. It is a really good fit to programmatically modify the class names. However, for demonstration, we have used a string. The following code shows the use of the `tokenList` filter with an expression:

```
<!DOCTYPE html>
<html>
<head>
  <script
src="../bower_components/webcomponentsjs/webcomponents.min.js"></script>
  <link rel="import" href="../bower_components/polymer/polymer.html">
  <title>PolymerJS tokenList Builtin Filter Expression</title>
</head>
<body>
  <template id="template1" is="auto-binding">
    <h1>Name : {{student.name}}</h1>
    <h1>Score : {{student.score}}</h1>
    <h1>Result : {{ {"Passed": student.score > 60} | tokenList}}</h1>
  </template>
  <script>
    (function() {
      var template1 = document.querySelector("#template1");
      template1.student= {
```

```

        "name": "Sandeep",
        "score": 70
    }
})();
</script>
</body>
</html>

```

The details of the preceding code are as follows:

- In the preceding code, `template1` is attached with the `student` object having two properties—`name` and `score`.
- Inside `template1`, the score of the student is compared against a passing marks `60` with the built-in filter `tokenList`. If the score is more than `60`, the **Passed** string is rendered in place of an expression.

The following screenshot shows the output of the preceding code containing the name of the student, score of the student, and their result. The score of the student is `70`, and compared with the pass mark of `60`, it returns true and the **Passed** string is rendered in a browser.

The screenshot shows a browser window displaying the output of the template expression. The page content is as follows:

```

Name : Sandeep
Score : 70
Result : Passed

```

Below the browser window, the developer tools panel shows the DOM tree and the generated HTML code. The DOM tree is as follows:

```

<!DOCTYPE html>
▼<html>
  ▶<head>...</head>
  ▼<body>
    ▶<template id="template1" is="auto-binding"
      bind>...</template>
      <h1>Name : Sandeep</h1>
      <h1>Score : 70</h1>
      <h1>Result : Passed</h1>
    ▶<script>...</script>
  </body>

```

The status bar at the bottom indicates the current tab is 'html'.

## The styleObject filter

The `styleObject` filter is used for converting a JSON object into a string of key value pair. This filter is a good fit for working with the `style` attribute of an element. The following code shows the use of the `styleObject` filter with an expression:

```
<!DOCTYPE html>
<html>
<head>
    <script
src="../bower_components/webcomponentsjs/webcomponents.min.js"> </script>
    <link rel="import" href="../bower_components/polymer/polymer.html">
    <title>PolymerJS styleObject Builtin Filter Expression</title>
</head>
<body>
    <template id="template1" is="auto-binding">
        <div style="{{myStyle | styleObject}}>Name : {{student.name}}</div>
    </template>
    <script>
        (function() {
            var template1 = document.querySelector("#template1");
            template1.student= {
                "name":"Sandeep"
            };
            template1.myStyle= {
                "color": "tomato",
                "font-size": "30px",
                "height": "40px",
                "width": "200px",
                "border": "1px solid green",
                "padding": "20px"
            };
        })();
    </script>
</body>
</html>
```

The details of the preceding code are as follows:

- The `myStyle` property of the `student` object has a set of key/value pairs, containing CSS properties
- The template expression is used with the `styleObject` filter to convert the

key/value pairs to complete the CSS string

The output of the preceding code is listed in the following screenshot, where we can see that the `myStyle` object's key/value pairs are converted to CSS string and gets applied to the `style` attribute of the `div` element:

The screenshot shows a browser window with the URL `localhost:8080/Chapter3Code/filter-expression/styleobject-filter-expression.html`. The page displays the text "Name : Sandeep" in a red font, centered within a green-bordered box. Below the page content, the browser's developer tools are open, specifically the Elements tab. The tree view shows the structure of the HTML document, including the `<template>`, `<div>`, and `<script>` elements. The status bar at the bottom indicates the selected node is a `div` element.

```
<!DOCTYPE html>
<html>
  <head>...</head>
  <body>
    <template id="template1" is="auto-binding" bind>...</template>
    <div style="color: tomato; font-size: 30px; height: 40px; width: 200px; border: 1px solid green; padding: 20px">Name : Sandeep</div>
    <script>...</script>
  </body>
</html>
```

## Custom filtering expression

Polymer supports development of our own custom filters to be used in the expression. Let's check out an example of creating a custom filter. The code for creating a custom filter and its use is as follows:

```
<!DOCTYPE html>
<html>
<head>
  <script
src="../bower_components/webcomponentsjs/webcomponents.min.js"> </script>
  <link rel="import" href="../bower_components/polymer/polymer.html">
  <title>PolymerJS Custom Filter Expression</title>
</head>
<body>
  <template id="template1" is="auto-binding">
```

```
<h1>{{student.name | sayHello}}</h1>
</template>
<script>
  (function() {
    var template1 = document.querySelector("#template1");
    template1.student= {
      "name":"Sandeep"
    };
    template1.sayHello= function(inputString) {
      return "Hello " + inputString;
    };
  })();
</script>
</body>
</html>
```

The details of the preceding code are as follows:

- The `template1` variable has an expression for displaying the name of the student with a custom filter named `sayHello`
- The definition of the `sayHello` filter method is present as a value of the `sayHello` attribute
- The `sayHello` filter method takes an expression value and prepends the `Hello` text before the name of the student

The following screenshot shows the output of the preceding code, where the `Hello` string is prepended to the name **Sandeep** due to the `sayHello` filter:

# Hello Sandeep

Elements Network Sources Timeline Profiles Resources

```
<!DOCTYPE html>
▼ <html>
  ▶ <head>...</head>
  ▼ <body>
    ▼ <template id="template1" is="auto-binding" bind>
      ▼ #document-fragment
        <h1>{{student.name | sayHello}}</h1>
      </template>
      <h1>Hello Sandeep</h1>
    ▶ <script>...</script>
  </body>
```

html body

## Global filtering expression

Polymer supports defining global filters to increase their reusability. A global filter is available as an API to the application developer. A global filter can be created by using the `PolymerExpression` object. A new filter can be added to the Polymer by creating a new property of the `PolymerExpression` object by using the `prototype` attribute. The code for creating a global filter is as follows:

```
<!DOCTYPE html>
<html>
<head>
  <script
src="../bower_components/webcomponentsjs/webcomponents.min.js"> </script>
  <link rel="import" href="../bower_components/polymer/polymer.html">
  <title>PolymerJS Global Custom Filter Expression</title>
</head>
<body>
  <template id="template1" is="auto-binding">
    <h1>{{student.name | sayBye}}</h1>
```

```

</template>
<script>
    (function() {
        var template1 = document.querySelector("#template1");
        template1.student= {
            "name":"Sandeep"
        };
        PolymerExpressions.prototype.sayBye = function(inputString) {
            return "Bye " + inputString;
        };
    })();
</script>
</body>
</html>

```

In the preceding code, a global filter named `sayBye` is created by defining a new property in `PolymerExpression.prototype`. The output of the preceding code is listed in the following screenshot:

The screenshot shows a browser window with the URL `localhost:8080/Chapter3Code/filter-expression/globalcustom-filter-expression.html`. The main content area displays the text **Bye Sandeep**. Below the content is the browser's developer tools element inspector. The 'Elements' tab is selected. The tree view shows the following structure:

- <!DOCTYPE html>
- ▼ <html>
  - <head>...</head>
  - ▼ <body>
    - ▼ <template id="template1" is="auto-binding" bind>
      - ▼ #document-fragment
        - <h1>{{student.name | sayBye}}</h1>
      - </template>
      - <h1>Bye Sandeep</h1>
    - <script>...</script>

At the bottom of the inspector, the tabs 'html' and 'body' are visible, with 'html' being the active tab.

# Developing Polymer custom elements

Polymer has built-in elements—core and paper. Polymer also supports development of our own custom element. Some of the benefits of developing custom elements are as follows:

- They are reusable
- They reduce the code size of the application
- They increase the developer's productivity
- They help in achieving encapsulation by hiding the definition of the element

## Note

In Polymer, everything is an element.

In the following section, we will learn to develop a custom element. The steps for developing a custom element are as follows:

1. Defining a custom element
2. Defining element attributes
3. Defining default attributes
4. Defining public properties and methods
5. Publishing properties
6. Defining a lifecycle method
7. Registering a custom element

## Defining a custom element

A custom element can be defined using the `<polymer-element>` tag. The code syntax for developing a custom element is as follows:

```
<polymer-element name="tag-name" constructor="TagName">
  <template>
    <!-- shadow DOM here -->
  </template>
  <script>
    Polymer({
      // properties and methods here
    });
  </script>
</polymer-element>
```

The details of the preceding syntax are as follows:

- In the preceding code, a custom element definition has been wrapped within the `<polymer-element>` tag
- The definition contains the `<template>` element containing the HTML markup for the custom element
- The definition also contains the `<script>` element containing the properties and methods for the custom element
- The `<polymer-element>` tag takes the following attributes while defining a new custom element:
  - Name: This is a required field and represents the name of the custom element. The name must be separated with a hyphen (-).
  - Attributes: This is an optional field and can be used for defining the attribute that the custom element can have.
  - Extends: This is an optional field and can be used for extending another element through inheritance.
  - Noscript: This is an optional attribute and can be used by a custom element while defining an attribute, when it does need any properties and methods. Put more simply, it creates a simple element that has only name and constructor.
  - Constructor: This is an optional attribute and represents the name of the constructor that can be used by the programmer to create a new instance of the custom element, using the `new` keyword.

## Defining element attributes

Custom attributes can be defined using the `attribute` property of `<polymer-element>`. This can take multiple attribute names separated by a *space*. A property defined inside the `attribute` property is published by default .The syntax of the attribute declaration is as follows:

```
<polymer-element name="tag-name"  
    attributes="property1 property2... ">  
</polymer-element>
```

## Defining default attributes

We can define default attributes to a custom element. The default attributes directly get attached with the custom element while rendering. The syntax for

defining a default attribute is as follows:

```
<polymer-element name="tag-name" myDefaultProperty1>
</polymer-element>
```

In the preceding code a default property named `myDefaultProperty1` is defined. To define any default properties just add the name to the `<polymer-element>` element as an attribute.

## Defining public properties and methods

We can define public properties and methods for a custom element while also defining it. The following code shows the syntax of declaring public properties and methods:

```
<polymer-element name="tag-name">
  <script>
    Polymer({
      message: "Hi!",
      get greeting() {
        return this.message + ' You are welcome.';
      },
      sayBye : function() {
        return ' Bye.';
      }
    });
  </script>
</polymer-element>
```

The details of the preceding code are as follows:

- In the mentioned definition, `message` is a property with a `get` methods `greeting` function. The `get` method can access the `message` property with the `this` keyword. The `this` keyword refers to the custom element `<tag-name>`.
- In the previous definition, the `sayBye` method is a public function and returns a `Bye` string.

## Publishing properties

The properties defined inside the attribute are by default public. There is another way to publish the properties of a custom element—using the `publish` attribute. The syntax for publishing properties is as follows:

```
<polymer-element name="tag-name">
```

```

<script>
Polymer({
  publish: {
    property1: "value1",
    property2: "value2",
  }
});
</script>
</polymer-element>

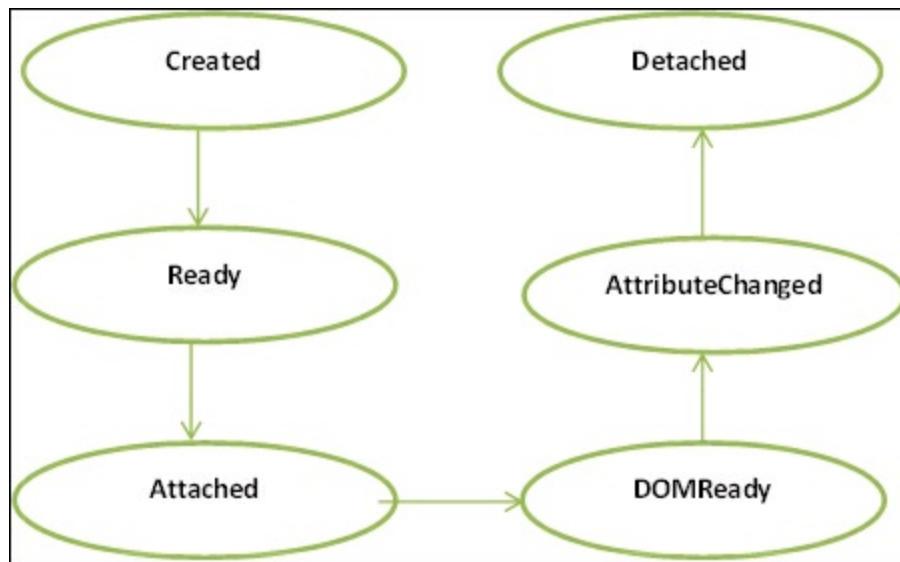
```

In the preceding code, `property1` and `property2` are published using the `publish` keyword. Publishing properties using the attribute approach is the preferable way, as it follows a declarative method. However, the use of the `publish` keyword approach can be selected in if the following statements are true:

- If there are long lists of attributes that need to be published for a custom element
- If we need to define default values for the properties
- If we need two-way declarative binding to a published property

## Defining a lifecycle method

The Polymer element goes through different states during its lifecycle. The following diagram shows the possible states of a custom element:



The different states of a custom element are as follows:

- **Created:** This is the state when an instance of the custom element is created

- **Ready**: This is the state when the shadow DOM is ready, event listeners are attached, and the property observers are set up
- **Attached**: This is the state when the custom element is attached to the DOM
- **DOMReady**: This is the state when the initial sets of custom element children are exists in DOM
- **AttributeChanged**: This is the state when one of the attribute values is changed
- **Detached**: This is the state when the custom element is removed from the DOM

The syntax of the callback methods for the lifecycle states, to define our own custom logic, is as follows:

```
Polymer('tag-name', {
  created: function() {
    //Code for created state callback
  },
  ready: function() {
    //Code for ready state callback
  },
  attached: function () {
    //Code for attached state callback
  },
  domReady: function() {
    //Code for domReady state callback
  },
  detached: function() {
    //Code for detached state callback
  },
  attributeChanged: function(attrName, oldVal, newVal) {
    // code for attribute changed state callback
  }
});
```

In the preceding syntax, we can see the callback method representing each lifecycle state of a custom element.

## Registering a custom element

The native way of defining a custom element is using the `Polymer` method in the `<polymer-element>` tag. The syntax of registering an element is as follows:

```
Polymer([ tag-name, ] [prototype]);
```

# Developing a sample custom element

Now, it is time to develop a simple `<say-hello>` element with a `mytext` attribute with the default text `World!!!` and a template to render in the browser. The code definition for the `<say-hello>` element is present inside the `hello-component.html` file and is as follows:

```
<polymer-element name="say-hello" constructor="SayHello"
attributes="mytext">
  <template>
    <h4>Hello {{mytext}} !!!</h4>
  </template>
  <script>
    Polymer({
      created: function() {
        this.mytext = "World"
      }
    });
  </script>
</polymer-element>
```

The details of the preceding code are as follows:

- A new custom element `<say-hello>` is defined using the `<polymer-element>` tag.
- The custom element has the `mytext` attribute, which is published as public and initialized with a default value `world` using the `created` callback method.
- The custom element has a constructor named `SayHello`. By using this constructor, we can programmatically create an instance of a `SayHello` element using the `new` keyword. The syntax for using a `new` keyword is as follows:

```
var sayHello1 = new SayHello();
```

The following code shows the use of the `<say-hello>` element in an HTML page:

```
<!DOCTYPE html>
<html>
<head>
  <script
src="../bower_components/webcomponentsjs/webcomponents.min.js"> </script>
  <link rel="import" href="../bower_components/polymer/polymer.html">
  <link rel="import" href="hello-component.html">
<title>Polymer sayHello component demo</title>
```

```
</head>
<body>
  <say-hello>
  </say-hello>
  <say-hello mytext="John"></say-hello>
  <script>
    //Using SayHello Constructor
    window.addEventListener('polymer-ready', function(e) {
      var sayHello1 = new SayHello();
      sayHello1.mytext = "PolymerJS"";;
      document.body.appendChild(sayHello1);
    });
  </script>
</body>
</html>
```

In the preceding code, we have called the `<say-hello>` element in the following three different ways:

- Without the `mytext` attribute
- With the `mytext` attribute value as `John`
- Using the JavaScript constructor `new SayHello` and assigning the `mytext` property to the `Polymer` string

The output of the preceding code is shown in the following screenshot with three different message generated by the `<say-hello>` element used:

**Hello World !!!**

**Hello John !!!**

**Hello PolymerJS !!!**

Elements | Network Sources Timeline Profiles Resources

```
▼ <body>
  ► <say-hello>...</say-hello>
  ► <say-hello mytext="John">...</say-hello>
  ► <script>...</script>
  ▼ <say-hello>
    ▼ #shadow-root
      | <h4>Hello PolymerJS !!!</h4>
    </say-hello>
  </body>
```

html body

# Extending a custom element

A custom element can be extended from another element using the `extends` attribute. We can only extend from one element at a time. Let's develop an element named `<say-good>`, which is extended from `<say-hello>`. The following code presents the `good-component.html` file and has the definition of `<say-good>` custom element:

```
<link rel="import" href="hello-component.html">
<polymer-element name="say-good" extends="say-hello"
    constructor="SayGood" attributes="time">
    <template>
        <shadow></shadow>Good {{time}}
    </template>
    <script>
        Polymer({
            created:function(){
                this.time="Morning";
            }
        });
    </script>
</polymer-element>
```

The details of the preceding code are as follows:

- A new custom element named `<say-good>` is created by extending the `<say-hello>` element with a published property named `time`
- The default value of the `time` property is assigned to `Morning` using the `created` callback method
- The template of the `<say-good>` element contains a **shadow insertion point** for the parent element markup with a text message `Good` and an expression `{{time}}`

The following code shows the use of the `<say-good>` element:

```
<!DOCTYPE html>
<html>
<head>
    <script
src="../bower_components/webcomponentsjs/webcomponents.min.js"> </script>
    <link rel="import" href="../bower_components/polymer/polymer.html">
    <link rel="import" href="good-component.html">
<title>Polymer sayGood component demo</title>
```

```
</head>
<body>
  <say-good mytext="John" time="Night">
  </say-good>
  <script>
    //Using SayGood Constructor
    window.addEventListener('polymer-ready', function(e) {
      var sayGood1 = new SayGood();
      sayGood1.mytext = "Smith";
      sayGood1.time = "Afternoon";
      document.body.appendChild(sayGood1);
    });
  </script>
</body>
</html>
```

The details of the preceding code are as follows:

- The `<say-good>` element is called using `mytext` and the `time` attribute with values `John` and `Night`, respectively.
- An object of the `SayGood` element is initialized using its constructor with the `new` keyword. The `mytext` object property and `time` is then assigned with values `Smith` and `Afternoon`, respectively.

The output of the preceding code is shown in the following screenshot, which shows the message from the parent element `<say-hello>` inserted in the shadow insertion point:

Hello John !!!

Good Night

Hello Smith !!!

Good Afternoon

Elements | Network Sources Timeline Profiles Resources

```
<!DOCTYPE html>
▼ <html>
  ► <head>...</head>
  ▼ <body>
    ► <say-good mytext="John" time="Night">...</say-good>
    ► <script>...</script>
    ► <say-good>...</say-good>
  </body>
```

html body

# Polymer methods

The PolymerJS library has some utility methods to deal with mixins, imports, and element registration. In the following section, we will learn how to use these methods.

## The Polymer mixin method

Polymer provides mixins for sharing common methods among custom elements. Mixin helps to reduce duplicate code. The syntax for creating a mixin is as follows:

```
Polymer.mixin({ //Common methods });
```

Let's check out an example of sharing common methods using mixins. The details of the example are as follows:

- The common mixin methods are present in the `common-mixin.html` file. The `window.commonMixins` object will contain the shared methods. The code for the `common-mixin.html` file is as follows:

```
<script>
  window.commonMixins = {
    //reusable method returning local name of element
    printMyName: function() {
      return this.localName;
    }
  };
</script>
```

- The `commonMixins` object has a `printMyName` method and is shared to the custom components, which use it.
- A new custom component `<print-hi>` is defined in the `printHi-component.html` file. The `<print-hi>` element has used the `commonMixins` object. The following code shows the definition of the `<print-hi>` component:

```
<polymer-element name="print-hi" constructor="PrintHi">
  <template>
    <h4>Hi!!!</h4>
  </template>
  <script>
    Polymer(Polymer.mixin({
      //local method
      printHi: function() {
```

```

        return "Hi";
    }
}, window.commonMixins));
</script>
</polymer-element>

```

- Another custom element `<print-bye>` has been defined in the `printBye-component.html` file. This component also uses the `commonMixins` object to get access to the `printMyName` method. The `printMyName` method returns the local name of the custom element. The code for the `<print-bye>` element is as follows:

```

<polymer-element name="print-bye" constructor="PrintBye">
  <template>
    <h4>Bye!!!</h4>
  </template>
  <script>
    Polymer(Polymer.mixin({
      printBye:function() {
        return "Bye";
      }
    }, window.commonMixins));
  </script>
</polymer-element>

```

- Now, it is time to test the mixin methods shared by these elements. The test code is present in `mixin-demo.html`. In this code, we have created two objects —one from the `PrintHi` constructor and the other is from the `PrintBye` constructor using the `new` keyword. Then, the `printMyName` name is called from each object, which is logged in the console. The following code contains the test code for checking the mixin methods:

```

<!DOCTYPE html>
<html>
<head>
  <script
src=".../bower_components/webcomponentsjs/webcomponents.m in.js">
</script>
  <link rel="import"
href=".../bower_components/polymer/polymer.html">
  <link rel="import" href="common-mixin.html">
  <link rel="import" href="printHi-component.html">
  <link rel="import" href="printBye-component.html">
  <title>Polymer Mixin Demo</title>
</head>
<body>
<script>

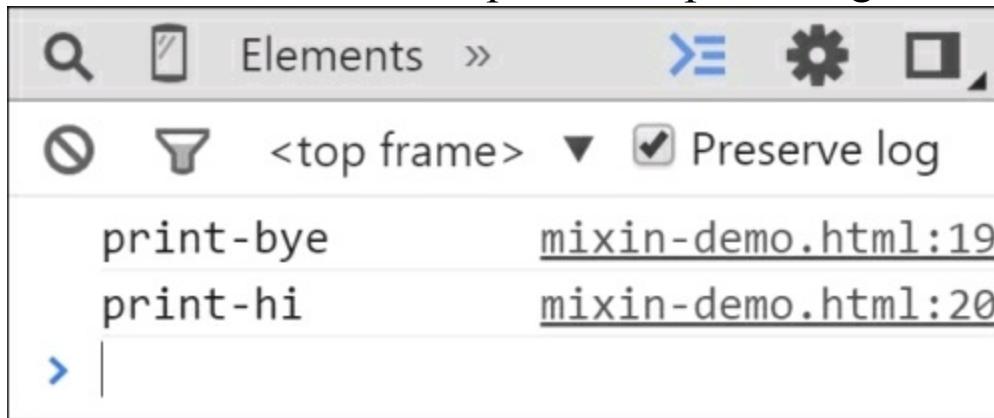
```

```

//Using SayGood Constructor
window.addEventListener('polymer-ready', function(e) {
  var printBye1 = new PrintBye(),
      printHi1 = new PrintHi(),
      myName1 = printBye1.printMyName(),
      myName2 = printHi1.printMyName();
  console.log(myName1);
  console.log(myName2);
});
</script>
</body>
</html>

```

- The complete code can be downloaded from Packt's website. The following screenshot shows the output of the preceding code:



## The Polymer import method

Polymer provides the `Polymer.import` method for importing the external HTML file. Let's check out an example of using the `import` method. The details of the example are as follows:

- The `number-detail.html` file contains a JavaScript file containing a variable `luckyNumber` assigned a value 7. The code present inside the `number-detail.html` file is as follows:

```

<script>
  var luckyNumber = 7;
</script>

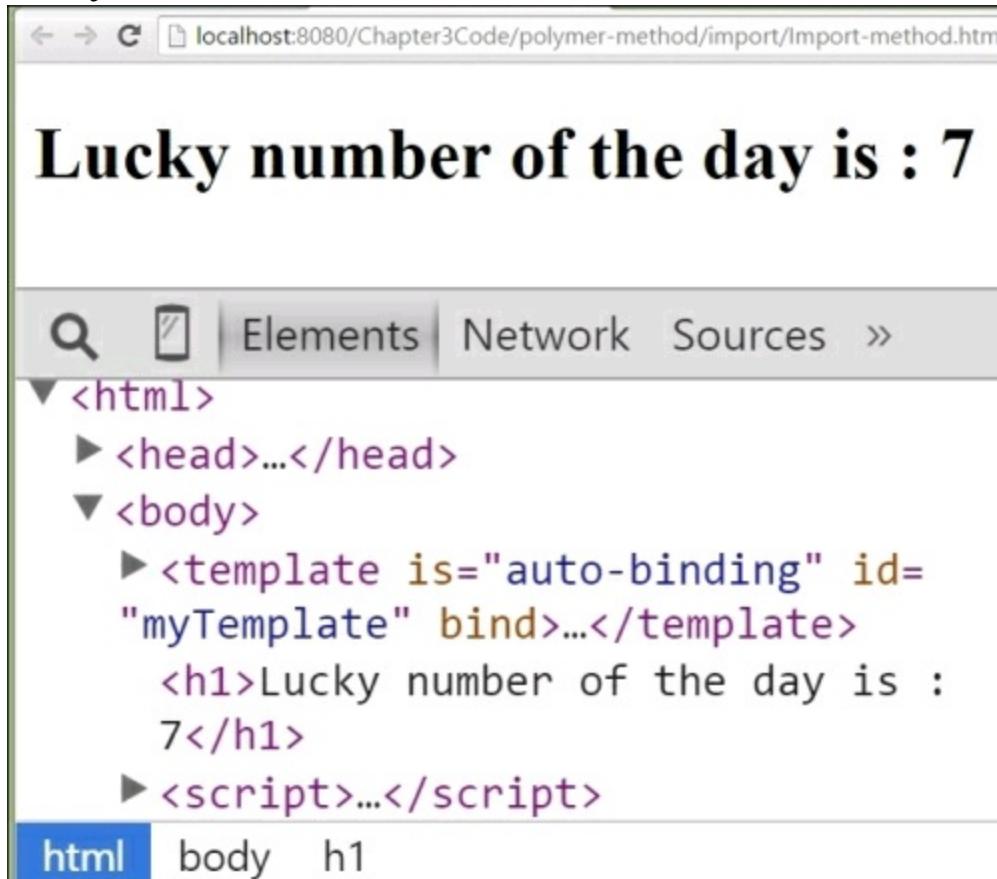
```

- The `number-detail.html` file has been imported in the `import-method.html` file. Once the `number-detail.html` file is imported the lucky number is displayed in the browser as the template is using auto-binding feature. The following code

shows the import of the number-detail.html file:

```
<!DOCTYPE html>
<html>
<head>
    <script
src="../../bower_components/webcomponentsjs/webcomponents.min.js">
</script>
    <link rel="import"
href="../../bower_components/polymer/polymer.html">
    <title>Polymer Import Method Demo</title>
</head>
<body>
    <template is="auto-binding" id="myTemplate">
        <h1>Lucky number of the day is : {{myLuckyNumber}}</h1>
    </template>
</body>
<script>
    Polymer.import(['number-detail.html'], function () {
        var myTemplate = document.querySelector("#myTemplate");
        myTemplate.myLuckyNumber = luckyNumber;
    });
</script>
</html>
```

- The following screenshot shows the output of the preceding code, where the lucky number 7 is rendered in the browser:



# The Polymer waitingFor method

Polymer provides the `waitFor` method to detect the unregistered element in the application. This method returns a list of elements that are not registered yet. Let's check out an example that demonstrates the usage of the `waitFor` method. The details of the example are as follows:

- A custom component `<good-morning>` is created, and intentionally we have delayed the registration for 3 seconds using the `setTimeout` method. Code for the `<good-morning>` element is present in the `morning-component.html` file and is listed as follows:

```
<polymer-element name="good-morning">
  <template>
    <h4>Good Morning</h4>
  </template>
  <script>
    window.setTimeout(function() {
      Polymer("good-morning");
    }, 3000)
  </script>
</polymer-element>
```

- In another HTML file, we have used the `waitFor` method to check whether it contains the `<good-morning>` element inside the array during the initial 3 seconds. The following code shows the use of the `waitFor` method:

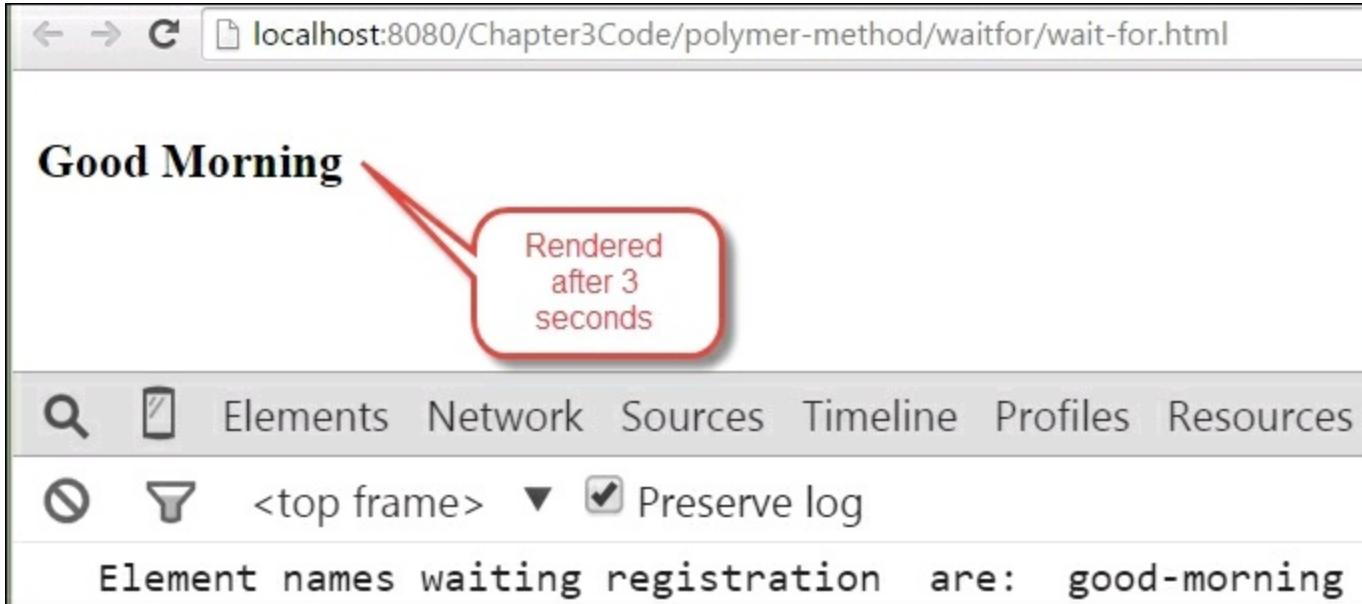
```
<!DOCTYPE html>
<html>
<head>
  <script
    src="../../bower_components/webcomponentsjs/webcomponents.min.js">
  </script>
  <link rel="import"
    href="../../bower_components/polymer/polymer.html">
    <link rel="import" href="morning-component.html">
    <title>Polymer waitFor Method Demo</title>
</head>
<body>
  <good-morning></good-morning>
</body>
</html>
<script>
  var elementList= Polymer.waitFor() ,
```

```

        length = elementList.length;
        for(var i=0; i<length; i++){
            var aElement = elementList[0];
            console.log("Element names waiting registration are:
",aElement.name);
        }
    </script>

```

- The following screenshot shows the output of the preceding code, which logs the unregistered element in a console:



## The Polymer forceReady method

Polymer provides the `forceReady` method to notify Polymer to register all the elements to the DOM. Let's check out an example that demonstrates the usage of the `forceReady` method. The code for the example is present in the `evening-component.html` file and the details are as follows:

- A new custom component `<good-evening>` is created and intentionally delays the registration for 3 seconds. The following code contains the definition of the `<good-evening>` element:

```

<polymer-element name="good-evening">
    <template>
        <h4>Good Evening</h4>
    </template>
    <script>
        window.setTimeout(function() {
            Polymer("good-evening");

```

```
} ,3000)
```

```
</script>
```

```
</polymer-element>
```

- In another HTML file, we have used the `forceReady` method to register all the custom elements. A callback method `Polymer.whenReady` is then called, which tries to register the `<good-evening>` element. The following code contains the use of the `forceReady` method:

```
<!DOCTYPE html>
<html>
<head>
    <script
src="..../bower_components/webcomponentsjs/webcomponents.min.js">
</script>
    <link rel="import"
href="..../bower_components/polymer/polymer.html">
        <link rel="import" href="evening-component.html">
        <title>Polymer forceReady Method Demo</title>
</head>
<body>
    <b><good-evening></good-evening></b>
</body>
</html>
<script>
    Polymer.forceReady();
    Polymer.whenReady(function() {
        console.log("Polymer is ready now...");
```

//Testing registration of good-evening element

```
        Polymer('good-evening');
    })
</script>
```

- As the `<good-evening>` element method is already registered due to use of the `forceReady` method, when we call the registration process again using `Polymer('good-evening')`, it will throw an error saying, it is already registered. The following screenshot shows the output of the preceding code with an error message logged in the console:

localhost:8080/Chapter3Code/polymer-method/forceready/force-ready.html

# Good Evening

Rendered immediately without delay

Elements Network Sources timeline Profiles >

✖  Preset log  
Polymer is ready now...

✖ Uncaught Already registered (Polymer) prototype for element good-evening

# Asynchronous task execution

Polymer provides a method named `async`, which helps with the execution of a code block after a curtailed period of time. The syntax of the `async` method is as follows:

```
async(method, arguments, timeout);
```

Let's check out an example demonstrating the use of `async` method. The details of the example are as follows:

- A new custom element `<my-text>` is created with the `message` property initialized to `Hello` in a `ready` block. An `async` method is used which will be executed after 2 seconds and another string is appended to the `message` property. The code for this example is as follows:

```
<!DOCTYPE html>
<html>
<head>
    <script
src="../bower_components/webcomponentsjs/webcomponents.min.js">
</script>
    <link rel="import"
href="../bower_components/polymer/polymer.html">
    <title>Polymer async method demo</title>
    <polymer-element name="my-text">
        <template>
            {message}
        </template>
        <script>
            Polymer({
                ready: function() {
                    this.message= "Hello"
                    this.async(function() {
                        this.message += "---->(after 2 second)--->Bye";}, null, 2000);
                }
            });
        </script>
    </polymer-element>
</head>
<body>
    <my-text>
    </my-text>
</body>
</html>
```

- The output of the preceding code is shown in the following screenshot, with two strings appended after 2 seconds of page rendering:

A screenshot of a browser's developer tools element inspector. The URL in the address bar is "localhost:8080/Chapter3Code/async-task/async-method.html". The main content area displays the text "Hello--->(after 2 second)--->Bye". A red callout bubble points from the word "Bye" to a red box around the text "(after 2 second)". Below the content, the DOM tree is visible:

```
> <polymer-element name="my-text">...</polymer-element>
  <my-text>
    <#shadow-root>
      "Hello--->(after 2 second)--->Bye"
    </my-text>
  </body>
```

The "html" tab is selected at the bottom left.

## Note

To find out more about the `async` method, refer to <https://www.polymer-project.org/docs/polymer/polymer.html#asyncmethod>.

# Developing a digital clock

In the previous section, we have learned about many concepts of Polymer. Now, it is time to develop a custom component. We will develop a digital clock that we have already created in [Chapter 1, Introducing Web Components](#). In this section, we will develop the same digital clock using Polymer. The code definition of a digital clock is present in the `clock-component.html` file and is as follows:

```
<polymer-element name="ts-clock">
  <template bind="{{clock}}>
    <style>
      :host .clock{
        display: inline-flex;
        justify-content: space-around;
        background: floralwhite;
        font-size: 2rem;
        font-family: serif;
      }
      :host .clock .hour, :host .clock .minute, :host .clock
.second{
        color: tomato;
        padding: 1.5rem;
        text-shadow: 0px 1px grey;
      }
    </style>
    <div class="clock">
      <div class="hour">{{ clock.hour }}</div>
      <div class="minute">{{ clock.minute }}</div>
      <div class="second">{{ clock.second }}</div>
    </div>
  </template>
  <script>
    Polymer({
      ready:function(){
        this.updateClock()
      },
      updateClock: function(){
        var date = new Date();
        this.clock ={
          hour : date.getHours(),
          minute : date.getMinutes(),
          second : date.getSeconds()
        };
        this.async(this.updateClock, null, 1000);
      }
    })
  </script>
</polymer-element>
```

```
});  
</script>  
</polymer-element>
```

The details of the preceding code are as follows:

- The name of the digital clock is `<ts-clock>`.
- The HTML markup is the same as that we used in the [Chapter 1, Introducing Web Components](#), containing three `div` elements as a place holder for hours (HH), minutes (MM), and seconds (SS).
- The template is bound to the `clock` object using the `bind` attribute.
- The definition of the digital clock contains an `updateClock` method, which creates the `clock` object containing three properties—hour, minute, and second—which get the value from a `new Date()` object.
- The `updateClock` method is called recursively and asynchronously using the `this.async` method after 1 second. It means that every 1 second, the `clock` object gets updated with new values, which, in turn, updates the template.

The following code shows the use of the `<ts-clock>` element in another HTML file:

```
<!DOCTYPE html>  
<html>  
<head>  
  <script  
src="../bower_components/webcomponentsjs/webcomponents.min.js"> </script>  
  <link rel="import" href="../bower_components/polymer/polymer.html">  
  <b><link rel="import" href="clock-component.html"></b>  
  <title>Polymer clock component demo</title>  
</head>  
<body>  
  <ts-clock>  
  </ts-clock>  
</body>  
</html>
```

The following screenshot shows the output of the preceding code with the current time in hours, minutes, and seconds format:

2 46 1

Elements | Network Sources Timeline

```
▼<ts-clock>
  ▼#shadow-root
    ▶<style>...</style>
    ▼<div class="clock">
      <div class="hour">2</div>
      <div class="minute">46</div>
      <div class="second">1</div>
    </div>
  </ts-clock>
```

html head

# Working with Yeoman

In [Chapter 2](#), *Introducing Polymer*, we installed Yeoman and explored some of the usage of Polymer. In this section, we will find more of them to work with Polymer. The `generator-polymer` module can be installed using the following command:

```
npm install -g generator-polymer
```

Considering that we have already installed Yeoman and `generator-polymer`, let's learn about the element, seed, and GitHub page generator.

## Yeoman element generator

Yeoman provides an element generator to create a skeleton of a custom element. The command for generating a skeleton of a custom element is as follows:

```
yo polymer: element my-element
```

Or

```
yo polymer: el my-element
```

The details of the preceding command are as follows:

- The name of the custom element is `my-element`. We can provide our own name, which must be separated with a hyphen(-)
- The directory structure created by this command is  
`app/elements/elements.html`

Let's create a custom element `<say-time>` using Yeoman element generator. The command for generating a skeleton of the `<say-time>` element is as follows:

```
yo polymer:element say-time
```

The following screenshot shows the terminal with Yeoman element generator in effect:

```
J:\Chapter3Code\yeo-demo>yo polymer:element say-time
? Would you like an external CSS file for this element? No
? Would you like to include an import in your elements.html file? (y/N) n
? Would you like to include an import in your elements.html file? No
  create app\elements\say-time\say-time.html
```

```
J:\Chapter3Code\yeo-demo>
```

On successful execution of the preceding command, it creates a directory structure app/element/say-time.html. The skeleton of the generated custom element for the <say-time> element is as follows:

```
<link rel="import" href="../../bower_components/polymer/polymer.html">
<polymer-element name="say-time" attributes="">
  <template>
    <style>
      :host {
        display: block;
      }
    </style>
  </template>
  <script>
    (function () {
      Polymer({
        // define element prototype here
      });
    })();
  </script>
</polymer-element>
```

There are few changes we need to make to adjust the library path, and some code needed for our chapter demonstration. We have overridden the created callback method by assigning the currentTime attribute with a new Date object. The modified code of the <say-time> component is as follows:

```
<link rel="import"
href="/Chapter3Code/bower_components/polymer/polymer.html">
<polymer-element name="say-time">
  <template>
    <style>
```

```

:host {
  display: block;
}
</style>
{{currentTime}}
</template>
<script>
(function () {
  Polymer({
    created: function() {
      this.currentTime = new Date();
    }
  });
})();
</script>
</polymer-element>

```

The following code shows the use of the `<say-time>` custom element in a current page:

```

<!DOCTYPE html>
<html>
<head>
  <script
src="/Chapter3Code/bower_components/webcomponentsjs/webcomponents.min.js">
</script>
<link rel="import" href="say-time.html">
  <title>Polymer sayTime component demo</title>
</head>
<body>
  <say-time>
  </say-time>
</body>
</html>

```

The output of the preceding code is rendered as the following screenshot, showing the current time due to the expression `{{currentTime}}` in the template:

Sat Jan 17 2015 20:55:03 GMT+0530 (India Standard Time)

The screenshot shows the Chrome DevTools Elements tab. At the top, there are icons for search, copy, and refresh, followed by tabs for Elements, Network, Sources, Timeline, Profiles, Resources, Audits, and a more options menu. The Elements tab is active. Below the tabs, the DOM tree is displayed. A `<say-time>` element is selected, indicated by a purple triangle icon. Inside it, a `#shadow-root` is shown, also with a purple triangle icon. The shadow root contains a `<style>` element with a host selector (`:host { display: block; }`) and a text node containing the date and time (`Sat Jan 17 2015 20:55:03 GMT+0530 (India Standard Time)`). The entire `<say-time>` element is enclosed in a pair of double quotes. At the bottom of the DevTools interface, there is a navigation bar with tabs for html, body, and say-time, where the html tab is currently selected.

## Yeoman seed generator

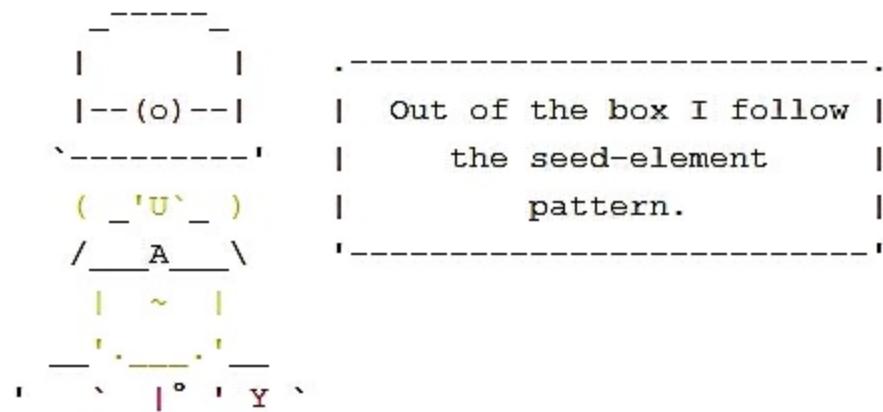
Yeoman Seed generator is used for developing a reusable element. It creates all the necessary boilerplate directories and files needed to publish a new custom element. The command for running a seed generator is as follows:

```
yo polymer: seed tag-name
```

The following screenshot shows the terminal with a seed generator in effect:



```
J:\Chapter3Code\yeo-seed>yo polymer:seed display-year
```



? What is your GitHub username? saan1984

```
create .gitignore
create .gitattributes
create .bowerrc
```

Once the preceding command is executed successfully, a directory structure is created for developing a reusable component, as shown in the following screenshot:



After the directory structure is generated, we can modify the definition of the `<display-year>` custom element in the `display-year.html` file. We have overridden the `created` callback method with the `currentYear` property. The modified code is as follows:

```
ready: function() {
  this.currentYear = new Date().getFullYear();
}
```

The modified template for the `<display-year>` element is as follows, with the

added expression {{currentYear}}:

```
<template>
  <h1>{{currentYear}}</h1>
</template>
```

You can find the complete source code containing all the directory structures with dependent files from Packt's website.

## Yeoman GitHub page generator

This generator is used for creating a GitHub page for the custom element. The syntax of the command for the GitHub page generator is as follows:

```
cd components/tag-name
yo polymer:gh
```

The following screenshot shows the terminal with the GitHub page generator in execution for the <display-year> custom element:

The screenshot shows a terminal window titled "Terminal". The command entered is "J:\Chapter3Code\yeo-seed\display-year>yo polymer:gh". The output includes a colorful ASCII art logo for Yeoman, followed by prompts asking for the GitHub username ("What is your GitHub username? saan1984") and the element's name ("What is your element's name: display-year"). The final message is "Cloning into 'display-year'...".

```
J:\Chapter3Code\yeo-seed\display-year>yo polymer:gh

 _-----_
|         |     .-----.
| --(o)--|     |     'Allo! Looking to    |
`-----'     |     generate a Github Page   |
 ( _ 'U` _ )  |     are yah?      |
 /__A__\     '-----'
 | ~ |
 | . . |
 | ° ' Y |

? What is your GitHub username? saan1984
? What is your element's name: display-year
Cloning into 'display-year'...
```

## Note

To find out more about the GitHub page generator, refer to

<https://github.com/yeoman/generator-polymer#gh>.

# Preparing for production using vulcanize

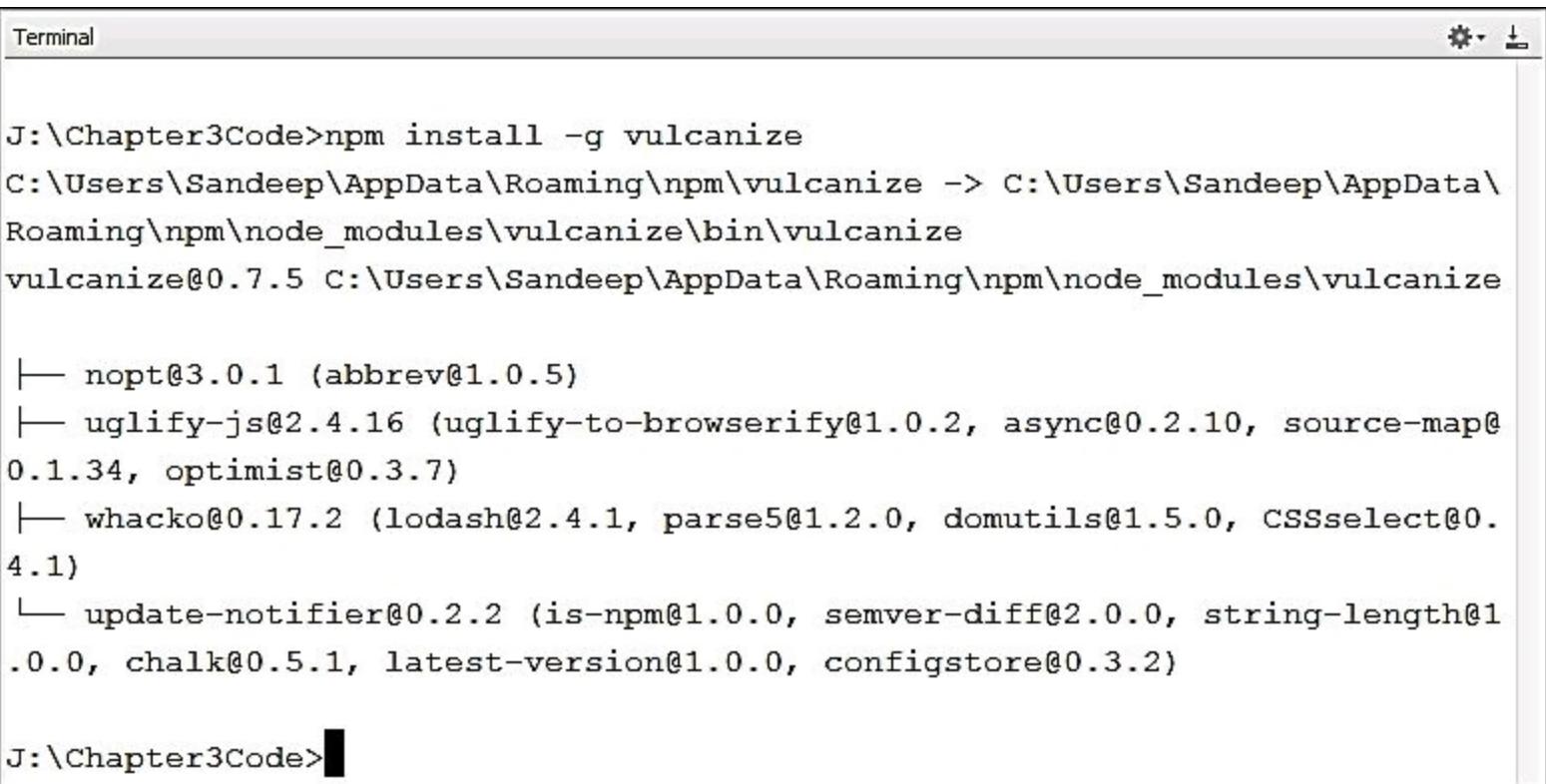
In Polymer-based web application development, we may find a situation where we need to use a lot of HTML Import for a web page. Each HTML Import is an HTTP call to the server. This can really hamper the application performance, and definitely the application is not a good candidate for production deployment. This problem can be solved by a vulcanize process that makes the Polymer application a more durable and production-ready application.

## Vulcanize installation

**Node package manager (npm)** is required before installing the **vulcanize** package. Considering npm is present in the system, use the following command to start the installation of vulcanize:

```
npm install -g vulcanize
```

The preceding command will install vulcanize globally. The following screenshot shows the terminal with the vulcanize installation process:



The screenshot shows a terminal window with the title "Terminal". The command "npm install -g vulcanize" is entered and executed. The output shows the installation of vulcanize@0.7.5 and its dependencies, including nopt, uglify-js, whacko, and update-notifier. The terminal prompt "J:\Chapter3Code>" is visible at the bottom.

```
J:\Chapter3Code>npm install -g vulcanize
C:\Users\Sandeep\AppData\Roaming\npm\vulcanize -> C:\Users\Sandeep\AppData\Roaming\npm\node_modules\vulcanize\bin\vulcanize
vulcanize@0.7.5 C:\Users\Sandeep\AppData\Roaming\npm\node_modules\vulcanize
  └── nopt@3.0.1 (abbrev@1.0.5)
    ├── uglify-js@2.4.16 (uglify-to-browserify@1.0.2, async@0.2.10, source-map@0.1.34, optimist@0.3.7)
    ├── whacko@0.17.2 (lodash@2.4.1, parse5@1.2.0, domutils@1.5.0, cssselect@0.4.1)
    └── update-notifier@0.2.2 (is-npm@1.0.0, semver-diff@2.0.0, string-length@1.0.0, chalk@0.5.1, latest-version@1.0.0, configstore@0.3.2)

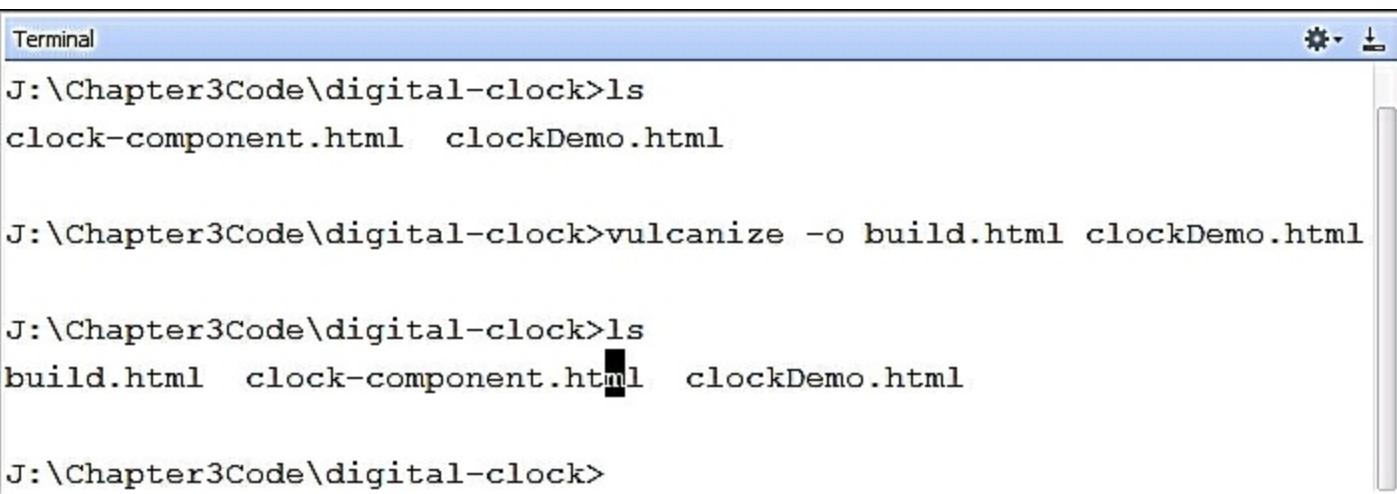
J:\Chapter3Code>
```

# Running vulcanize process

Once vulcanize is installed in the system, we can run the process to optimize the files, to make them production ready. The syntax of the vulcanize command is as follows:

```
Vulcanize -o targetFile.html sourceFile.html
```

The vulcanize process concatenates the files into a single file for deployment. The following screenshot shows the vulcanize process in progress for the `clockDemo.html` file:



A screenshot of a terminal window titled "Terminal". The window shows the following command-line session:

```
J:\Chapter3Code\digital-clock>ls
clock-component.html  clockDemo.html

J:\Chapter3Code\digital-clock>vulcanize -o build.html clockDemo.html

J:\Chapter3Code\digital-clock>ls
build.html  clock-component.html  clockDemo.html

J:\Chapter3Code\digital-clock>
```

On completion of the preceding command, the `build.html` file is generated, which concatenates the dependent files. Now, the `build.html` file is optimized and ready for production. Vulcanize is also available as Grunt and Gulp tasks as well. Take a look at <https://www.npmjs.com/package/grunt-vulcanize> for the `grunt-vulcanize` package and <https://www.npmjs.com/package/gulp-vulcanize> for the `gulp-vulcanize` package.

## Note

To find out more about the vulcanize process, refer to <https://github.com/Polymer/vulcanize>.

# Summary

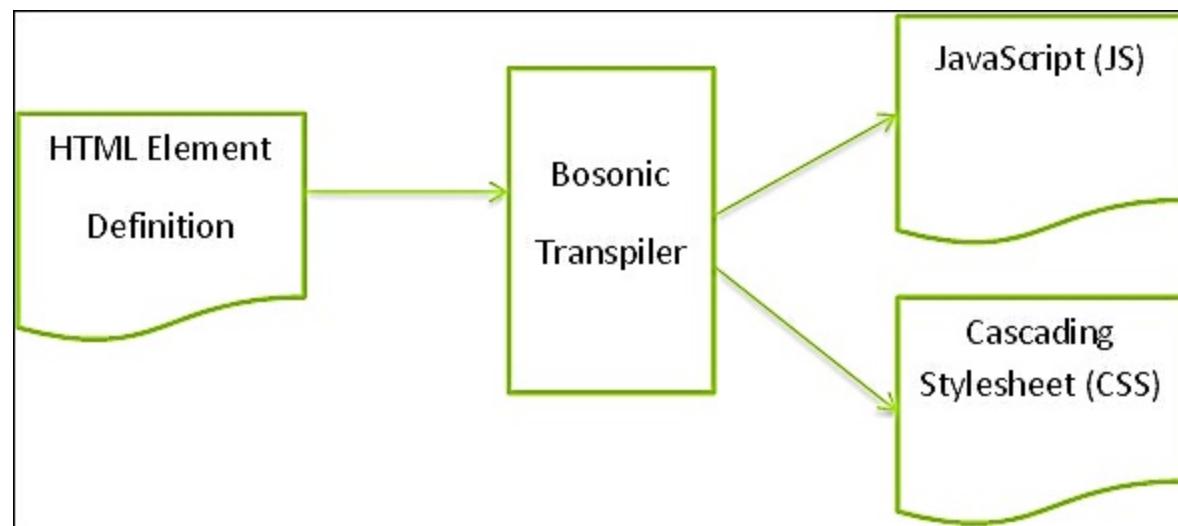
In this chapter, you learned about the key concepts of the Polymer library such as expression, filter expression, and element lifecycle. You have also explored custom element development using the Polymer library, followed by a sample digital clock development. In the next chapter, you will learn about Bosonic framework.

# Chapter 4. Exploring Bosonic Tools for Web Component Development

Bosonic provides a set of tools that help developers build a web component development. In this chapter, we will learn about configuration, lifecycle, built-in elements, and custom element development using the Bosonic tool.

## What is Bosonic?

Bosonic is a library for web component development based on the W3C specification. The Bosonic library came about so as to support not-so-modern browsers, such as IE9. Bosonic is built on top of the PolymerJS polyfill and its own polyfill. Bosonic acts like a transpiler. A **transpiler** is a simple compiler, which takes a source code as input and output in another programming language. The following diagram shows the Bosonic acting as a transpiler:



The Bosonic transpiler takes the element definition in the HTML format and generates the respective JavaScript and CSS file.

# Browser support

As mentioned in the first chapter, web component specification is very new and is not implemented by all browsers. However, Bosonic tries to support more recent browsers by using polyfill. As of today, Bosonic supports the following listed browsers:

- Mozilla Firefox 25+
- Google Chrome 30+
- Internet Explorer 9+
- Safari 6+
- Opera 12+

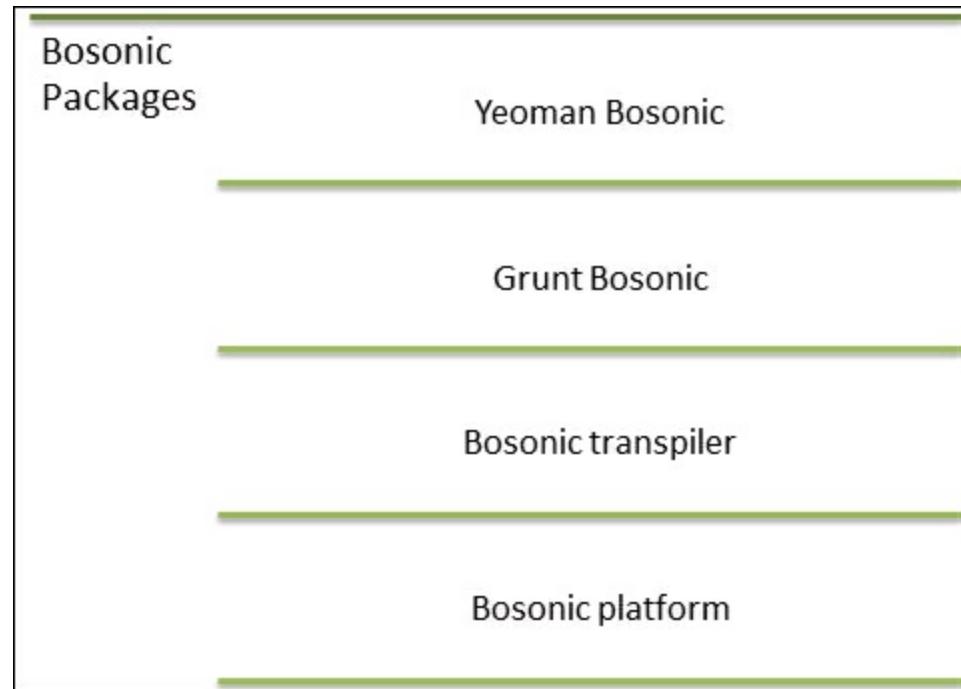
# Configuring Bosonic

Bosonic library is hosted by GitHub. The complete library can be downloaded from the following link:

<https://github.com/bosonic/bosonic>

# Bosonic packages

The Bosonic elements are made up of four packages. We can consider these packages as layers of a technology stack that supports Bosonic elements. The following diagram shows the Bosonic packages:



The details of the mentioned packages are listed as follows:

- **Bosonic platform:** This package contains all the polyfill JavaScript files to provide the missing features from web component specification
- **Grunt Bosonic:** This is a Grunt task that uses the transpiler to build your elements
- **Bosonic transpiler:** This is an **node package manager (npm)** module for converting the Bosonic element definition to CSS and JS
- **Yeoman Bosonic:** This is a Yeoman generator that creates the skeleton of a new element

# Built-in elements

The Bosonic library comes with a set of built-in components that can be used by a developer for web application development. The complete list of built-in elements can be found by visiting the following URL:

<http://bosonic.github.io/elements.html>

We will explore a few of them in this chapter.

## The b-sortable element

The `b-sortable` element can be used to group and sort the DOM elements. The `<b-sortable>` element can be installed using the following command:

```
npm install --save b-sortable
```

The `--save` switch is used to include the `b-sortable` element code in the local project. After the successful execution of the preceding command, it generates a directory structure.

The following screenshot shows the directory and file structure generated by the preceding command:

```
+  
x J:\Chapter4Code>cd builtin-element  
  
J:\Chapter4Code\builtin-element>npm install --save b-sortable  
b-sortable@0.2.0 node_modules\b-sortable  
└── bosonic@0.4.0  
  
J:\Chapter4Code\builtin-element>ls  
node_modules  
  
J:\Chapter4Code\builtin-element>cd node_modules  
  
J:\Chapter4Code\builtin-element\node_modules>cd b-sortable  
  
J:\Chapter4Code\builtin-element\node_modules\b-sortable>ls  
AUTHORS      README.md  karma.conf.js  package.json  
Gruntfile.js  demo       lib           src  
LICENSE       dist       node_modules  test  
  
J:\Chapter4Code\builtin-element\node_modules\b-sortable>
```

From the preceding screenshot, we can see that the parent directory named `node_module` is created with a subdirectory named `b-sortable`, which contains the real JavaScript code definition for the element.

The dependent files for the `<b-sortable>` element can be installed using the following command:

```
npm install
```

The following screenshot shows the terminal with the `b-sortable` element by installing dependent files using the preceding command:

```
+  
✖ J:\Chapter4Code\builtin-element\node_modules\b-sortable>npm install  
  npm WARN optional dep failed, continuing fsevents@0.3.4  
/  
  
> phantomjs@1.9.13 install J:\Chapter4Code\builtin-element\node_modu  
les\b-sortable\node_modules\karma-phantomjs-launcher\node_modules\ph  
antomjs  
> node install.js  
  
Download already available at C:\Users\Sandeep\AppData\Local\Temp\ph  
antomjs\phantomjs-1.9.8-windows.zip  
Extracting zip contents
```

Once the entire dependent files are generated using npm, we can start the Grunt task for transpiling the `b-sortable.html` file. To get a copy of the `<b-sortable>` element in the `demo` directory, we use the following command:

```
grunt demo
```

The following screenshot shows the Grunt task execution, which generates the corresponding CSS and JS files in the `demo` directory:

```
+ J:\Chapter4Code\builtin-element\node_modules\b-sortable>grunt demo
✖ Running "clean:dist" (clean) task
Cleaning dist...OK

Running "bosonic:components" (bosonic) task

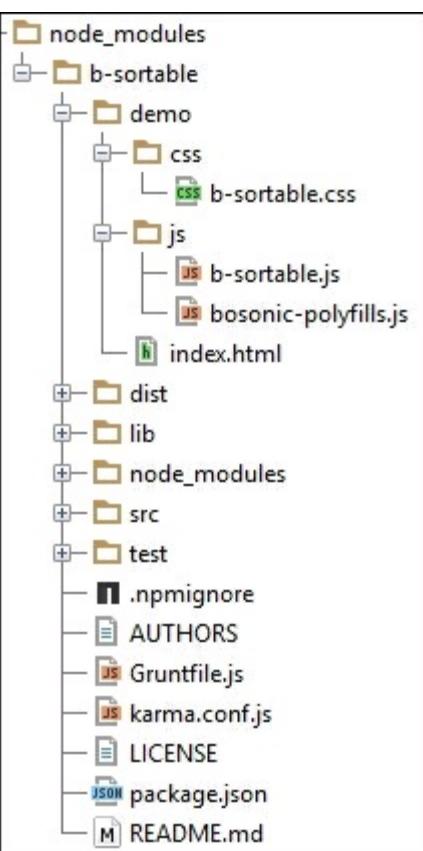
Running "copy:lib" (copy) task
Copied 1 files

Running "copy:dist" (copy) task
Copied 2 files

Running "connect:demo" (connect) task
Started connect web server on http://0.0.0.0:8020

Running "watch" task
Waiting...
```

Once the Grunt task is successfully executed, the corresponding CSS and JS files are generated inside the `demo` directory. The following screenshot shows the updated directory structure:



Now, we can view the demo of the `<b-sortable>` element by running the `index.html` file present inside the `demo` subdirectory:

The screenshot shows a browser window with the URL `localhost:8080/Chapter4Code/builtin-element/node_modules/b-sortable/demo/index.html`. Below the address bar is a list of six items: "Item 1", "Item 2", "Item 3", "Item 4", "Item 5", and "Item 6", each enclosed in a light gray rectangular box. The browser's developer tools are open, specifically the Elements tab. In the DOM tree, under the `<b-sortable>` element, there is a `<ul>` element containing six `<li>` elements, each with the attribute `draggable="true"` and the class `class`. The items are listed as "Item 1", "Item 2", "Item 3", "Item 4", "Item 5", and "Item 6". The developer tools also show the `html`, `body`, and `script` tabs at the bottom.

## The `b-toggle-button` element

The `b-toggle-button` element has two different states, *on* and *off*, representing the checked and unchecked element. The `<b-toggle-button>` element can be installed using the following command:

```
npm install --save b-toggle-button
```

After the successful execution of the preceding command, it generates a directory structure. The following screenshot shows the directory and file structure generated by the preceding command:

## Terminal



```
+ J:\Chapter4Code\builtin-element>npm install --save b-toggle-button
X b-toggle-button@0.1.0 node_modules\b-toggle-button
  └── bosonic@0.4.0

J:\Chapter4Code\builtin-element>cd node_modules

J:\Chapter4Code\builtin-element\node_modules>ls
b-sortable  b-toggle-button

J:\Chapter4Code\builtin-element\node_modules>cd b-toggle-button

J:\Chapter4Code\builtin-element\node_modules\b-toggle-button>ls
AUTHORS      LICENSE      demo  karma.conf.js  package.json  test
Gruntfile.js  README.md   dist   node_modules   src

J:\Chapter4Code\builtin-element\node_modules\b-toggle-button>
```

The dependent files for the `<b-toggle-button>` element can be installed using the following command:

```
npm install
```

The following screenshot shows the terminal with the `b-toggle-button` element by installing dependent files using preceding command:

## Terminal



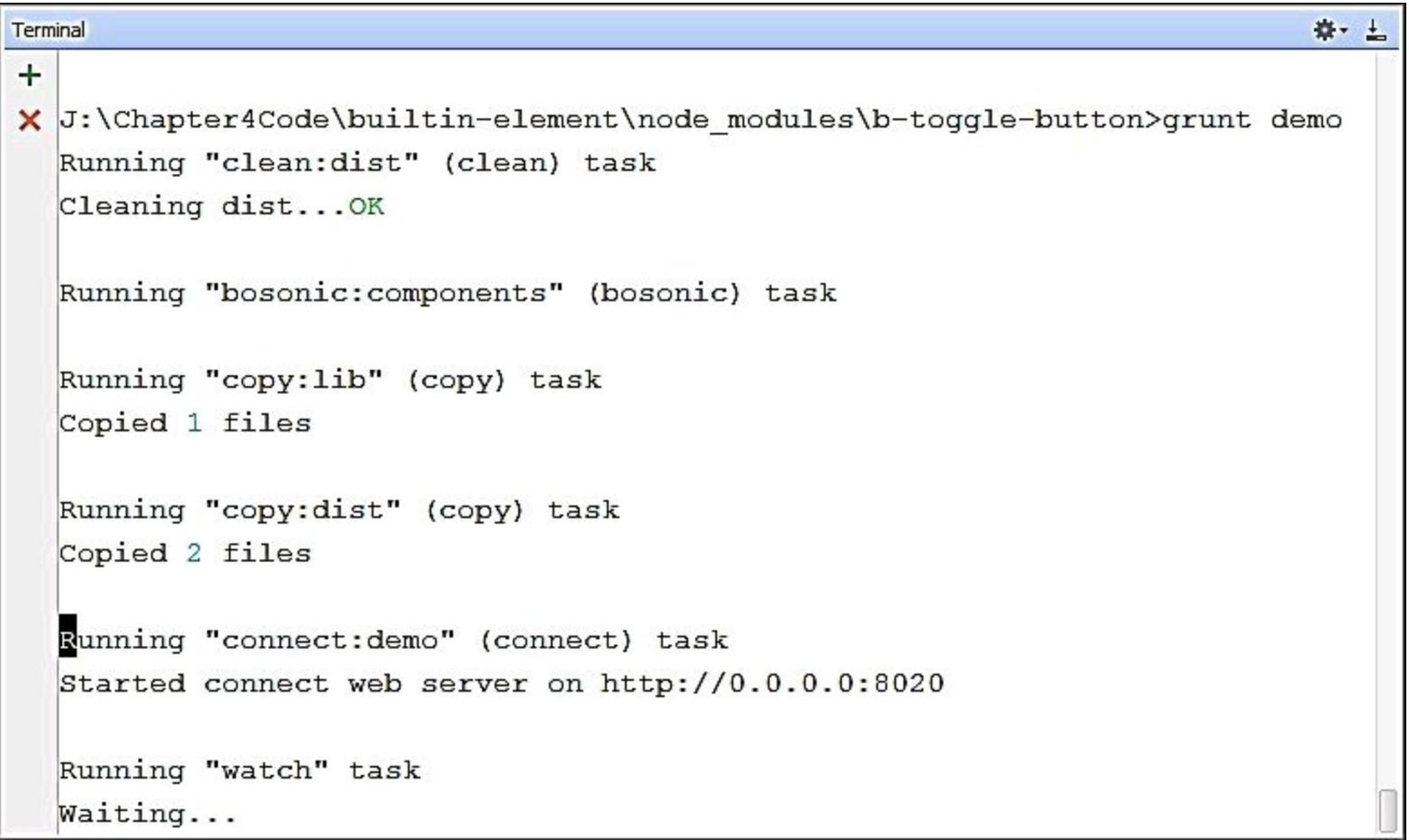
```
+ J:\Chapter4Code\builtin-element\node_modules\b-toggle-button>npm install
X /
  > phantomjs@1.9.16 install J:\Chapter4Code\builtin-element\node_modules\b-
  -toggle-button\node_modules\karma-phantomjs-launcher\node_modules\phantom
  js
```

Once the entire dependent files are generated using npm, we can start the Grunt task for transpiling the `b-toggle-button.html` file. To get a copy of the `<b-toggle-`

button> element in the `demo` directory, we use the following command:

```
grunt demo
```

The following screenshot shows the Grunt task execution, which generates the corresponding CSS and JS files in the `demo` directory:



A screenshot of a terminal window titled "Terminal". The window shows the output of a Grunt task named "demo". The tasks listed are "clean:dist", "bosonic:components", "copy:lib", "copy:dist", "connect:demo", and "watch". The "connect" task has a note indicating it started a web server on port 8020. The "watch" task is currently running, indicated by the message "Waiting...".

```
+ J:\Chapter4Code\builtin-element\node_modules\b-toggle-button>grunt demo
Running "clean:dist" (clean) task
Cleaning dist...OK

Running "bosonic:components" (bosonic) task

Running "copy:lib" (copy) task
Copied 1 files

Running "copy:dist" (copy) task
Copied 2 files

Running "connect:demo" (connect) task
Started connect web server on http://0.0.0.0:8020

Running "watch" task
Waiting...
```

Now, we can view the demo of the `<b-toggle-button>` element by running the `index.html` file present inside the `demo` subdirectory. Here is the screenshot for this:

# with caption

OFF

Elements Network Sources Timeline Profile

```
▼ <b-toggle-button id="toggle" role="checkbox" tabindex="0" aria-checked="false">
  ► <style>...</style>
  ▼ <div class="b-toggle-button">
    <span class="b-toggle-on">ON</span>
    <span class="b-toggle-thumb"></span>
    <span class="b-toggle-off">OFF</span>
  </div>
</b-toggle-button>
```

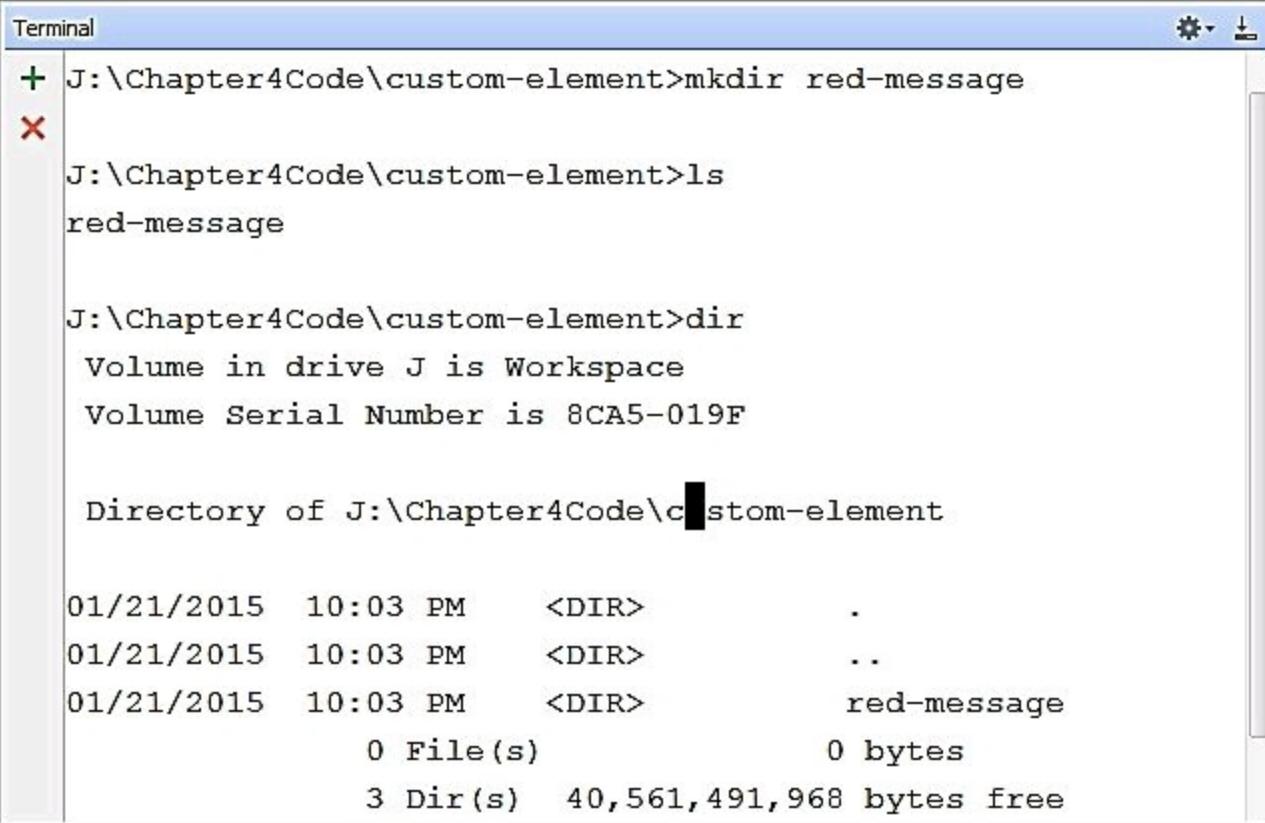
html body

# Developing custom component

The Bosonic framework supports the development of a custom element. In this section, we will learn step-by-step details for developing a custom element. We will build a custom element named `<red-message>`. The detailed steps are as follows:

## Step 1 – creating the red-message element directory

Create a directory named `red-message` using your terminal and the `mkdir` command. The following screenshot shows the terminal with a directory name `red-message` creation. We can verify the directory creation using the `dir` or `ls` command in the terminal, depending on the operating system used by the developer.



```
Terminal
+ J:\Chapter4Code\custom-element>mkdir red-message
X

J:\Chapter4Code\custom-element>ls
red-message

J:\Chapter4Code\custom-element>dir
Volume in drive J is Workspace
Volume Serial Number is 8CA5-019F

Directory of J:\Chapter4Code\custom-element

01/21/2015  10:03 PM    <DIR>        .
01/21/2015  10:03 PM    <DIR>        ..
01/21/2015  10:03 PM    <DIR>        red-message
                           0 File(s)          0 bytes
                           3 Dir(s)  40,561,491,968 bytes free
```

## Step 2 – changing the current directory to red-message

Change the current directory to `red-message` using the `cd` command in the terminal. The following screenshot shows the terminal with the `cd` command in execution:

A screenshot of a terminal window titled "Terminal". The window has a toolbar at the top with icons for settings and download. The main area shows a command line interface. A green plus sign icon is in the top-left corner of the main area. The command "J:\Chapter4Code\custom-element>cd red-message" is entered, followed by a line separator "J:\Chapter4Code\custom-element\red-message>" and a black cursor block.

## Step 3 – generating the skeleton for <red-message>

We need to generate the skeleton of the Bosonic custom element using Yeoman Bosonic generator. The package name for Bosonic generator is `generator-bosonic`, and it can be installed using the `npm install --save generator-bosonic` command. The skeleton of the `<red-message>` element can be generated using the `yo bosonic` command in the terminal. Refer to the following screenshot:

Terminal

+ J:\Chapter4Code\custom-element\red-message>yo bosonic

X

-----  
| |  
|--(o)--| .-----.  
'-----' | Welcome to Yeoman,  
( \_ 'U`\_ ) | ladies and gentlemen!  
/\_\_\_A\_\_\_\ |  
| ~ |  
' . . ' .  
, ` |° , Y ,  
'

You're using the Bosonic generator. You will now generate a boilerplate for a new Web Component.

? What do you want to call your new element? (red-message)

? What do you want to call your new element? red-message

create package.json  
create Gruntfile.js  
create karma.conf.js  
create test\red-message.js  
create src\red-message.html  
create demo\index.html  
create .gitignore

npm WARN package.json red-message@0.0.0 No repository field

.

## Step 4 – verifying the directory structure

Verify the directory structure generated by Yeoman Bosonic generator using the `ls` command. The following screenshot shows the directory structure generated by the generator:

```
+ J:\Chapter4Code\custom-element\red-message>ls
✗ Gruntfile.js  demo  karma.conf.js  node_modules  package.json  src  test

J:\Chapter4Code\custom-element\red-message>cd demo

J:\Chapter4Code\custom-element\red-message\demo>ls
index.html

J:\Chapter4Code\custom-element\red-message\demo>cd ../src

J:\Chapter4Code\custom-element\red-message\src>ls
red-message.html

J:\Chapter4Code\custom-element\red-message\src>cd ../test

J:\Chapter4Code\custom-element\red-message\test>ls
red-message.js
```

## Step 5 – defining code for the <red-message> element

Now, it's time to write the definition for the `<red-message>` element in the `red-message.html` file present in the `src` directory. The modified content of the `red-message.html` file is listed as follows:

```
<element name="red-message">
  <style>
    h1{
      color: red;
    }
  </style>
  <template>
    <h1>Welcome to Bosonic framework.</h1>
  </template>
  <script>
    ({
      createdCallback: function() {
        this.appendChild(this.template.content.cloneNode(true));
      }
    });
  
```

```
</script>
</element>
```

The details of the preceding code are listed as follows:

- The `<template>` element contains an `h1` element with a message that is going to be rendered
- The `<style>` element contains the style for the `color: red` attribute for the `h1` element
- The `<script>` element overrides the `createdCallback` method, which clones the template content and appends it to the `<red-message>` element

## Step 6 – modifying the index.html demo file

To test the `red-message` element, we need to include the `red-message` tag in the `index.html` file. The following code shows the content of the `index.html` file:

```
<!DOCTYPE html>
<html>
<head>
    <title>Demo</title>
    <meta charset="utf-8">
    <script src="js/bosonic-polyfills.js"></script>
    <script src="js/red-message.js"></script>
    <link href="css/red-message.css" rel="stylesheet">
</head>
<body>
    <red-message> </red-message>
</body>
</html>
```

As shown in the preceding code, `red-message.js` and `red-message.css` are the files that are generated by the transpiler. These files will be used in the next step.

## Step 7 – generating distribution files using Grunt

In this step, we need to call the transpiler which will generate the CSS and JS file from the `red-message.html` file. A Grunt task watcher can watch the changes in the `red-message.html` file and generate the `red-message.css` and `red-message.js` files. A Grunt task watcher can be executed using the `grunt demo` command. The following screenshot shows the terminal with Grunt watcher in execution:

Terminal

```
+ J:\Chapter4Code\custom-element\red-message>grunt demo
✖ Running "clean:dist" (clean) task
Cleaning dist...OK

Running "bosonic:components" (bosonic) task

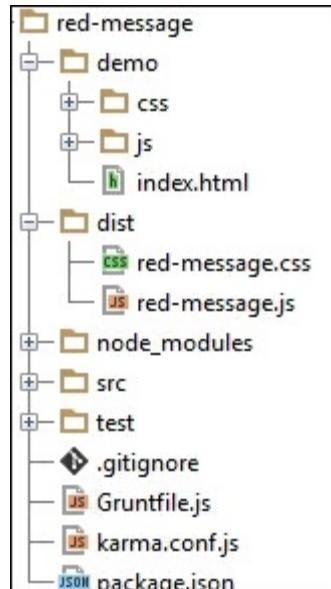
Running "copy:lib" (copy) task
Copied 1 files

Running "copy:dist" (copy) task
Copied 2 files

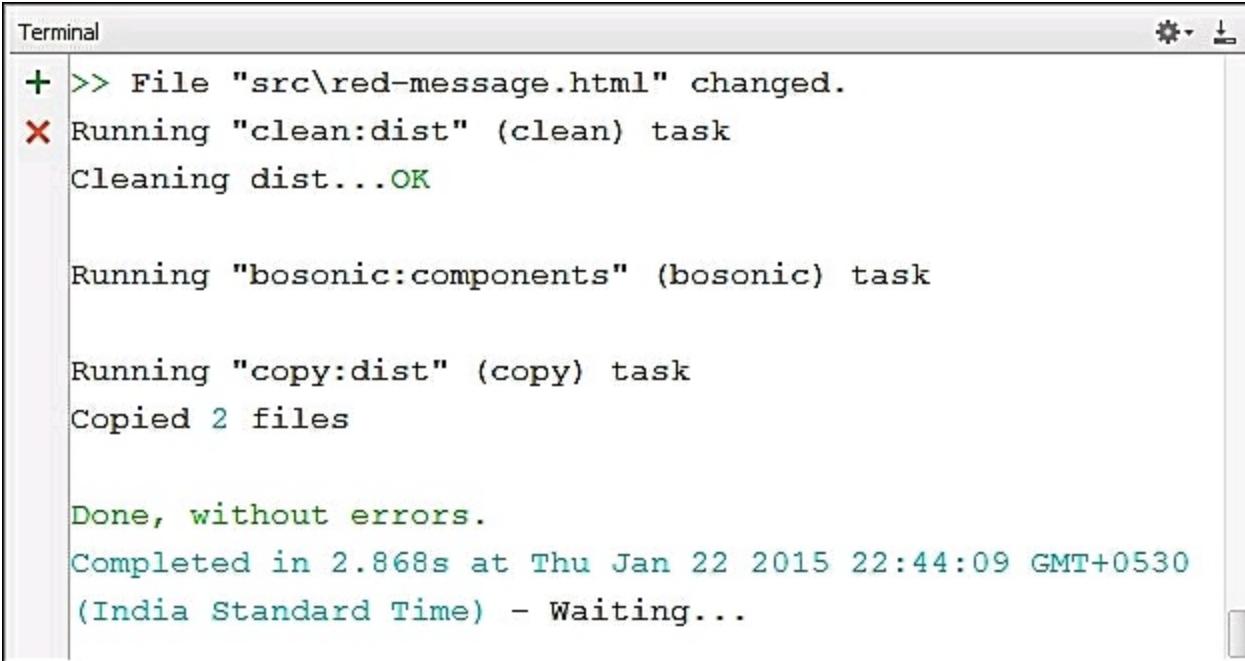
Running "connect:demo" (connect) task
Started connect web server on http://0.0.0.0:8020

Running "watch" task
Waiting...
```

After successful execution of the Grunt compiler, it generates the `red-message.js` and `red-message.css` files in the `dist` directory and copies a set of these files to the `demo` directory. The following screenshot shows the generated files after compilation:



Any changes to the `red-message.html` file will be picked up by the Grunt task watcher and also made to the corresponding CSS and JS files. The following screenshot displays the Grunt task watcher logs in the terminal:



The terminal window shows the following log output:

```
+ >> File "src\red-message.html" changed.
✖ Running "clean:dist" (clean) task
Cleaning dist...OK

Running "bosonic:components" (bosonic) task

Running "copy:dist" (copy) task
Copied 2 files

Done, without errors.
Completed in 2.868s at Thu Jan 22 2015 22:44:09 GMT+0530
(India Standard Time) - Waiting...
```

The following code has the content of the `red-message.js` file which is generated by the Grunt transpiler:

```
(function () {
    var RedMessagePrototype = Object.create(HTMLElement.prototype, {
        createdCallback: {
            enumerable: true,
            value: function () {
                this.appendChild(this.template.content.cloneNode(true));
            }
        });
    window.RedMessage = document.registerElement('red-message', {
        prototype: RedMessagePrototype });
    Object.defineProperty(RedMessagePrototype, 'template', {
        get: function () {
            var fragment = document.createDocumentFragment();
            var div =
fragment.appendChild(document.createElement('div'));
            div.innerHTML = ' <h1>Welcome to Bosonic framework.</h1> ';
            while (child = div.firstChild) {
                fragment.insertBefore(child, div);
            }
        }
    })
})();
```

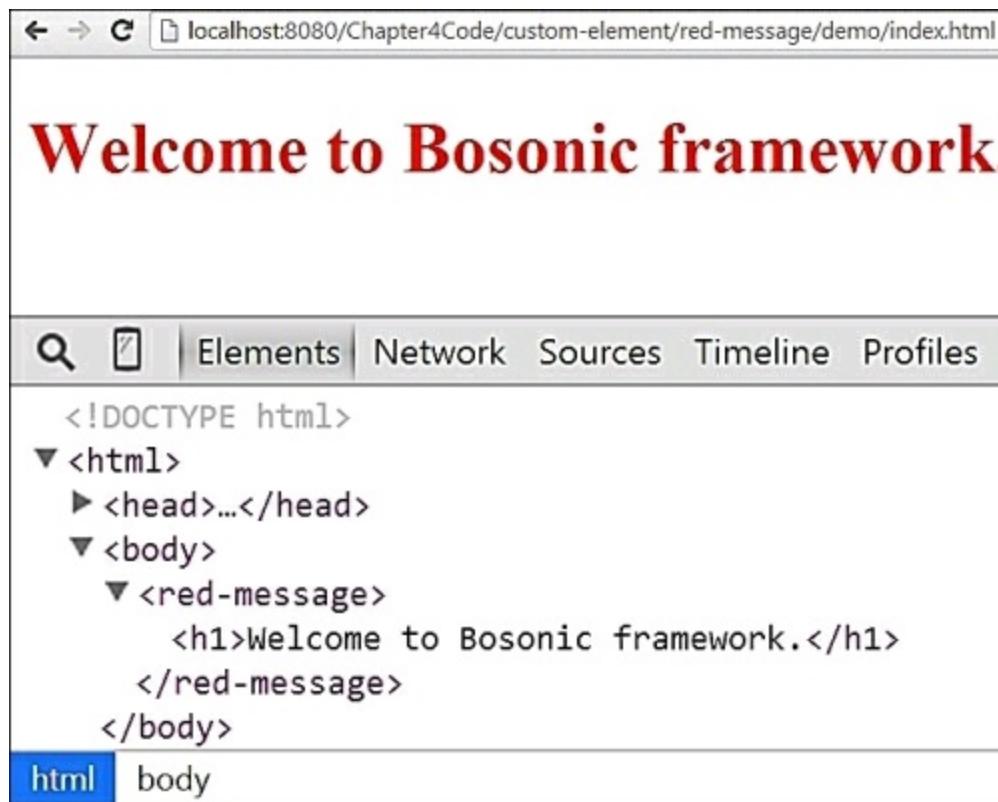
```
        fragment.removeChild(div);
        return { content: fragment };
    }
}) ;
} () ;
```

The details of the preceding code are listed as follows:

- A new object prototype `RedMessagePrototype` is created using the `Object.createElement` method
- It uses the `Object.defineProperty` method to create a property for the `RedMessagePrototype` object
- It registers the element with DOM as `red-message` using the `document.registerElement` method

## Step 8 – running the index.html file

We can see the use of the `<red-message>` custom element by running the `index.html` file. The following screenshot shows the output of `index.html`, which shows the text message in the color red:



# Bosonic lifecycle

A Bosonic element goes through different states during its lifecycle. Bosonic provides callback methods for each state to override with our own code. The lifecycle callback methods are listed as follows:

- `createdCallback`: This callback method is for handling the `created` state. An element is in the `created` state when it is instantiated at first.
- `attachedCallback`: This callback method is for handling the DOM `attached` state. An `attached` state represents the element when it is attached to DOM tree.
- `detachedCallback`: This callback method is for handling the DOM `detached` state. A `detached` state represents the element when it is removed from the DOM.
- `attributeChangedCallback`: This callback method is for handling the changes in attributes.

## Example of lifecycle

In this section, we will develop a custom element `<say-hello>` demonstrating the lifecycle callback method. A skeleton for creating a custom element `<say-hello>` can be generated using the previously explained steps. The following screenshot shows the terminal with the previous steps in execution:

Terminal

```
+ J:\Chapter4Code\life-cycle>mkdir say-hello
x
J:\Chapter4Code\life-cycle>cd say-hello

J:\Chapter4Code\life-cycle\say-hello>yo bosonic

      _-----_
     |       |
     |---(o)--|
     \-----|   Welcome to Yeoman,
      ( _ 'U` _ )   |   ladies and gentlemen!
      /__A__\  |
      | ~ |
      ' .--. '
      ' ` ' Y '


You're using the Bosonic generator. You will now generate
a boilerplate for a new Web Component.
? What do you want to call your new element? (say-hello)
? What do you want to call your new element? say-hello
  create package.json
  create Gruntfile.js
  create karma.conf.js
  create test\say-hello.js
```

After the directory and file structure is generated, we can start the Grunt task, which will watch over the modified element and generate the `dist` directory containing the respective CSS and JS files. The `say-hello.html` file present in the `src` directory has the code definition for the `<say-hello>` element. The following code shows the definition of the `<say-hello>` element:

```
<element name="say-hello" message="">
  <template>
    <h1>Hello <span></span></h1>
  </template>
  <script>
  ({
    createdCallback: function() {
      console.log(this.localName+" Element is created.");
    }
  })
```

```

        var shadowRootNode = this.createShadowRoot(),
            templateContent = this.template.content.cloneNode(true),
            spanPlaceHolder = templateContent.querySelector("span");
spanPlaceHolder.innerText = this.getAttribute("message");
shadowRootNode.appendChild(templateContent);
},
attachedCallback: function() {
    console.log(this.localName+" is attached to DOM.");
},
detachedCallback: function() {
    console.log(this.localName+" is removed from DOM.");
},
attributeChangedCallback: function(attributeName) {
    var newMessageValue = this.getAttribute("message");
console.log(attributeName+" value is changed to "+ newMessageValue);
    this.createdCallback();
}
});
</script>
</element>

```

The details of the preceding code are listed as follows:

- The name of the custom element is `say-hello` and a `message` attribute with no value.
- The `<template>` element contains the HTML markup containing a header `<h1>` and a `<span>` tag. The `<span>` tag acts as a placeholder for the `message` attribute value.
- The element definition contains four lifecycle callback methods. The details of these callback methods are listed as follows:
  - The `createdCallback` method clones the template content, reads the `message` attribute value, and inserts it as a text node to the `<span>` element. It has a console to log the created event.
  - The `attachedCallback` method has just the console log to print the message when the element is attached to DOM.
  - The `detachedCallback` method has a console log to print the message when the element is removed from the DOM.
  - The `attributeChangesCallback` method calls the `createdCallback` method whenever the value of the `message` attribute is changed. It also contains the console log to print the message whenever any attribute is changed.

The `<say-hello>` element will generate the `say-hello.css` and `say-hello.js` files

in the `dist` directory with a copy inside the `demo` directory. The `demo` directory contains the `index.html` file, which shows the use of the `<say-hello>` element. The code for the `index.html` file is listed as follows:

```
<!DOCTYPE html>
<html>
<head>
  <title>Say-Hello Element Demo</title>
  <meta charset="utf-8">
  <script src="js/say-hello.js"></script>
</head>
<body>
  <div id="elementContainer">
    <b><say-hello message="Web Component"></say-hello></b>
  </div>

  <input type="text" id="messageElement" placeholder="Enter some message">
  <br><br>

  <button onclick="changeMessageHandler()">Change Message</button>
  <button onclick="deleteElementHandler()">Delete SayHello Element</button>
  <button onclick="createElementHandler()">Create SayHello Element</button>

  <script type="text/javascript">
    //changes the value of the message
    var changeMessageHandler = function() {
      var sayHelloElement = document.querySelector("say-hello");
      messageText = document.getElementById("messageElement").value;
      sayHelloElement.setAttribute("message", messageText);
    },
    //delete the say-hello element
    deleteElementHandler = function() {
      var sayHelloElement = document.querySelector("say-hello");
      sayHelloElement.remove();
    },
    //create new say-hello element
    createElementHandler = function() {
      var sayHelloElement = document.createElement("say-hello");
      elementContainer = document.getElementById("elementContainer");
      sayHelloElement.setAttribute("message", "Developers");
      elementContainer.appendChild(sayHelloElement);
    };
  </script>
</body>
```

</html>

In the `index.html` file, the code contains an `<input>` element having the `messageElement` ID value and three buttons for modifying the `<say-hello>` element. The details of these elements are listed as follows:

- The `<input>` element takes the value from the user, which can be used as a value for the `message` attribute.
- The first button **Change Message** value has an onclick handler attached to the `changeMessageHandler` method. This method takes the value entered by the user in the `<input>` element and updates the `message` attribute. The change in the `message` attribute calls the `attributeChangedCallback` method, which in turn calls the `createdCallback` method and the new value of the message then gets rendered in the browser.
- The second button **Delete SayHello Element** has an onclick handler attached to the `deleteElementHandler` method. This method finds the `say-hello` element and removes it from the DOM using the `remove` method. It fires the `detachedCallback` lifecycle method.
- The third button **Create SayHello Element** has an onclick handler attached to the `createElementHandler` method. This method creates a new `say-hello` element using the `document.createElement` method and then attaches the `say-hello` element to `elementContainer`. This process fires the `createdCallback` lifecycle method.

The output of the demo code in `index.html` will render the following screenshot:

# Hello Web Component

The screenshot shows the browser's developer tools with the "Elements" tab selected. The DOM tree is displayed, starting with the root <!DOCTYPE html> and <html>. Inside the <body>, there is a <div id="elementContainer">. Within this container, there is a <say-hello message="Web Component"> element. This element has a shadow root containing an <h1> element with the text "Hello Web Component". Below the DOM tree, the status bar shows the path: html > body > #elementContainer > say-hello > #shadow-root.

```
<!DOCTYPE html>
<html>
  <head>...</head>
  <body>
    <div id="elementContainer">
      <say-hello message="Web Component">
        <#shadow-root>
          <h1>Hello Web Component</h1>
        </say-hello>
      </div>
    </body>
  </html>
```

html body #elementContainer say-hello #shadow-root

Now, enter a text message such as Bosonic Framework in the <input> element and hit the **Change Message** button. The following screenshot shows the output after entering the text in the <input> element:

## Hello Bosonic Framework

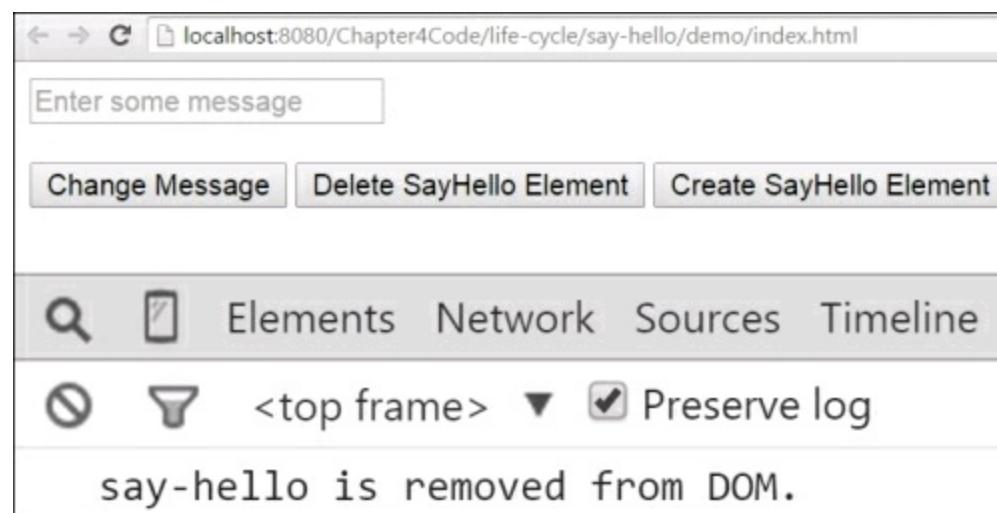
The screenshot shows the browser's developer tools with the "Elements" tab selected. The DOM tree is identical to the previous screenshot, but the status bar now shows the path: html > body > #elementContainer > say-hello > #shadow-root. The status bar also displays a log message: "message value is changed to Bosonic Framework" and "say-hello is created.".

```
message value is changed to Bosonic Framework
say-hello is created.
```

In the preceding screenshot, we can see that the new **Hello Bosonic Framework**

message gets rendered in the browser. In the console, we can find two logs generated from the `attributeChangedCallback` and `createdCallback` methods, as the `message` attribute value is changed with the user entered text and the element is reinitialized.

Now, press the **Delete SayHello Element** button which will remove the element form the DOM tree. The following screenshot shows the output after the deletion of the `<say-hello>` element from DOM:



In the preceding screenshot, we can see that the `<say-hello>` element is removed from the DOM. In the console, a message is logged which is generated by the `detachedCallback` method.

Now, press the **Create SayHello Element** button which will create a new `<say-hello>` element and append it to `elementContainer`. The following screenshot shows the new `<say-hello>` element created after pressing the button:

# Hello Developers

[Change Message](#) [Delete SayHello Element](#) [Create SayHello Element](#)

Elements Network Sources Timeline

 <top frame> ▾  Preserve log

say-hello is created.

message value is changed to Developers

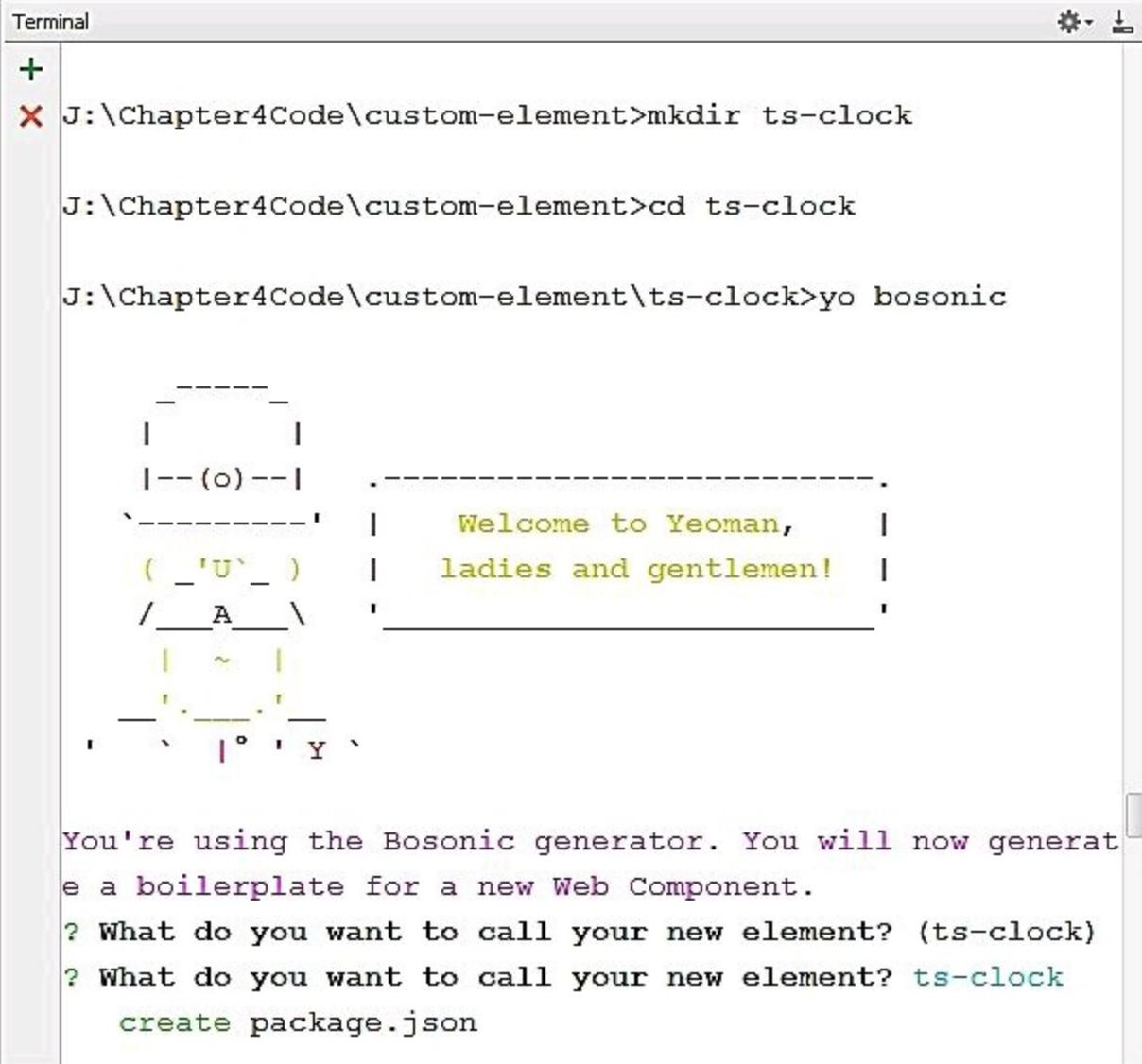
say-hello is created.

say-hello is attached to DOM.

In the preceding screenshot, we can see that the new `<say-hello>` element is rendered in the browser with the `message` attribute value set to `Developers`. In the console, we can see that four different log message are generated by the `createdCallback`, `attributeChangedCallback`, and `attachedCallback` method.

# Digital clock development

In this section, we will develop the `<ts-clock>` custom element using the Bosonic framework. The following screenshot shows the terminal with the command to generate the default directory and file structure for creating a `<ts-clock>` element:



A screenshot of a terminal window titled "Terminal". The window shows the following command sequence:

```
J:\Chapter4Code\custom-element>mkdir ts-clock  
J:\Chapter4Code\custom-element>cd ts-clock  
J:\Chapter4Code\custom-element\ts-clock>yo bosonic
```

The terminal then displays a decorative welcome message from Yeoman:

```
-----  
|       |  
|--(o)--| .-----.  
`-----' |   Welcome to Yeoman,  
 ( _ 'U`_ ) |   ladies and gentlemen!  
 /__A__\ '|  
 | ~ |  
 '-' .-' -'  
' |° ' Y '  
-----
```

Below the welcome message, the terminal prompts the user to generate a boilerplate for a new Web Component:

```
You're using the Bosonic generator. You will now generate a boilerplate for a new Web Component.  
? What do you want to call your new element? (ts-clock)  
? What do you want to call your new element? ts-clock  
  create package.json
```

Now, we can run the Grunt task runner to watch the changes to the `ts-clock.html` file. The task runner transpiles the `ts-clock.html` file and generates the `ts-clock.css` and `ts-clock.js` files in the `dist` directory, and a copy in the `demo` directory. The following screenshot shows the terminal with the Grunt task runner in execution:

```
+  
X J:\Chapter4Code\custom-element\ts-clock>grunt demo  
Running "clean:dist" (clean) task  
  
Running "bosonic:components" (bosonic) task  
  
Running "copy:lib" (copy) task  
Copied 1 files  
  
Running "copy:dist" (copy) task  
Copied 2 files  
  
Running "connect:demo" (connect) task  
Started connect web server on http://0.0.0.0:8020  
  
Running "watch" task  
Waiting...
```

Now, the platform is set up for writing the code for the `<ts-clock>` digital clock element. The `ts-clock.html` file contains the code for the definition of the digital clock. The details of the code in the `ts-clock.html` file are listed as follows:

- The `<ts-clock>` element will have three attributes `hour`, `minute`, and `second`. The following code shows the syntax for the code definition for the `<ts-clock>` element:

```
<element name="ts-clock" hour="" minute="" second="">  
  <template>  
    <!--HTML template content for ts-clock element -->  
    <!--CSS style attributes for shadow DOM element-->  
  </template>  
  <script>  
    // Lifecycle callback method for ts-clock element  
  </script>  
</element>
```

- The `<template>` element contains the CSS style and HTML elements. The CSS styles and HTML element that we used for the template is used from the previous chapter. The HTML element has three different `div` elements, which has `hour`, `minute`, and `second` classes wrapped inside a `div` element with the

`clock` class. The following code shows the HTML and CSS content present inside the `<template>` element:

```
<template>
  <style>
    :host .clock{
      display: inline-flex;
      justify-content: space-around;
      background: floralwhite;
      font-size: 2rem;
      font-family: serif;
    }
    :host .clock .hour,
    :host .clock .minute,
    :host .clock .second{
      color: tomato;
      padding: 1.5rem;
      text-shadow: 0px 1px grey;
    }
  </style>
  <div class="clock">
    <div class="hour"></div>
    <div class="minute"></div>
    <div class="second"></div>
  </div>
</template>
```

- The `<script>` element contains the real code definition for the `<ts-clock>` element. The following code shows the content of the `<script>` element:

```
<script>
  ({
    createdCallback: function() {
      var shadowRootNode = this.createShadowRoot(),
          content = this.template.content.cloneNode(true);
      shadowRootNode.appendChild(content);
    },
    attachedCallback: function() {
      var clockElement = this;
      window.setInterval(function() {
        var date = new Date();
        clockElement.setAttribute("hour", date.getHours());
        clockElement.setAttribute("minute",
date.getMinutes());
        clockElement.setAttribute("second",
date.getSeconds());
      }, 1000);
    },
  })
```

```

attributeChangedCallback: function(attributeName) {
    var shadowRootNode = this.shadowRoot,
        hourPlaceholder =
shadowRootNode.querySelector('.hour'),
        minutePlaceholder =
shadowRootNode.querySelector('.minute'),
        secondPlaceHolder =
shadowRootNode.querySelector('.second');
    switch (attributeName) {
        case "hour":
            hourPlaceholder.innerText =
this.getAttribute("hour");
            break;
        case "minute":
            minutePlaceholder.innerText =
this.getAttribute("minute");
            break;
        case "second":
            secondPlaceHolder.innerText =
this.getAttribute("second");
            break;
    }
}
});
</script>

```

The details of the preceding code are listed here:

- The `createdCallback` method creates `shadowRoot` for the host element. It finds and clones the content of the element to activate it. After activation, it appends the HTML content to `shadowRoot`.
- The `attachedCallback` method has a `setInterval` method, which contains the code for setting the attributes `hour`, `minute`, and `second` with the current time using a `date` object. The code block gets executed in every 1 second and updates the value of the `hour`, `minute`, and `second` attributes.
- The `attributeChangedCallback` method gets called whenever there is change of value in the `hour`, `minute`, and `second` attributes. It has a `switch` block containing cases like `hour`, `minute`, and `second`. Based on the change in the attribute, one of the `switch case` gets activated, which updates the text content of the DOM element.

The `demo` directory contains the code for testing the `<ts-clock>` element. The `index.html` file contains the `<ts-clock>` element with `hour`, `minute`, and `second` having default values `HH`, `MM`, and `SS`, respectively. The content of the `index.html`

file is listed as follows:

```
<!DOCTYPE html>
<html>
<head>
  <title>Digital Clock Demo</title>
  <meta charset="utf-8">
  <script src="js/ts-clock.js"></script>
  <link href="css/ts-clock.css" rel="stylesheet">
</head>
<body>
  <ts-clock hour="HH" minute="MM" second="SS">
  </ts-clock>
</body>
</html>
```

The output of the preceding code is rendered in the following screenshot showing hour (HH), minute (MM), and second (SS) in the 24-hour format:



The following screenshot shows the Chrome developer tool with the HTML inspection for the output of the preceding code. We can find the Shadow DOM subtree of the `<ts-clock>` element containing the CSS style and HTML element.

```
<!DOCTYPE html>
▼<html>
► <head>...</head>
▼ <body>
▼ <ts-clock hour="22" minute="35" second="33">
▼ #shadow-root
▼ <style>
  :host .clock{ display: inline-flex; justify-content: space-around; background: floral white; font-size: 2rem; font-family: serif; }
  :host .clock .hour,:host .clock .minute,
  :host .clock .second{ color: tomato; padding: 1.5rem; text-shadow: 0px 1px grey; }
</style>
▼ <div class="clock">
  <div class="hour">22</div>
  <div class="minute">35</div>
  <div class="second">33</div>
</div>
</ts-clock>
```

html

body

# Summary

In this chapter, we learned about the Bosonic framework that is used to create a custom element with the lifecycle callback methods. In the next chapter, we will learn about the Mozilla Brick framework based on X-Tag.

# Chapter 5. Developing Web Components Using Mozilla Brick

In this chapter, we will learn about the Mozilla Brick UI components. We will cover the installation and configuration of Brick in an application. We will also explore what is upcoming in Mozilla Brick 2.0.

## What is the Brick library?

The Mozilla Brick library provides a set of elements based on the web component specification. The motto behind the Brick framework is *strong opinions, tightly scoped*. This means that the Brick framework does not focus on code structure, data binding, or any other problem. It provides the web interface that can be easily integrated with any other library. Brick elements are developed on top of the custom element API. This custom element API involves shadow DOM, HTML Imports, template element, and custom element. Brick provides a standard interface that can easily be interpolated with other application frameworks.

Mozilla Brick provided a **UIKit** for the web that can be used by the developers to create web application. Brick components are written in vanilla style JavaScript for development.

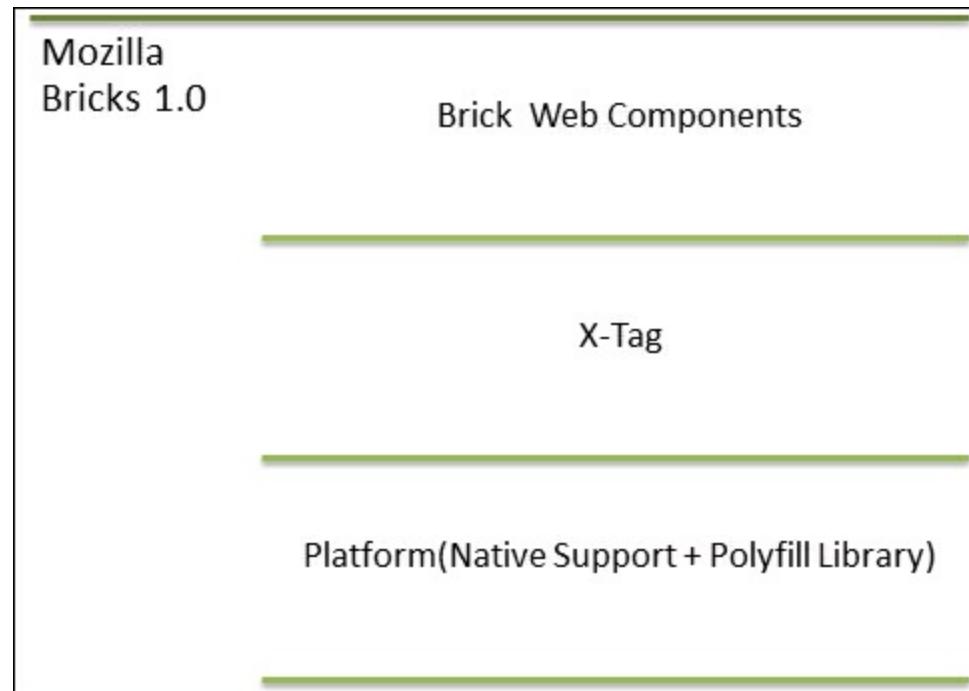
### Note

To find out more about Mozilla Brick framework visit:

<https://mozbrick.github.io/>

# Mozilla Brick 1.0

The current released version of Mozilla Brick is 1.0. The following diagram shows the building blocks of the Mozilla Brick framework:

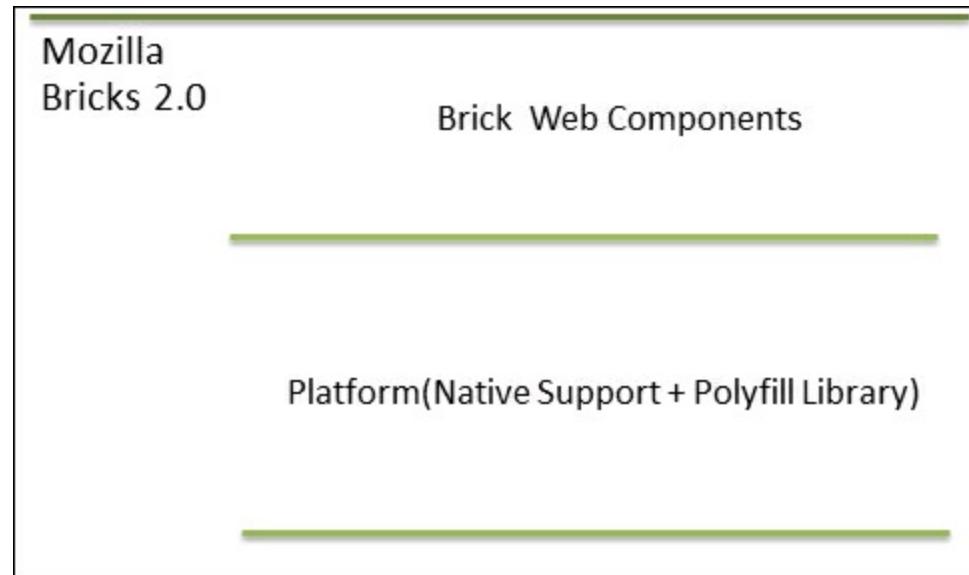


The details of the preceding Mozilla Brick 1.0 block diagram are listed here:

- **Platform:** This represents the native support for web component specification and the polyfill script files for unsupported features.
- **X-Tag:** This represents the X-Tag libraries to create elements.
- **Brick web components:** This represents the UI toolkit of built-in elements for developers to use in web applications.

# Mozilla Brick 2.0

Mozilla Brick 2.0 is the new version of Brick framework. This library is released for application development. The following screenshot shows the block diagram of Mozilla Brick 2.0:



The details of the preceding Mozilla Brick 2.0 block diagram are listed here:

- **Platform:** This represents the native support for web component specification and the polyfill script files for unsupported features. This polyfill file is `platform.js`, which has been used by the PolymerJS framework. In the current release of PolymerJS, the `platform.js` file has been renamed to `webcomponents.js`.

## Note

You can find more information on polyfill using the following link:

<http://webcomponents.org/polyfills>

- **Brick web components:** This represents a set of built-in UI elements that can be used by the developer in web application.

# Installing Mozilla Brick

The Mozilla Brick framework can be installed using the Bower package manager. If Bower is not installed on your system, then use the following link to install it:

<http://bower.io/#install-bower>

Assuming that the Bower is installed on your system, use the following command to install Mozilla Brick:

```
bower install mozbrick/brick
```

The following screenshot shows the terminal with Mozilla Brick installation using the Bower package manager:



```
+ J:\Chapter5Code>bower install mozbrick/brick
bower not-cached    git://github.com/mozbrick/brick.git#*
bower resolve      git://github.com/mozbrick/brick.git#*
bower download     https://github.com/mozbrick/brick/
archive/2.0.0.tar.gz
bower progress     brick#* received 0.5MB
bower extract      brick#* archive.tar.gz
bower mismatch    Version declared in the json (2.0.
```

Once the Mozilla Brick is installed successfully, it creates a parent directory `bower_components` where other Brick components are copied to their respective subdirectories. The following screenshot shows the terminal with the created directory structure:

+ J:\Chapter5Code>cd bower\_components

X

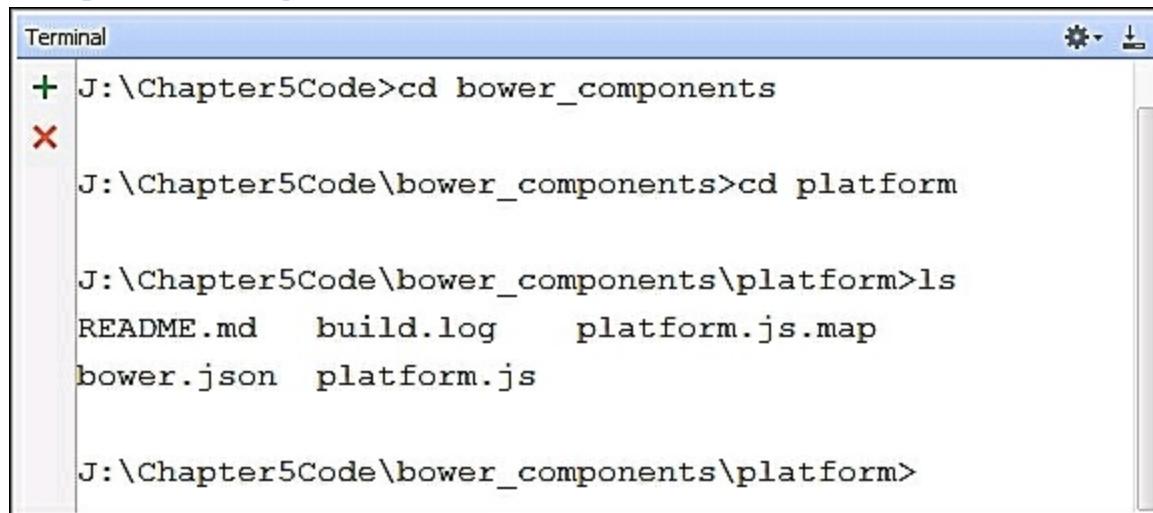
J:\Chapter5Code\bower\_components>ls

brick	brick-layout
brick-action	brick-menu
brick-appbar	brick-storage-indexeddb
brick-calendar	brick-tabbar
brick-common	es6-promise
brick-deck	font-awesome
brick-flipbox	platform
brick-form	

# Configuring Mozilla Brick

The Mozilla Brick library can be configured to a web application by including two files inside the `<head>` element. These two files are listed as follows:

- `platform.js`: This file contains all the polyfill scripts needed for supporting web component specification. The `platform.js` polyfill library is present inside the `platform` subdirectory. The following screenshot shows the terminal with the `platform.js` file:



A screenshot of a terminal window titled "Terminal". The window shows the following command-line session:

```
+ J:\Chapter5Code>cd bower_components
X
J:\Chapter5Code\bower_components>cd platform
J:\Chapter5Code\bower_components\platform>ls
README.md    build.log    platform.js.map
bower.json   platform.js

J:\Chapter5Code\bower_components\platform>
```

- `Brick.html`: This file combines all the Brick elements into a single file and places it inside the `dist` subdirectory under `brick` directory. The following screenshot shows the terminal for the `brick.html` directory structure:

```
+  
X J:\Chapter5Code>cd bower_components  
  
J:\Chapter5Code\bower_components>cd brick  
  
J:\Chapter5Code\bower_components\brick>ls  
CONTRIBUTING.md  bower.json      manifest.webapp  
LICENSE           dist          package.json  
README.md         gulpfile.js    tasks  
assets            karma.conf.js  
  
J:\Chapter5Code\bower_components\brick>cd dist  
  
J:\Chapter5Code\bower_components\brick\dist>ls  
brick-action      brick-menu  
brick-appbar      brick-storage-indexeddb  
brick-calendar    brick-tabbar  
brick-common      brick.html  
brick-deck        es6-promise  
brick-flipbox     font-awesome  
brick-form        platform  
brick-layout  
  
J:\Chapter5Code\bower_components\brick\dist>
```

The Brick framework library can be added to a web page by including the `platform.js` polyfill and the `brick.html` file. The following code shows the HTML code for including Brick framework in the web page:

```
<script src="bower_components/brick/dist/platform/platform.js">  
</script>  
<link rel="import" href="bower_components/brick/dist/brick.html">
```

# Built-in components

Mozilla Brick 1.0 has many built-in components for web application development. In this section, we will explore some of them.

## The brick-calendar element

The `brick-calendar` element represents a calendar component and can be used by calling the following custom tag in the HTML page:

```
<brick-calendar></brick-calendar>
```

The `brick-calendar` element can be used by including polyfill and calendar definition files in the web application. The following code can be included to use the Brick's calendar component:

```
<script src="bower_components/platform/platform.js"></script>
<link rel="import" href="bower_components/brick- calendar/dist/brick-
calendar.html">
```

The following screenshot shows the `brick-calendar` component rendered in a browser, and the HTML element inspection using Chrome developer console:

The screenshot displays a calendar for January 2015 on the left and the Chrome developer console on the right. The calendar shows days from 28 to 31 of December 2014, and 1 to 31 of January 2015. The 26th of January is highlighted with a blue background. On the right, the developer console shows the DOM tree under the 'Elements' tab. The tree starts with the root `<!DOCTYPE html>`, followed by `<html>`, `<head>`, and `<body>`. Inside the body, there is a `<brick-calendar chosen="2015-01-26">` element. This element contains a shadow root with two styles and a content node. Below this is a `<div class="calendar">` element, which contains a `<div class="month" role="grid">` element. The entire structure is closed with a closing tag for the `<brick-calendar>` element. At the bottom of the developer console, the tabs 'html' and 'body' are visible, with 'html' currently selected.

# The brick-flipbox element

The `brick-flipbox` element is used for flipping between content using animations, and can be used by calling the following custom tag in the HTML page:

```
<brick-flipbox>
  <!--Content goes here -->
</brick-flipbox>
```

The `brick-flipbox` element can be used by including polyfill and flipbox definition files in the web application. The following code can be included to use the Brick's `flipbox` component:

```
<script src="bower_components/platform/platform.js"></script>
<link rel="import" href="bower_components/brick-flipbox/dist/brick-flipbox.html">
```

Let's check out an example of using `brick-flipbox` in a web application. The following code contains the use of `flipbox`:

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8">
  <title>Brick FlipBox Element Demo</title>
  <script src="../bower_components/platform/platform.js"></script>
  <link rel="import" href="../bower_components/brick-
flipbox/dist/brick-flipbox.html">
  <style>
    .myBox {
      width: 100px;
      height: 80px;
      margin-bottom: 20px;
    }
    .myBox > div:first-child {
      background: red;
    }
    .myBox > div:last-child {
      background: blue;
    }
  </style>
</head>
<body>
```

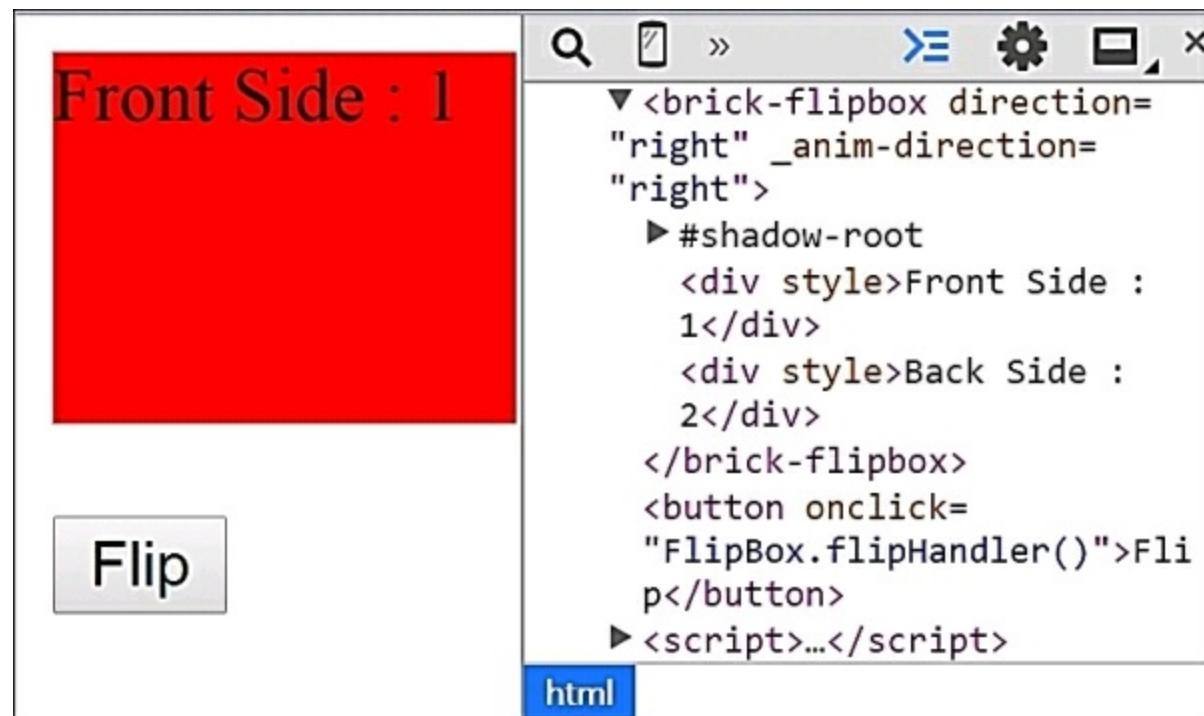
```

<brick-flipbox class="myBox">
  <div>Front Side : 1</div>
  <div>Back Side : 2</div>
</brick-flipbox>
<button id="toggleButton">Flip</button>

<script>
  window.addEventListener('WebComponentsReady',function(){
    var flipbox = document.querySelector('brick-flipbox'),
        toggleButton = document.getElementById('toggleButton');
    flipbox.direction = 'left';
    toggleButton.addEventListener("click", function(){
      flipbox.toggle();
      console.log("flipped : "+flipbox.flipped);
    });
  });
</script>
</body>
</html>

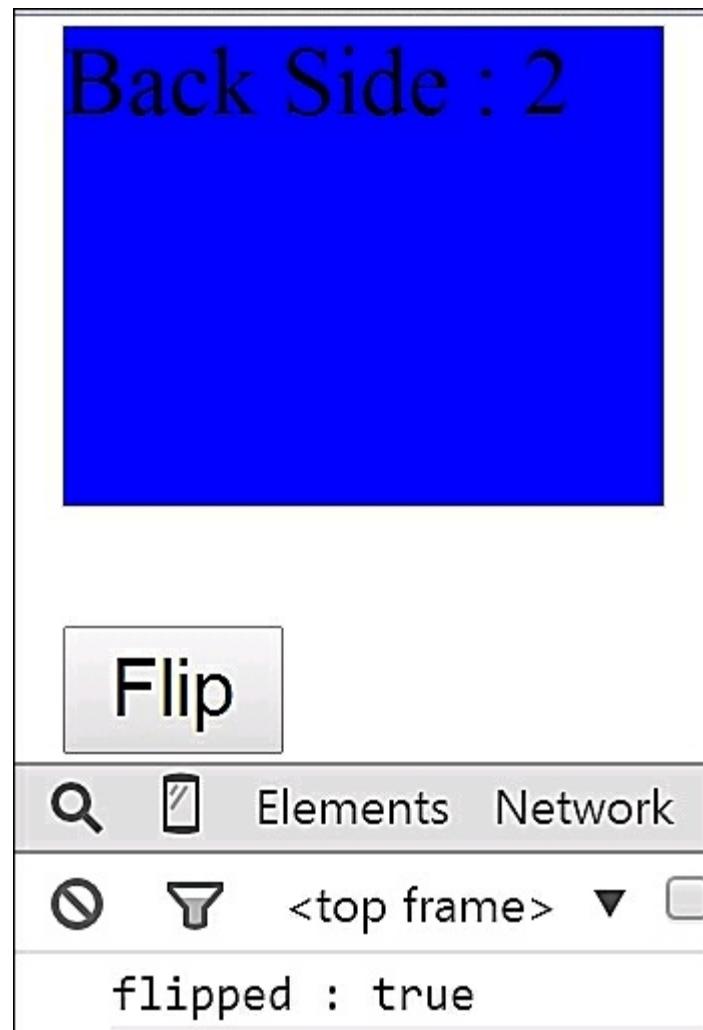
```

The output of the preceding code is listed in the following screenshot showing the content of flipbox and a **Flip** button:



When the **Flip** button is pressed, then the callback method gets called. The callback method gets the flipbox instance and calls the `toggle` method. The `toggle` method flips the content based on the `direction` property, which is set to `left`.

This method prints the value of the `flipped` property, which returns the Boolean value `true` if the flipbox is flipped. The following screenshot shows the console log message when the **Flip** button is pressed:



## Note

You can find more information about the `brick-flipbox` element using following link:

<https://mozbrick.github.io/docs/brick-flipbox.html>

## The `brick-deck` element

The `brick-deck` element contains a set of cards. Using deck elements, the cards can be ordered. The cards inside the deck can have their own transition. A `brick-deck` element can be used by including the following code in the page:

```
<brick-deck>
```

```
<brick-card></brick-card>
<brick-card></brick-card>
...
</brick-deck>
```

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

- The `<brick-deck>` element is the parent element to be used for the deck.
- The `<brick-card>` element is the child element for the deck.

The `brick-deck` element can be used by including polyfill and deck definition files in the web application. The following code can be included to use the Brick's deck component:

```
<script src="bower_components/platform/platform.js"></script>
<link rel="import" href="bower_components/brick-deck/dist/brick-deck.html">
```

Let's checkout an example using `brick-deck` in a web application. The following code contains the use of a deck containing cards with different fruit names:

```
<!DOCTYPE html>
<html>
<head lang="en">
    <meta charset="UTF-8">
    <title>Brick Deck Element Demo</title>
    <script src="../bower_components/platform/platform.js"></script>
    <link rel="import" href="../bower_components/brick-deck/dist/brick-deck.html">
    <style>
        brick-deck{
            width: 150px;
            height: 150px;
            margin-bottom: 20px;
        }
        brick-card:nth-child(even) {
            background: darkgreen;
            color:#ffffff;
        }
        brick-card:nth-child(odd) {
            background: blue;
            color:#ffffff;
        }
    </style>
</head>
<body>
```

```

<brick-deck selected-index="0" transition-type="slide-up">
    <brick-card selected>Card 0 : Mango</brick-card>
    <brick-card>Card 1 : Apple</brick-card>
    <brick-card>Card 2 : Orange</brick-card>
    <brick-card>Card 3 : Grapes</brick-card>
</brick-deck>
<button id="nextButton">Next Card</button>
<button id="prevButton">Previous Card</button>
<script>
    window.addEventListener('WebComponentsReady', function() {
        var deck = document.querySelector('brick-deck'),
            nextButton = document.getElementById('nextButton'),
            prevButton = document.getElementById('prevButton');
        nextButton.addEventListener("click", function() {
            deck.nextCard();
        });
        prevButton.addEventListener("click", function() {
            deck.previousCard();
        });
    });
</script>
</body>
</html>

```

The details of the preceding code are listed as follows:

- A deck is created using the `<brick-deck>` element with the following properties set to some predefined values. The details of these properties are listed as follows:
  - `selected-index`: This property represents the default selected card inside the deck. In this example, this property is set to 0.
  - `transition-type`: This property represents the animation direction that the deck will apply to the child cards. In this example, this has the `slide-up` value.
- There are two buttons representing next and previous card navigation.
- The two buttons having the `nextCard` and `prevCard` IDs are attached with the `click` event. This event binding is done inside the `WebComponentsReady` event callback.
- When the user presses the `nextCard` button, the deck moves to the next card, and when the user presses the `prevCard` button, the deck moves to the previous card.

The following screenshot shows the output of the preceding code where the default

card with index 0 is selected:

```
<brick-deck selected-index="0" transition-type="slide-up">
  #shadow-root
    <brick-card selected transition>Card 0 : Mango</brick-card>
    <brick-card>Card 1 : Apple</brick-card>
    <brick-card>Card 2 : Orange</brick-card>
    <brick-card>Card 3 : Grapes</brick-card>
</brick-deck>
<button id="prevButton">Previous Card</button>
<button id="nextButton">Next Card</button>
```

When the user presses the `nextCard` button, the deck moves to the next card with the index 1. The value of the `selected-index` property is now 1. The following screenshot shows the next card with the **Apple** fruit:

```
<brick-deck selected-index="1" transition-type="slide-up">
  #shadow-root
    <brick-card>Card 0 : Mango</brick-card>
    <brick-card selected transition-direction="forward" transition>Card 1 : Apple</brick-card>
    <brick-card>Card 2 : Orange</brick-card>
    <brick-card>Card 3 : Grapes</brick-card>
  </brick-deck>
  <button id="prevButton">Previous Card</button>
  <button id="nextButton">Next
```

## Note

You can find more information about the `brick-deck` card element using the following link:

<http://mozbrick.github.io/docs/brick-deck.html>

## The `brick-tabbar` element

The `brick-tabbar` element represents a tab. A tab element provides a single content area with multiple panels, each associated with a header in a list. A `brick-tabbar` element can be used by including the following code in the page:

```
<brick-tabbar>
  <brick-tabbar-tab></brick-tabbar-tab>
  ...
</brick-tabbar>
```

The details of the preceding code are listed as follows:

- The `<brick-tabbar>` element is the parent element to be used for a group of tabs.
- The `<brick-tabbar-tab>` element is the child tabbar and represents the individual tab.

The `brick-tabbar` element can be used by including polyfill and tabbar definition files in the web application. The following code can be included to use the Brick's tabbar component:

```
<script src="bower_components/platform/platform.js"></script>
<link rel="import" href="bower_components/brick-tabbar/dist/brick-tabbar.html">
```

Let's check out an example using `brick-tabbar` in a web application. The following code contains the use of tabbar containing different tabs:

```
<!doctype html>
<html>
<head>
    <meta charset="utf-8">
    <title>Brick Tabbar Element Demo</title>
    <script src="../bower_components/platform/platform.js"></script>
    <link rel="import" href="../bower_components/brick-
tabbar/dist/brick-tabbar.html">
</head>
<body>
<b>brick-tabbar</b>
    <b>brick-tabbar-tab id="fruits" selected>
        Fruits
    </b>
    <b>brick-tabbar-tab id="flowers">
        Flowers
    </b>
    <b>brick-tabbar-tab id="vegetables">
        Vegetables
    </b>
</b>
<button name="fruits">Select Fruits</button>
<button name="flowers">Select Flowers</button>
<button name="vegetables">Select Vegetables</button>

<script>
```

```
window.addEventListener('WebComponentsReady',function() {
    var buttonList = document.querySelectorAll("button");
    for (var i = 0; i < buttonList.length; ++i) {
        var item = buttonList[i];
        item.addEventListener("click", function(e) {
            var name = e.target.name,
                tabBar = document.getElementById(name);
            tabBar.select();
        });
    }
});
```

</script>

</body>

</html>

The details of the preceding code are listed here:

- There are three different tabs: **Fruits**, **Flowers**, and **Vegetables**. They are defined using `<brick-tabbar-tab>` inside a `<brick-tabbar>` element.
- There are three different buttons defined for selecting fruits, flowers, and vegetables. The callback function attached to each button has a `select` method to dynamically select a tab element.

The following screenshot shows the output of the preceding code with three different tabs **Fruits**, **Flowers**, and **Vegetables**. The **Fruits** tab element is selected by default:

The screenshot shows a browser window with the URL `localhost:8080/Chapter5Code/builtin-element/brickTabBarDemo.html`. At the top, there are three tabs labeled **Fruits**, **Flowers**, and **Vegetables**. Below them are three buttons: **Select Fruits**, **Select Flowers**, and **Select Vegetables**. The developer tools element inspector is open, with the **Elements** tab selected. The tree view shows the following structure:

```
<brick-tabbar width:>
  #shadow-root
    <brick-tabbar-tab id="fruits" selected="true">
      Fruits
    </brick-tabbar-tab>
    <brick-tabbar-tab id="flowers">
      Flowers
    </brick-tabbar-tab>
    <brick-tabbar-tab id="vegetables">
      Vegetables
    </brick-tabbar-tab>
  </brick-tabbar>
  <button name="fruits">Select Fruits</button>
  <button name="flowers">Select Flowers</button>
  <button name="vegetables">Select Vegetables</button>
```

The **html** tab is also visible at the bottom left of the inspector.

When the user presses the **Select Vegetables** button, the tab selection focuses the changes to **Vegetables** with the `selected` property set to `true`. The following screenshot shows that the **Vegetables** tab element is selected programmatically:

The screenshot shows a browser window with the URL `localhost:8080/Chapter5Code/builtin-element/brickTabBarDemo.html`. At the top, there is a navigation bar with icons for back, forward, and refresh. Below it is a header with three tabs: `Fruits`, `Flowers`, and `Vegetables`. The `Vegetables` tab is currently selected. Underneath the tabs, there are three buttons: `Select Fruits`, `Select Flowers`, and `Select Vegetables`. The `Select Vegetables` button is highlighted with a blue border. The main content area shows the source code of the `brick-tabbar` component. The code includes the following structure:

```
<brick-tabbar width:>
  >#shadow-root
    <brick-tabbar-tab id="fruits">
      Fruits
    </brick-tabbar-tab>
    <brick-tabbar-tab id="flowers">
      Flowers
    </brick-tabbar-tab>
    <brick-tabbar-tab id="vegetables" selected="true">
      Vegetables
    </brick-tabbar-tab>
  </brick-tabbar>
  <button name="fruits">Select Fruits</button>
  <button name="flowers">Select Flowers</button>
  <button name="vegetables">Select Vegetables</button>
```

The `html` tab is selected at the bottom left of the developer tools.

## Note

You can find more information about the `brick-tabbar` element using the following link:

<https://mozbrick.github.io/docs/brick-tabbar.html>

## The `brick-action` element

The `brick-action` element binds an event with a method of a different element. A `brick-action` element has a listener, which continuously listens to the source element to detect the specified event and calls the method of the target element. A `brick-action` element can be used by including the following code in the page:

```
<brick-action></brick-action>
```

The `brick-action` element can be used by including polyfill and action definition files in the web application. The following code can be included to use the Brick's action component:

```
<script src="bower_components/platform/platform.js"></script>
<link rel="import" href="bower_components/brick-action/dist/brick-action.html">

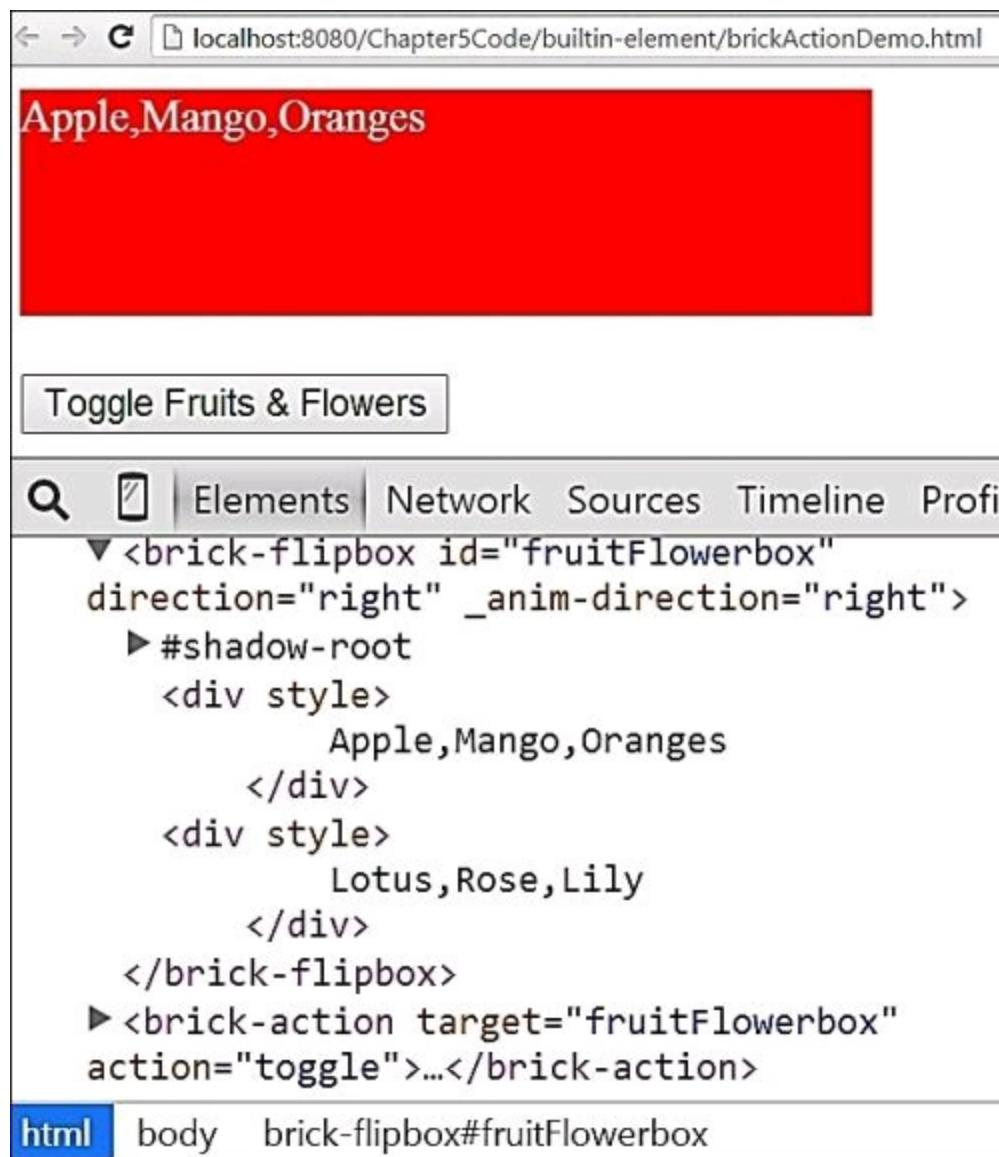
Let's check out an example using brick-action in a web application. The following code contains the use of an action element with a flipbox:
```

```
<!doctype html>
<html>
<head>
    <meta charset="utf-8">
    <title>Brick Action Element Demo</title>
    <script src="../bower_components/platform/platform.js"></script>
    <link rel="import" href="../bower_components/brick-
flipbox/dist/brick-flipbox.html">
    <link rel="import" href="../bower_components/brick-action/dist/brick-action.html">
    <style>
        brick-flipbox {
            width: 300px;
            height: 80px;
            margin-bottom: 20px;
            color: #fff;
        }
        brick-flipbox > div:first-child {
            background: red;
        }
        brick-flipbox > div:last-child {
            background: blue;
        }
    </style>
</head>
<body>
<brick-flipbox id="fruitFlowerbox">
    <div>
        Apple, Mango, Oranges
    </div>
    <div>
        Lotus, Rose, Lily
    </div>
</brick-flipbox>
<brick-action target="fruitFlowerbox" action="toggle">
    <button>Toggle Fruits & Flowers</button>
</brick-action>
</body>
</html>
```

The details of the preceding code are listed as follows:

- A `<brick-flipbox>` element has two different lists of fruits and flowers with the `id` attribute set to `fruitFlowerbox`.
- A `<brick-action>` element is defined with the `target` and `action` attribute. The `target` attribute is set to `fruitFlowerbox` and the `action` attribute is set to the `toggle` method.
- When the user clicks on the button present inside the `<brick-action>` element, it finds the `target` element `fruitFlowerbox` and executes its `toggle` method.

The following screenshot shows the output of the preceding code where the `brick-action` element is bound to a flipbox:



When the user presses the **Toggle Fruits & Flowers** button the `toggle` method

gets called on the flipbox. The following screenshot shows the output of the flipbox after the **Toggle Fruits & Flowers** button is clicked on:

localhost:8080/Chapter5Code/builtin-element/brickActionDemo.html

Lotus,Rose,Lily

Toggle Fruits & Flowers

Elements Network Sources Timeline Prof

```
▼<brick-flipbox id="fruitFlowerbox" direction="right" _anim-direction="right" flipped>
  ►#shadow-root
    <div style>
      Apple,Mango,Oranges
    </div>
    <div style>
      Lotus,Rose,Lily
    </div>
  </brick-flipbox>
▼<brick-action target="fruitFlowerbox" action="toggle">
```

html body brick-flipbox#fruitFlowerbox

## Note

You can find more information about the `brick-action` element using the following link:

<https://mozbrick.github.io/docs/brick-action.html>

## The `brick-menu` element

The `brick-menu` element represents a simple menu containing different items inside it for selection. A menu element can be used inside a page by including the

following code:

```
<brick-menu>
  <brick-item></brick-item>
...
</brick-menu>
```

The `brick-menu` element can be used by including polyfill and menu definition files in the web application. The following code can be included to use the Brick's menu component:

```
<script src="bower_components/platform/platform.js"></script>
<link rel="import" href="bower_components/brick-menu/dist/brick-menu.html">
```

Let's check out an example using `brick-menu` in a web application. The following code contains the use of the menu element with different items:

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8">
  <title>Brick Menu Element Demo</title>
  <script src="../bower_components/platform/platform.js"></script>
  <link rel="import" href="../bower_components/brick-menu/dist/brick-menu.html">
</head>
<body>

<brick-menu>
  <brick-item selected>Fruits</brick-item>
  <brick-item>Flowers</brick-item>
  <brick-item>Vegetables</brick-item>
  <brick-item>Beverages</brick-item>
</brick-menu>

</body>
</html>
```

In the preceding code, a menu is defined using the `<brick-menu>` element. It contains Fruits, Flowers, Vegetables, and Beverages as the `<brick-item>` elements. The following screenshot shows the output of the preceding code with a rendered menu:

localhost:8080/Chapter5Code/builtin-element/brickMenuDemo.html

Fruits

Flowers

Vegetables

Beverages

Elements Network Sources Timeline Profiles

▼ <html>

  ► <head>...</head>

  ▼ <body>

    ► <brick-menu>

      ► #shadow-root

        <brick-item selected>Fruits</brick-item>

        <brick-item>Flowers</brick-item>

        <brick-item>Vegetables</brick-item>

        <brick-item>Beverages</brick-item>

      </brick-menu>

html body

# The X-Tag library

The Mozilla Brick 1.0 framework is based on the X-Tag library. You can find details about X-Tag in [Chapter 1, Introducing Web Components](#). The following sections focus on custom web component development using X-Tag.

You can create your own bundle and download the X-Tag library using the following link:

<http://www.x-tags.org/download>

The following screenshot shows the downloaded page with checkboxes, and with different features that can be selected to make a custom build:

The screenshot shows a web page for downloading the X-Tag library. At the top, there is a navigation bar with links for 'Mixin-Request [0.2.1] Blog XHR made easy <X> Docs About Download'. The 'Download' link is underlined, indicating it is the active section. Below the navigation, there are two main sections: 'Libraries' and 'Polyfills'. Under 'Libraries', there is one item: 'Core [0.9.9] - X-Tag core library for creating custom elements as defined by the w3c Web Components spec.' with a checked checkbox. Under 'Polyfills', there are five items, each with a checked checkbox: 'MutationObserver [0.2.0] - Mutation Observers Polyfill', 'WeakMap [0.2.1] - Shim for ES6 WeakMap', 'Custom-Elements-Polyfill [0.0.5] - Custom Element polyfill for Web Components.', 'Dom-Token-List-Polyfill [0.0.2] - Polyfills DOM Token List in IE9', and 'Font-Awesome [4.1.0] - Font Awesome'. At the bottom of the page, there is a green button labeled 'Build Bundle' and a blue button labeled 'Download'. A status message '7 components selected. Minified' is displayed between the two buttons.

The downloaded build directory contains two files. Details about these files are

listed here:

- `x-tag-components.min.css`: This is a minified CSS file containing style attributes for the downloaded content.
- `x-tag-components.min.js`: This is a minified JS file containing core X-Tag library and polyfill script files.

## Note

If you want to know more about X-Tag, use the following link:

<http://x-tag.readme.io/v1.0/docs/getting-started>

# Developing a digital clock using X-Tag

In this section, we will develop a `<ts-clock>` element using X-Tag libraries with lifecycle methods. The definition code for `<ts-clock>` element has the following three different sections:

- The X-Tag core libraries with polyfill script bundled as a single minified JS file named `x-tag-components.min.js`. This JS file should be included in the top `ts-clock.html` file containing the definition of a digital clock in the `<script>` element. The following code shows the `<script>` element for X-Tag libraries:

```
<script src="x-tag-components.min.js"></script>
```

- The `<template>` element contains the HTML markup for the digital clock and CSS style attributes for the clock's template element. The template code of the digital clock element is listed as follows:

```
<template id="clockTemplate">
  <style>
    :host .clock {
      display: inline-flex;
      justify-content: space-around;
      background: floralwhite;
      font-size: 2rem;
      font-family: serif;
    }
    :host .clock .hour,
    :host .clock .minute,
    :host .clock .second {
      color: tomato;
      padding: 1.5rem;
      text-shadow: 0px 1px grey;
    }
  </style>
  <div class="clock">
    <div class="hour"></div>
    <div class="minute"></div>
    <div class="second"></div>
  </div>
</template>
```

- The `<script>` element contains the registration and definition of the `<ts-clock>` element using the `xtag.register` method. The registration script for the `<ts-clock>` element is listed as follows:

```
<script>
  (function() {
    var thisDoc = document._currentScript.ownerDocument;
    xtag.register('ts-clock', {
      lifecycle: {
        created: function() {
          var shadowRoot = this.createShadowRoot(),
              template=thisDoc.getElementById('clockTemplate'),
              templateContent = template.content,
              activeClockTemplate=templateContent.cloneNode(true);
              shadowRoot.appendChild(activeClockTemplate);
        },
        accessors: {
          hour: {attribute: {}},
          minute: {attribute: {}},
          second: {attribute: {}}
        },
        inserted: function() {
          var clockElement = this;
          window.setInterval(function() {
            var date = new Date();
            clockElement.setAttribute("hour", date.getHours());
            clockElement.setAttribute("minute",date.getMinutes());
            clockElement.setAttribute("second",date.getSeconds());
            }, 1000);
        },
        attributeChanged: function(attributeName, oldValue, newValue) {
          var shadowRootNode = this.shadowRoot,
              hourPlaceholder = shadowRootNode.querySelector('.hour'),
              minutePlaceholder =
shadowRootNode.querySelector('.minute'),
              secondPlaceholder =
shadowRootNode.querySelector('.second');
              switch (attributeName) {
                case "hour":
                  hourPlaceholder.innerText =
this.getAttribute("hour");
                  break;
                case "minute":
                  minutePlaceholder.innerText =
this.getAttribute("minute");
                  break;
                case "second":
                  secondPlaceholder.innerText =
this.getAttribute("second");
                  break;
              }
        }
      }
    })
  }
```

```

        }
    });
})();
</script>

```

The details of the preceding code are listed here:

- The `thisDoc` variable contains the reference of `ownerDocument` of the `_currentScript` value before jumping to the registering of the `<ts-clock>` element.
- The `created` callback method creates a `shadowRoot` element using the `createShadowRoot` method, and appends the template content by cloning it using the `cloneNode` method.
- The `<ts-clock>` element has three attributes `hour`, `minute`, and `second` defined inside the `accessors` block. These attributes are linked to `{}`, which binds these properties as attributes of the `<ts-clock>` element.
- The `inserted` callback contains the `window.setTimeout` method. It contains the code for setting the value of the `hour`, `minute`, and `second` attribute using the `setAttribute` method. This code block is called every 1 second.
- The `attributeChanged` callback contains code for DOM manipulation based on the new value of the changed attribute.

The `<ts-clock>` element can be used by importing the definition to a web page using `HTMLImport`. The following code shows the use of the `<ts-clock>` element by importing `HTMLImport` to a page:

```

<!DOCTYPE html>
<html>
<head lang="en">
    <meta charset="UTF-8">
    <title>ts-clock element demo</title>
    <link rel="import" href="ts-clock.html">
</head>
<body>
    <ts-clock></ts-clock>
<script>
    window.addEventListener('WebComponentsReady', function() {
        var tsClock = document.querySelector("ts-clock");
        window.setTimeout(function() {
            console.log("Hour : ", tsClock.getAttribute('hour'));
            console.log("Minute : ", tsClock.getAttribute('minute'));
            console.log("Second : ", tsClock.getAttribute('second'));
        }, 1000)
    })
});
})();

```

```
});  
</script>  
</body>  
</html>
```

The details of the preceding code are listed as follows:

- The `<ts-clock>` element is included to the page using the following code:  
`<link rel="import" href="ts-clock.html">`
- The `<ts-clock>` element is called by including the following code inside the `<body>` element:  
`<ts-clock></ts-clock>`
- Inside the `WebComponentsReady` event callback function, the `<ts-clock>` element is referred using the `document.querySelector` method
- The value of the `hour`, `minute`, and `second` attribute is accessed using the `getAttribute` method

The following screenshot shows the output of the preceding code, where a digital clock is rendered by including the `<ts-clock>` custom element:

The screenshot shows the Chrome Developer Tools interface with the 'Elements' tab selected. At the top, the URL is 'localhost:8080/Chapter5Code/digital-clock/clockDemo.html'. Below the URL, the page content displays a digital clock with the time '20 47 19' in large red digits. The 'Elements' tab shows the DOM structure:

```
<body>
  <ts-clock hour="20" minute="47" second="19">
    #shadow-root
      <style>...</style>
      <div class="clock">
        <div class="hour">20</div>
        <div class="minute">47</div>
        <div class="second">19</div>
      </div>
    </ts-clock>
    <script>...</script>
  </body>
```

The 'html' tab is also visible at the bottom left.

In the preceding screenshot, we can see that the `hour`, `minute`, and `second` attributes are created because the `accessors` properties are set to `{}`. The following screenshot shows the Chrome developer console with the log messages printed by the `getAttribute` method for getting the values of `hours`, `minutes`, and `seconds`:

localhost:8080/Chapter5Code/digital-clock/clockDemo.html



20      53      11

Elements Network Sources Timeline Profiles

✖️  <top frame> ▼  Preserve log

Hour : 20  
Minute : 53  
Second : 11

# Summary

In this chapter, we learned about the Mozilla Brick library for web application development using web component specification. In the next chapter, we will learn about the ReactJS framework.

# Chapter 6. Building Web Components with ReactJS

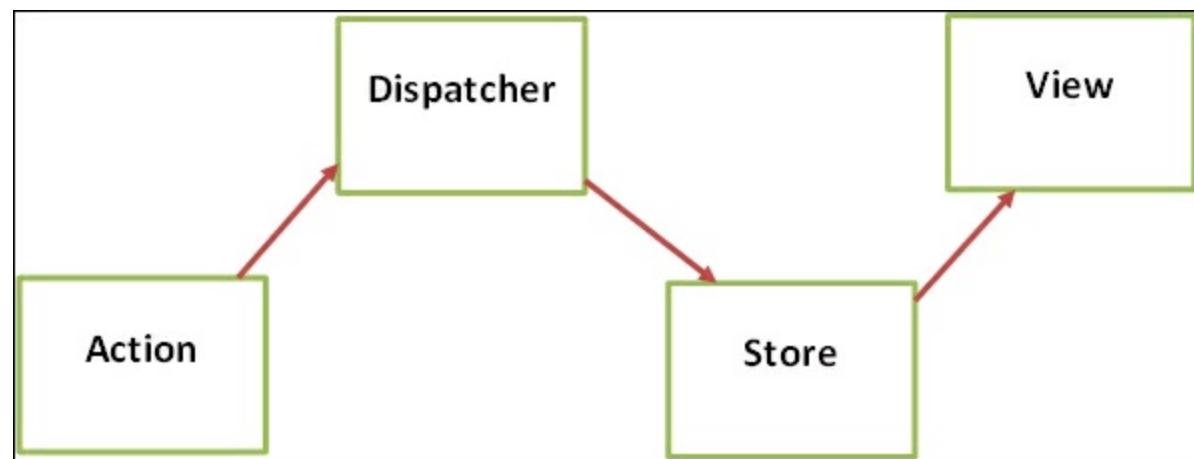
In this chapter, we will learn about the "react way" for web component development offered by the Facebook team. We will also learn to develop a web component using the ReactJS library. We will explore the flux architecture of a web application and understand how the data flows in it. Finally, we will develop a custom component using the ReactJS library.

## The reactive approach

Before understanding the ReactJS framework in detail, we should know some of the applications that have been developed. The comment section of Facebook, LiveFyre, and Disqus are developed using ReactJS. These applications are developed by following the flux architecture.

# The flux architecture

A flux application has three important parts—**dispatcher**, **store**, and **views**. These parts are connected by **action**. The following diagram shows the three building blocks and action data flow in the flux architecture:



The details of these three parts of flux architecture are as follows:

- **View**: This represents the UI components that are rendered in the browser to be used by the end users.
- **Store**: This holds the application data and business logic. It is similar to the model of the MVC framework. **Store** represents a collection of data objects, whereas model represents a single object.
- **Dispatcher**: This is the central hub through which all actions have to pass. It contains all the callback registered by the stores.

Applications developed in flux architecture work as follows:

1. Generally, actions are generated by the user, interacting with the views. These actions are asynchronous in nature.
2. These actions are then passed to the dispatcher for processing. The dispatcher then invokes the callback method registered in the store.
3. The execution of the callback method may change the state of the data. The store then emits the change event with the updated data. The view listens to the change event and accesses the modified data from the store using event handler. The view then calls their own `setState` method, which causes the re-rendering of itself and its children element.

4. The data flow is unidirectional. Flux architecture does not allow two-way data binding, which can cause more cascading updates.

### Tip

The data flow in flux-based application is unidirectional.

## Flux key features

The flux architecture provides some key features that a flux-based web application must follow. These key features are as follows:

- **Synchrony:** All the callback methods registered for each action are synchronous in execution, but the action is triggered asynchronously from the source.
- **Inversion of control:** The flow of control is transferred to the appropriate store object and targeted callback.
- **Semantic actions:** The action triggered from the source contains some semantic information, which helps the store object decide the appropriate method execution.
- **No cascading actions:** Flux disallows cascading actions that generate multiple updates.

### Note

To find out more about flux architecture use the following link:

<http://fluxxor.com/what-is-flux.html>

# Installing ReactJS

The ReactJS library can be installed using Bower. Assuming that Bower is already present in your machine, use the following command to install the ReactJS library:

```
bower install react
```

The following screenshot shows the terminal with ReactJS installation in progress using Bower:



A screenshot of a terminal window titled "Terminal". The command "bower install react" is being run. The output shows the process: "bower cached" for the git repository, "bower validate" for version 0.12.2, and "bower install" for the react component itself. The final result is "react#0.12.2 bower\_components\react".

```
+ J:\Chapter6Code>bower install react
X bower cached      git://github.com/facebook/react-bower
  .git#0.12.2
  bower validate      0.12.2 against git://github.com/facebook/react-bower.git#*
  bower install      react#0.12.2

  react#0.12.2 bower_components\react
```

After successful execution of the Bower command, the system loads the ReactJS library file inside the `react` subdirectory present inside the `bower_components` directory. The following screenshot shows the files present inside the ReactJS library:

```
+ J:\Chapter6Code>cd bower_components
x
J:\Chapter6Code\bower_components>cd react

J:\Chapter6Code\bower_components\react>ls
JSXTransformer.js  react-with-addons.js
LICENSE           react-with-addons.min.js
PATENTS          react.js
bower.json        react.min.js

J:\Chapter6Code\bower_components\react>
```

The ReactJS library has three different parts:

- **ReactJS core library:** This is the core ReactJS library. The name of this file is `ReactJS` and the minified version of this library is `react.min.js`.
- **ReactJS addon:** This represents the additional utility packages that can be used with ReactJS. The name of this file is `react-with-addons.js` and the minified version of this library is `react-with-addons.min.js`.
- **JSX transformer library:** This library can be used in the browser to transfer the JSX code to JavaScript. The name of this file is `JSXTransformer.js`.

# Configuring ReactJS

The ReactJS library can be used as a standalone or with the JSX transformer file. We will find out more about JSX in the coming section. For now, we can include the following `<script>` tag to start using the ReactJS library:

```
<script src="bower_components/react/react.js"></script>
<script src="bower_components/react/JSXTransformer.js"></script>
```

The ReactJS library can also be used directly from the Facebook CDN to increase the performance of the page load. The `<script>` code to load the ReactJS library from the CDN is as follows:

```
<script src="http://fb.me/react-0.12.2.js"></script>
<script src="http://fb.me/JSXTransformer-0.12.2.js"></script>
```

## Note

During the writing of this book, the ReactJS library has the 0.12.2 version.

# Using ReactJS

We can use the ReactJS library by including the `ReactJS` or `react.min.js` file using the `<script>` element. We can create a DOM element using the `React.createElement` method. The syntax for the `createElement` method is as follows:

```
var reactElement = React.createElement(type, properties, children);
```

The details of the preceding syntax are as follows:

- `type`: This represents the HTML element name in a string format
- `properties`: This represents an object with the key-value pair for attribute name and values
- `children`: This represents the child nodes that an element can have

A React element can be rendered in the browser using the `React.render` method. The syntax for the `render` method is as follows:

```
React.render(reactElement, targetDOMNode)
```

The details of the preceding syntax are as follows:

- `reactElement`: It represents the ReactJS element which need to be rendered in the browser.
- `targetDOMNode`: It represent the DOM node where the ReactJS element needs to be appended for displaying in browser.

Now, let's checkout an example for displaying the `React` element using the `createElement` and `render` method. The following code shows the use of these methods:

```
<!DOCTYPE html>
<html>
<head lang="en">
  <meta charset="UTF-8">
  <title>ReactJS Without JSX Demo</title>
  <script src="bower_components/react/react.js"></script>
  <style>
    .headerStyle{
      color: red;
    }
  </style>
```

```
</style>
</head>
<body>
  <script>
    var helloMessage = React.createElement('h1',
      { className: 'headerStyle' }, 'Hello ReactJS');
    React.render(helloMessage, document.body);
  </script>
</body>
</html>
```

The details of the preceding code are as follows:

- A header element `h1` is created using the `React.createElement` method. This header element has a `headerStyle` class and a child node of type `TEXT` with a `Hello ReactJS` value. Reference of this header element is saved in the `helloMessage` variable.
- The `React.render` method is used to display the header element `helloMessage` in the `document.body` position.

The output of the preceding code is rendered in the following screenshot containing the **Hello ReactJS** string message in the color red:



# What is JSX

ReactJS comes with another flavor named JSX. **JSX** stands for **JavaScript XML**. It follows XML type element declaration for web component development. There are many benefits of using JSX:

- **Familiarity:** Developers are familiar with XML, and JSX provides a similar type of element declaration.
- **Semantics:** JSX is easier to understand as it follows a declarative type of programming.
- **Separation of concern:** JSX provides a clean way to encapsulate all the logic and markup in one definition.

## Custom components with JSX

ReactJS provides the `JSXTransformer.js` script file for transpiling of the JSX code in the browser. A JSX code block is defined using the following syntax:

```
<script type="text/jsx">
    //JSX code goes here
</script>
```

A custom element can be created using the `React.createClass` method. The syntax for the `createClass` method is as follows:

```
React.createClass (objectSpecification)
```

In the preceding syntax, the `objectSpecification` class takes a `render` method and other optional lifecycle methods and properties. We will learn more about the lifecycle method and properties in the coming sections. For now, let's check out an example of developing a custom element using JSX. The following code contains the use of the `createClass` method to develop a custom element:

```
<!DOCTYPE html>
<html>
<head lang="en">
    <meta charset="UTF-8">
    <title>ReactJS With JSX Demo</title>
    <script src="bower_components/react/react.js"></script>
    <script src="bower_components/react/JSXTransformer.js"></script>
    <style>
```

```

.headerStyle{
    color: red;
}
</style>
</head>
<body>
<script type="text/jsx">
    var HelloMessage = React.createClass({
        render: function() {
            return (
                <h1 className="headerStyle">
                    Hello ReactJS
                </h1>
            );
        }
    });
    React.render(<HelloMessage/>, document.body);
</script>
</body>
</html>

```

The details of the preceding code are as follows:

- A custom element named `HelloMessage` is created using the `React.createClass` method inside a `<script>` block with `type` set to `text/jsx`.
- The specification object contains the `render` method, which has a JSX element. The `h1` element has a `className` attribute. The `className` attribute is a JSX attribute, but looks familiar to an HTML element attribute.
- The specification object for `HelloMessage` contains a `render` method which returns a JSX element.
- The `<HelloMessage>` element is then rendered using the `React.render` method inside the `<body>` tag.

The output of the preceding code is rendered as the following screenshot. We can find a similar result that the **Hello ReactJS** message is rendered in red color:

The screenshot shows a browser window with the URL 'localhost:8080/Chapter6Code/react-with-jsx.html'. The main content area displays a large red 'Hello ReactJS' heading. Below the heading, the browser's developer tools are open, specifically the 'Elements' tab. The DOM tree is visible, starting with an '<html>' tag. Inside the 'body' tag, there is an '<h1>' element with the text 'Hello ReactJS'. This element has inline styles applied via the 'style' attribute, which is represented as a JavaScript object: { styleAttribute: "styleValue" }. The 'Elements' tab also shows other browser-specific nodes like '<!DOCTYPE>' and various network and source files.

## ReactJS inline style

In ReactJS, we can add an inline style using the `style` attribute and ReactJS expression. ReactJS takes inline style as a JavaScript *anonymous object* containing a key/value pair representing properties and their values separated with a *colon* (:). The following syntax shows the JavaScript object for inline style:

```
var styleObject={  
    styleAttribute: "styleValue",  
};
```

The details of the preceding syntax are as follows:

- **styleAttribute:** This represents the CSS property name as key. The name should follow **camelCase** representation. For example, the `box-shadow` style attribute becomes `boxShadow`. The vendor prefix attribute starts with a capital letter except `ms`(Microsoft Internet Explorer).
- **styleValue:** This represents a value for the CSS property, and it is in string format. For example, `1px solid grey` should be wrapped in double quotes like "`1px solid grey`".

Let's check out an example for using inline style in a ReactJS element. The code for the ReactJS element is as follows:

```

<!DOCTYPE html>
<html>
<head lang="en">
    <meta charset="UTF-8">
    <title>ReactJS inline style demo</title>
    <script src="bower_components/react/react.js"></script>
    <script src="bower_components/react/JSXTransformer.js"></script>
</head>
<body>
<script type="text/jsx">
    var messageStyle={
        color: "red",
        border: "1px solid grey",
        boxShadow: "2px 2px 2px lightGrey",
        padding: "20px",
        width: "200px"
    },
    GoodMorning = React.createClass({
        render: function() {
            return (
                <div style={messageStyle}>
                    Good Morning Developers
                </div>
            );
        }
    });
    React.render(<GoodMorning/>, document.body);
</script>
</body>
</html>

```

The details of the preceding code are as follows:

- The `messageStyle` object contains an anonymous object containing the CSS attribute's name as `key` and the styles as a string `value`
- This `messageStyle` object is then bound with the `style` attribute of the `div` element using curly braces

The output of the preceding code is shown in the following screenshot, with the `style` attributes applied to the `div` element:

The screenshot shows a browser window with the URL 'localhost:8080/Chapter6Code/react-inline-style.html'. The page content is 'Good Morning Developers' in red text inside a red-bordered box. In the developer tools, the 'Elements' tab is selected. The DOM tree shows:

```
<html>
  <head lang="en">...</head>
  <body>
    <div style="color:red; border:1px solid grey; box-shadow:2px 2px 2px lightGrey; padding:20px; width:200px;" data-reactid=".0">Good Morning Developers</div>
  </body>
</html>
<!DOCTYPE>
```

A red callout bubble with the text 'in-line styles applied to element' points to the 'style' attribute of the div element in the DOM tree.

## ReactJS event handling

The event handling in ReactJS is similar to DOM events, with a difference in naming the handler. For example, an `onclick` handler is renamed to `onClick` in ReactJS. This follows the camelCase syntax to name the handler. Let's check out an example of event handling. The following code shows a simple example of event handling in ReactJS:

```
<!DOCTYPE html>
<html>
<head lang="en">
  <meta charset="UTF-8">
  <title>ReactJS Event Demo</title>
  <script src="bower_components/react/react.js"></script>
  <script src="bower_components/react/JSXTransformer.js"></script>
</head>
<body>
<script type="text/jsx">
  var SayHello = React.createClass({
    helloHandler: function(event) {
      alert("Hello Developers");
    },
  },
```

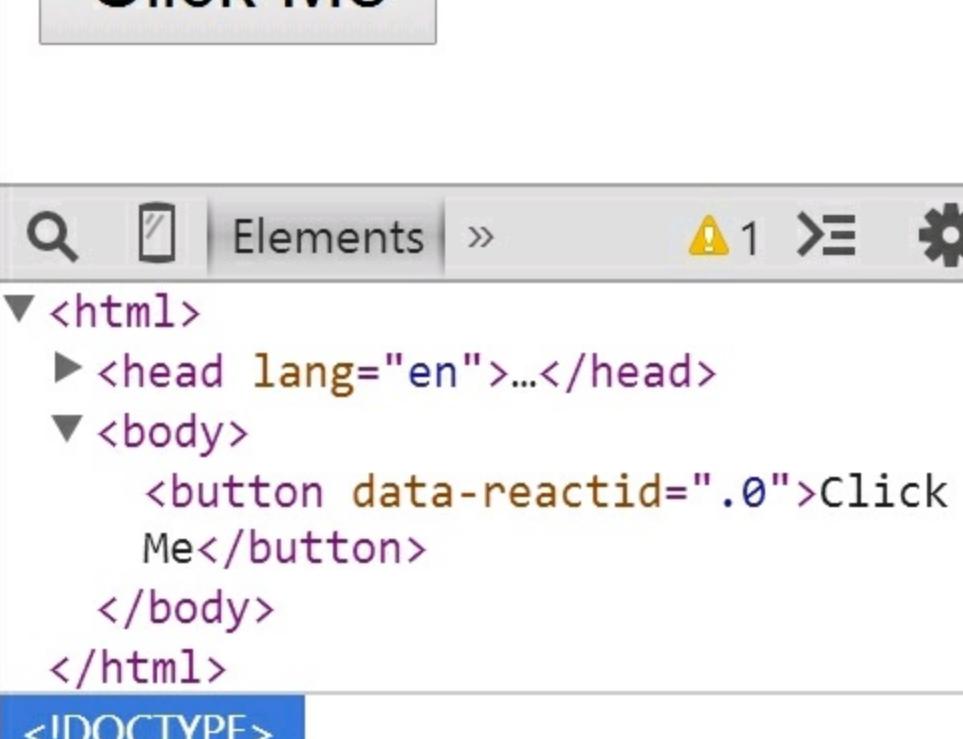
```
render: function() {
  return (
    <button onClick = {this.helloHandler}>
Click Me
</button>
  );
}
);
React.render(<SayHello/>, document.body);
</script>
</body>
</html>
```

The details of the code are listed as follows:

- The `<SayHello>` element is created using the `React.createClass` method. The `render` method contains the `<button>` element with an `onClick` event handler, which is bound to the `helloHandler` method using `this` keyword. The `this` keyword refers to the `<SayHello>` custom element.
- The `helloHandler` method definition is present inside the specification object. This method contains an `alert` method with a string message.

The output of the preceding code contains a button with the **Click Me** text and is rendered as the following screenshot:

Click Me



The screenshot shows the Chrome DevTools Elements tab. At the top, there are icons for search, device, and refresh, followed by "Elements" and a warning icon with "1". Below the tabs is a tree view of the DOM structure:

- ▼ <html>
  - <head lang="en">...
  - ▼ <body>
    - <button data-reactid=".0">Click Me</button>
  - </body>
- </html>

The <button> element is highlighted with a blue background.

When the user clicks on this button, the `helloHandler` method gets called and an alert window appears, containing the **Hello Developers** message. The output of the preceding code is rendered in the following screenshot:

Click Me

The page at localhost:8080 says:

Hello Developers

OK

>≡



```
▼ <html>
  ► <head lang="en">...</head>
  ▼ <body>
    <button data-reactid=".0">Click
      Me</button>
  </body>
</html>
<!DOCTYPE>
```

## Note

To find out more about the event handling system in ReactJS library use the following link: <http://facebook.github.io/react/docs/events.html>

## Useful non-DOM attributes

In this section, we will learn some non-DOM attributes that can be used with the JSX element. The following list contains some of the non-DOM attributes:

- `key`: This is an optional attribute that can be used to uniquely identify each component in the page
- `ref`: This is an optional attribute that can be used to access the child element from outside the `render` method
- `dangerouslySetInnerHTML`: This attribute can be used inside the JSX element to set HTML content inside the component

Let's check out a simple example with the use of these three attributes. The code for this example is as follows:

```
<!DOCTYPE html>
<html>
<head lang="en">
    <meta charset="UTF-8">
    <title> ReactJS NonDOM Attribute Demo </title>
    <script src="bower_components/react/react.js"></script>
    <script src="bower_components/react/JSXTransformer.js"></script>
</head>
<body>
<script type="text/jsx">
    var StudentsReport = React.createClass({
        render: function() {
            var studentDetails = {
                headerHTML: {
                    __html: "<i>Student List</i>"
                },
                subject: "Mathematics",
                list: [
                    {roll:123, name:"Sandeep"}, {roll:124, name:"Surabhi"}
                ]
            };
            return (
                <div>
                    <h1 dangerouslySetInnerHTML={studentDetails.headerHTML}>
                    </h1>
                    <h3 ref="subjectName">{studentDetails.subject}</h3>
                    <ol>
                        {studentDetails.list.map(function(student) {
                            return <li key={student.roll}>{student.name}</li>;
                        })}
                    </ol>
                    <button onClick={this.logSubject}>Log Subject</button>
                </div>
            );
        },
        logSubject: function(event) {
            console.log("Subject React Element: ", this.refs.subjectName);
            console.log("Subject DOM Element: ", this.refs.subjectName.getDOMNode());
        }
    });
    React.render(<StudentsReport />, document.body);
</script>
</body>
</html>
```

The details of the preceding code are as follows:

- The `<StudentsReport>` element contains the `render` method, which has the `studentDetails` object containing `headerHTML`, `subject`, and `list` properties.
- The `headerHTML` properties contain another object with the `_html` property, which has HTML content wrapped in an `<i>` element. The `headerHTML` property is used with the `dangerouslySetInnerHTML` attribute of a React element present inside the `render` block. The `dangerouslySetInnerHTML` attribute is used to bind the HTML content in runtime. The code to bind HTML is as follows:

```
<h1 dangerouslySetInnerHTML={studentDetails.headerHTML}>
</h1>
```

- The `list` property contains an array of student objects with the `roll` and `name` properties. A `map` function is used to iterate over the `list` array. While iterating `<li>` elements, the `key` attribute takes its value from `roll` property and the inner text takes its value from `name` property. The `key` attribute is used for uniquely identifying the `<li>` student element. The code to bind student details is as follows:

```
<ol>
  {studentDetails.list.map(function(student) {
    return <li key={student.roll}>{student.name}</li>;
  })}
</ol>
```

- The `subject` property contains the name of the course taken by all the students. In the `render` block, `subject` is bound with the `h1` element and with a `ref` attribute named `subjectName`. The `render` block also contains the `<button>` element which is attached with an `onClick` handler referring to the `logSubject` method. The `logSubject` method does the following two things:

- It retrieves the entire referred element using the `this.refs` property and finds `studentName` and prints it in the console. It returns a ReactJS element. The code to access `studentName` is as follows:

```
this.refs.subjectName
```

- It retrieves the DOM version of the React element using the `getDOMNode` method. The code to access the DOM version of `studentName` is as follows:

```
this.refs.subjectName.getDOMNode()
```

The output of the preceding code is rendered as the following screenshot, where

the `dangerouslySetInnerHTML` attribute renders the HTML content:

The screenshot shows the Chrome developer tools' Elements tab with the DOM tree for a "Student List" component. The tree includes a root `div` with `data-reactid=".0"`, containing an `h1` with `data-reactid=".0.0"` and inner text "Student List", an `h3` with `data-reactid=".0.1"` and inner text "Mathematics", an `ol` with `data-reactid=".0.2"` containing two `li` items with `data-reactid=".0.2.$123"` and `".0.2.$124"` respectively, and a `button` with `data-reactid=".0.3"` and inner text "Log Subject". A red callout points to the `dangerouslySetInnerHTML` attribute in the `h1` element, stating "dangerouslySetInnerHTML attribute renders the HTML content". Another red callout points to the `data-reactid` attribute in the `ol` element, stating "Student's roll used as key attribute in data-reactid attribute". The bottom navigation bar shows the path: html > body > div.

When the user clicks on the **Log Subject** button, it prints the ReactJS and DOM element for the `this.refs.subjectName` value in the console. The following screenshot shows the Chrome developer console with two printed messages:

1. Sandeep
2. Surabhi

Log Subject

Elements Network

this.refs.subjectName  
returns ReactJS element

<top frame> ▼ Pr Dev log

Subject React Element: Inline JSX script:29

► ReactDOMComponent {\_tag: "h3", tagName: "H3", props: Object, \_owner: Constructor, \_LifeCycleState: "MOUNTED"}

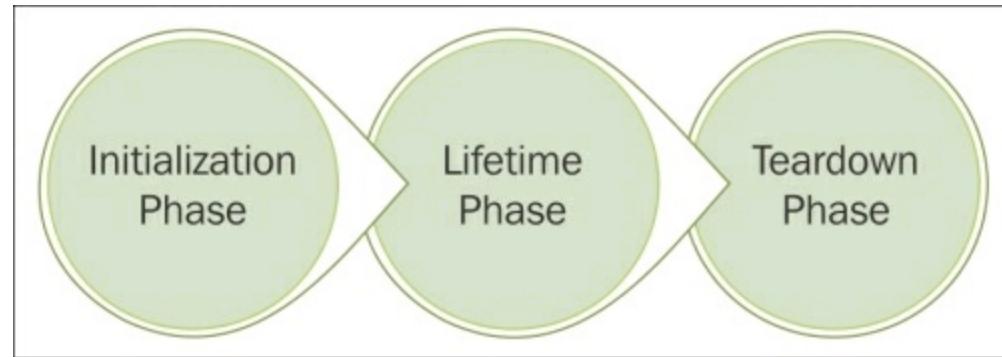
getDOMNode() returns a DOM element

Subject DOM Element:

<h3 data-reactid=".0.1">Mathematics</h3>

# ReactJS component lifecycle

Every object in the word has a lifecycle and passes through different states during its lifetime. We can categorize these states into three different phases. The following diagram shows these three common phases that an element goes through in its lifetime:



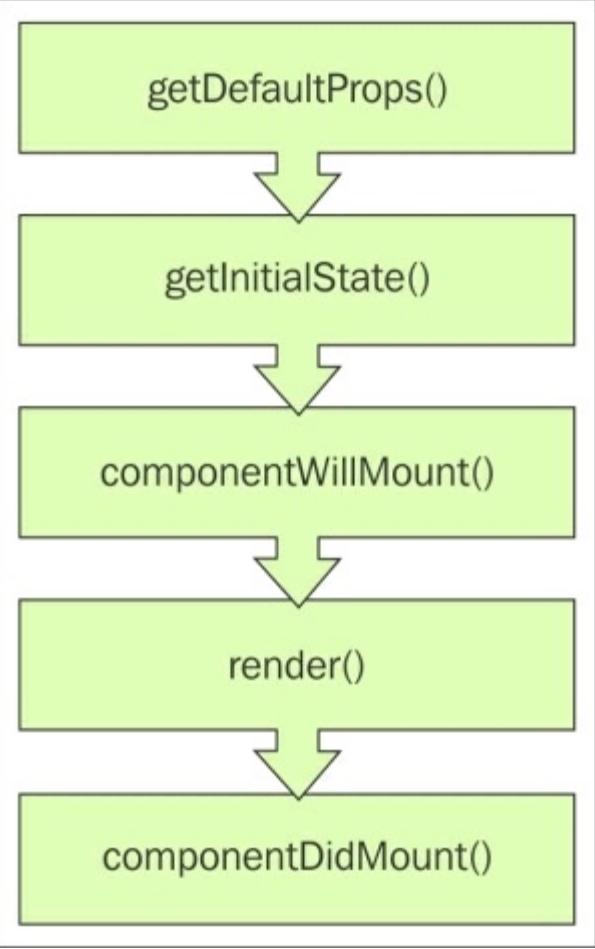
The details of these phases are listed as follows:

- **Initialization:** This is the phase when the instance for the element is created for the first time. Subsequent instances during the lifetime have slight changes.
- **Lifetime:** This is the phase when an element lives and changes its states and properties.
- **Teardown:** This is the phase when an element finishes its execution and cleanup takes place.

Similarly, each ReactJS element has a lifecycle and has different callback methods to handle each state. We will now learn how the ReactJS element goes through each of these phases.

## ReactJS initialization phase

In this phase, a ReactJS element instance is created for the first time and rendered in the browser. ReactJS provides a set of methods for the setup and preprocessing during this phase. During initialization of an element, the methods are called in a specific order. The following diagram shows the order in which the callback methods are called during the initialization phase:



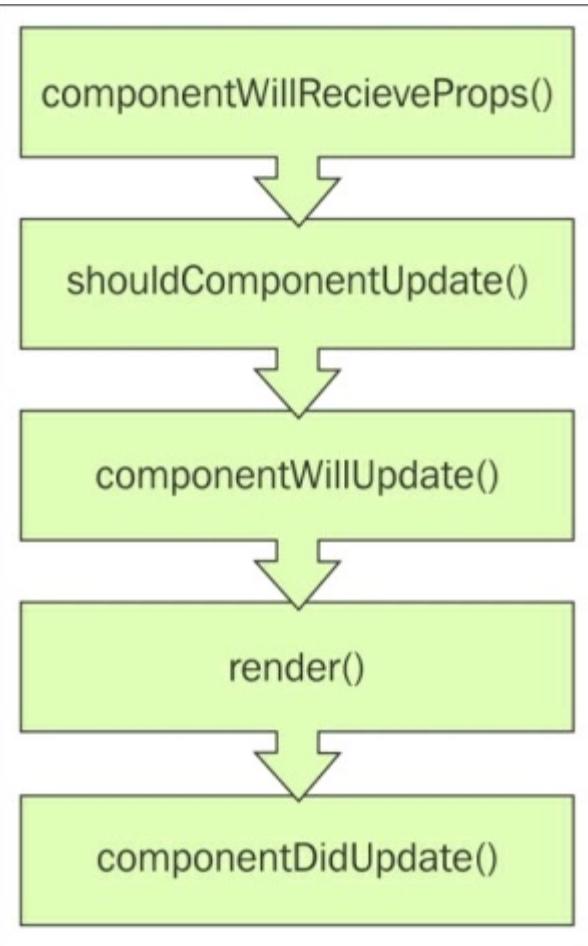
The details of these methods are as follows:

- `getDefaultProps`: This method is used to create default properties for a ReactJS element. This method must return an object or NULL.
- `getInitialState`: This method is used to create states for the component. This method must return an object or NULL.
- `componentWillMount`: This method gets executed just before the component is mounted to the page.
- `render`: This method returns the ReactJS component tree for rendering in the browser.
- `componentDidMount`: This method gets executed just after the initial rendering of the component in the browser.

## ReactJS lifetime phase

Once the ReactJS component is instantiated successfully, the lifetime phase starts. In this phase, the component can go through many changes. These changes include a change in the current state, or a change in any property value. ReactJS provides a

set of callback methods for overriding and implementing our own logic for these states .The following diagram shows the order in which the callback methods are called during lifetime phase:

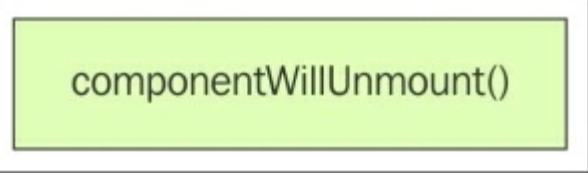


The details of these methods are as follows:

- `componentWillReceiveProps`: This method gets executed whenever a new property is created for the component.
- `shouldComponentUpdate`: This method returns a Boolean value `true` or `false`. The `true` flag indicates that the `render` method will be called when a new property is received.
- `componentWillUpdate`: This method get executed just before the new property is received by the component. It provides an opportunity for preprocessing before the `render` method is called.
- `render`: This method returns the updated component tree for rendering in the browser.
- `componentDidUpdate`: This method gets executed just after the modified changes are rendered in the browser.

# ReactJS teardown phase

This phase is the last state of a ReactJS component. The instance of the component is removed and memory cleanups take place. The following diagram shows the callback method that gets executed during this period:



```
componentWillUnmount()
```

The `componentWillUnmount` method gets executed immediately before a component is unmounted from the DOM.

## ReactJS lifecycle example

In this section, we will develop a simple ReactJS component to demonstrate the order of the lifecycle methods in which they are executed. The following code contains the definition of the `<Welcome>` element with all of its lifecycle callback method:

```
<script type="text/jsx">
  var Welcome = React.createClass({
    get defaultProps: function() {
      console.log("inside defaultProps method");
      return{
        myName: "Sandeep"
      };
    },
    getInitialState: function() {
      console.log("inside getInitialState method");
      return null;
    },
    componentWillMount: function() {
      console.log("inside componentWillMount method");
    },
    componentDidMount: function() {
      console.log("inside componentDidMount method");
    },
    render: function() {
      console.log("inside render method");
      return (
        <div>
          <h1>Welcome {this.props.myName}</h1>
          <button onClick={this.changeName}>

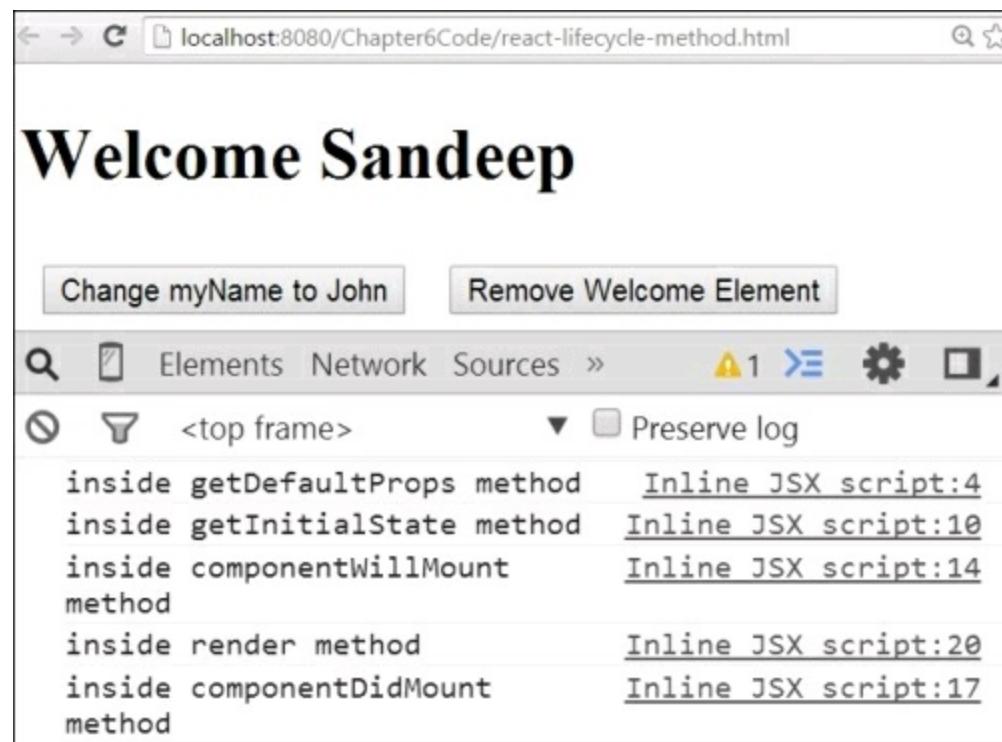
```

```

        Change myName to John
    </button>
    <button onClick={this.removeHandler}>
        Remove Welcome Element
    </button>
</div>
);
},componentWillReceiveProps:function(){
    console.log("inside componentWillReceiveProps method");
},shouldComponentUpdate:function(){
    console.log("inside shouldComponentUpdate method");
    return true;
},componentWillUpdate:function(){
    console.log("inside componentWillUpdate method");
},componentDidUpdate:function(){
    console.log("inside componentDidUpdate method");
},componentWillUnmount:function(){
    console.log("inside componentWillUnmount method");
},changeName:function(){
    this.setProps({myName :"John"})
},removeHandler: function(){
    var thisElement = this.getDOMNode();
    this.unmountComponent(thisElement);
}
});React.render(<Welcome/>, document.body);
</script>

```

The following screenshot shows the output of the `<Welcome>` element when it is initialized and rendered for the first time:



When the user presses the first button to change the `myName` property to `John`, the lifetime callback method gets executed. The following screenshot shows the output of the `<Welcome>` element for the change in the `myName` property:

The screenshot shows a browser window with the URL `localhost:8080/Chapter6Code/react-lifecycle-method.html`. The main content area displays the text **Welcome John**. Below the content are two buttons: **Change myName to John** and **Remove Welcome Element**. The browser's developer tools are open, specifically the **Elements** tab. In the bottom right corner of the developer tools, there is a log icon with a magnifying glass and a star, followed by the text `<top frame>`. To the right of this, there is a dropdown menu set to `Preserve log`. The log area contains the following entries:

Method	Location
inside <code>componentWillReceiveProps</code> method	<a href="#">Inline JSX script:34</a>
inside <code>shouldComponentUpdate</code> method	<a href="#">Inline JSX script:37</a>
inside <code>componentWillUpdate</code> method	<a href="#">Inline JSX script:41</a>
inside <code>render</code> method	<a href="#">Inline JSX script:20</a>
inside <code>componentDidUpdate</code> method	<a href="#">Inline JSX script:44</a>

When the user presses the second button to remove the `<Welcome>` element, the method of the teardown phase gets called. The following screenshot shows the output of the preceding code for the `<Welcome>` element, when it is removed from the DOM:

The screenshot shows a browser window with the same URL as the previous screenshot. The developer tools are open, and the log area shows a single entry:

Method	Location
inside <code>componentWillUnmount</code> method	<a href="#">Inline JSX script:47</a>

## Note

To find out more about the ReactJS component lifecycle methods, use the

following link:

<http://facebook.github.io/react/docs/component-specs.html>

# Stateful custom component

ReactJS provides features to create and programmatically update the state during the creation of a ReactJS element. This increases the interaction of the application. The states are private and mutable to the component and referred to using `this.state`. The React component provides the `this.setState` method to update the values of the state.

Let's check out an example of using the state in the ReactJS element. The following code contains the definition of the `<StudentResult>` element:

```
<script type="text/jsx">
  var StudentResult = React.createClass({
    getInitialState: function() {
      return{
        result: " "
      };
    },
    calculateResult: function() {
      var refScore = this.refs.studentScore.getDOMNode().value,
          score = parseInt(refScore,10);
      if(score > 30){
        this.setState({result: "PASS"})
      }else if(score < 30){
        this.setState({result: "FAIL"})
      }else{
        this.setState({result:""})
      }
    },
    render: function() {
      return (
        <div className="score-container">
          <input type="number" ref="studentScore"
            onChange={this.calculateResult}
            placeholder="Enter student score"/>
          <h2>Result :{this.state.result}</h2>
        </div>
      );
    }
  });
  React.render(<StudentResult/>, document.body);
</script>
```

The details of the code are as follows:

- The `<studentResult>` element has an initial state name `result` with a value empty string.
- The `render` method contains an `<input>` type `number` to enter a student score. This element has the `ref` attribute with `studentScore` for accessing it inside the component. A callback method `calculateResult` is bound with the `onChange` event.
- The `calculateResult` method gets the entered score by the user and decides **PASS** or **FAIL**. If the score is more than 30, the result is shown as **PASS**. If it is less than 30, the result is classes as a `FAIL` or else an empty string.
- The `render` method displays the result using the `{this.state.result}` expression.

The following screenshot shows the output of the `<studentResult>` component:

The screenshot shows a browser window with the URL `localhost:8080/Chapter6Code/react-stateful-component.html`. The page content includes a text input field with the placeholder "Enter student score" and a large heading "Result :". Below the heading, there is some explanatory text. The browser's developer tools are open, specifically the Elements tab of the DevTools panel. The DOM tree is visible, showing the following structure:

```

<!DOCTYPE>
<body>
  <div data-reactid=".0">
    <input data-reactid=".0.0" type="number" placeholder="Enter student score"/>
    <h2 data-reactid=".0.1">
      <span data-reactid=".0.1.0">Result :</span>
      <span data-reactid=".0.1.1"></span>
    </h2>
  </div>
</body>

```

The DevTools also show the React component hierarchy with `data-reactid` attributes assigned to each node.

When the user enters a score the change handler gets executed, and **PASS** or **FAIL** gets printed in the browser. The following screenshot shows the output of the `<StudentResult>` component when the user enters `45` in the `<input>` element:

The screenshot shows a browser window with the URL `localhost:8080/Chapter6Code/react-stateful-component.html`. In the main content area, the text "Result :PASS" is displayed. Above it, there is an input field containing the number "45". Below the main content, the browser's developer tools are open, specifically the Elements tab. The DOM tree is visible, showing the following structure:

```
<!DOCTYPE>
<body>
  <div data-reactid=".0">
    <input type="number" placeholder="Enter student score" data-reactid=".0.0">
    <h2 data-reactid=".0.1">
      <span data-reactid=".0.1.0">Result :</span>
      <span data-reactid=".0.1.1">PASS</span>
    </h2>
  </div>
</body>
```

The input field has a `data-reactid=".0.0"` attribute. The `<h2>` element has a `data-reactid=".0.1"` attribute. Inside the `<h2>` element, there are two `<span>` elements with `data-reactid=".0.1.0"` and `.0.1.1"` attributes respectively, containing the text "Result :" and "PASS".

# Precompiled JSX for production

In ReactJS, the `JSXTransformer.js` file provides the in-browser JSX compilation to vanilla JavaScript format. The in-browser compilation is not recommended for production due to the following reasons:

- **Additional computation:** It slows down the rendering speed due to the overhead of JSX compilation on the client's side at runtime.
- **File size:** The `JSXTransformer.js` file size is big. This increases the additional load to the client side.

ReactJS comes with a precompiled JSX process to resolve the in-browser compilation issue. Using this approach, the developer can compile the JSX to vanilla JavaScript in offline mode. We can achieve this offline JSX compilation by installing the `react-tool` with the npm package. Assuming that npm is present in the system, use the following command to install `react-tool` in the system:

```
npm install -g react-tools
```

The following screenshot shows the terminal with the `react-tool` package installation in progress:

The screenshot shows a terminal window with the title "Terminal". The command `npm install -g react-tools` is being run. The output shows the package being installed from the npm registry and its dependencies being resolved. The terminal window has a light gray background and a dark gray header bar. The scroll bar on the right side of the terminal window is visible.

```
+ J:\Chapter6Code>npm install -g react-tools
× C:\Users\Sandeep\AppData\Roaming\npm\jsx -> C:\Users\Sandeep\AppData\Roaming\npm\node_modules\react-tools\bin\jsx
react-tools@0.12.2 C:\Users\Sandeep\AppData\Roaming\npm\node_modules\react-tools
  └── jstransform@8.2.0 (base62@0.1.1, esprima-fb@8001.1001.0-dev-harmony-fb, source-map@0.1.31)
    └── commoner@0.10.1 (commander@2.5.1, private@0.1.6, install@0.1.8, q@1.1.2, graceful-fs@3.0.5, iconv-lite@0.4.6, mkdirp@0.5.0, glob@4.2.2, recast@0.9.17)

J:\Chapter6Code>
```

To demonstrate the JSX precompilation, we created two subdirectories `dev` and `production` under the `precompile` directory. A new ReactJS component `<ShowDate>` is defined inside the `react-date.js` file under the `dev` directory. The following screenshot shows the terminal with new directory structure:

```
Terminal
+ J:\Chapter6Code>cd precompile
x
J:\Chapter6Code\precompile>ls
dev  production

J:\Chapter6Code\precompile>cd dev

J:\Chapter6Code\precompile\dev>ls
react-date.js
```

The JSX codes for the `<ShowDate>` element in the `react-date.js` file are as follows:

```
var ShowDate = React.createClass({
  get defaultProps: function() {
    return{
      today: new Date().toString()
    };
  },
  render: function() {
    return (
      <h1>
        Today: {this.props.today}
      </h1>
    );
  }
});
React.render(<ShowDate/>, document.body);
```

The details of the preceding code are as follows:

- The `<ShowDate>` element definition contains the `get defaultProps` method that returns an object with the key name as `today` and the value as `new Date().toString()`

- The <ShowDate> element definition contains the render method returning an h1 element with the this.props.today expression for display

Now, we can precompile the JSX code present in the dev/react-date.js file to the vanilla JavaScript. The command for JSX compilation to vanilla JavaScript is as follows:

```
jsx dev production --no-cache-dir
```

The following screenshot shows a terminal with JSX precompilation, which converts the react-date.js file to vanilla JavaScript and moves it to the production directory:

```
+ J:\Chapter6Code\precompile>jsx dev production --no-cache-dir
built Module("react-date")
["react-date"]

J:\Chapter6Code\precompile>cd production

J:\Chapter6Code\precompile\production>ls
react-date.js

J:\Chapter6Code\precompile\production>█
```

The compiled vanilla JavaScript code produced by JSX precompiler is as follows:

```
var ShowDate = React.createClass({
  displayName: "ShowDate",
  getDefaultProps: function() {
    return{
      today: new Date().toString()
    };
  },
  render: function() {
    return (
      React.createElement("h1", null,
        "Today: ", this.props.today
      )
    );
  }
});
```

```
});  
React.render(React.createElement(ShowDate, null), document.body);
```

The details of the preceding code are as follows:

- It contains a `displayName` property, which has the string value `ShowDate` as the name of the ReactJS element
- The `render` method creates an `h1` element using the `React.createElement` method, the text content `Today` and the `this.props.today` value

The vanilla JavaScript code present inside the `production/react-date.js` file is precompiled and can be used directly in an HTML page without the in-browser compilation provided by the `JSXTransformer.js` file. The code for using the `<ShowDate>` element is as follows:

```
<!DOCTYPE html>  
<html>  
<head lang="en">  
  <meta charset="UTF-8">  
  <title>ReactJS precompilation demo</title>  
  <script src="../bower_components/react/react.js"></script>  
</head>  
<body>  
<script src="production/react-date.js"></script>  
</body>  
</html>
```

The following screenshot shows the output of the preceding code, where the `<ShowDate>` element is displaying the current date:

# Today: Fri Feb 06 2015

The screenshot shows the Chrome DevTools Elements tab. At the top, there are icons for search, copy, and refresh, followed by tabs for Elements, Network, Sources, Timeline, Profiles, and a more options menu. Below the tabs, the DOM tree is displayed. It starts with an <html> element, which contains a <head> element with a lang attribute set to "en". The <body> element contains an <h1> element with a data-reactid attribute set to ".0". Inside the <h1> element are two <span> elements: one with data-reactid=".0.0" containing the text "Today:", and another with data-reactid=".0.1" containing the text "Fri Feb 06 2015". The <h1> and <body> elements have closing tags. At the bottom of the tree, there is a <!DOCTYPE> declaration.

## JSX file watcher

The `react-tools` package comes with a file watcher, which can be used to observe the code changes in the JSX file and automatically generates the vanilla JavaScript code. The following command is used to startup the file watcher:

```
jsx --watch dev production
```

The following screenshot shows the terminal with the JSX file watcher, which observes the `dev` directory for any JSX code change in the `react-date.js` file. Any changes in the JSX code can be logged by the file watcher in the terminal:

+

✗ J:\Chapter6Code\precompile>jsx --watch dev production  
built Module("react-date")  
["react-date"]  
react-date.js\_\_jb\_bak\_\_ changed; rebuilding...  
react-date.js changed; rebuilding...  
react-date.js\_\_jb\_old\_\_ changed; rebuilding...  
built Module("react-date")  
["react-date"]  
["react-date"]

# Developing a digital clock using ReactJS

In this section, we will develop a digital clock element <TsClock> using the ReactJS library. The steps to develop a digital clock are as follows:

## Step1 – defining the digital clock lifecycle script

The following code contains the lifecycle callback method definition for a <TsClock> digital clock element developed in the ReactJS library:

```
<script type="text/jsx">
    var TsClock = React.createClass({
        get defaultProps: function() {
            return{
                hour: "HH",
                minute: "MM",
                second: "SS"
            };
        },
        render: function() {
            return (
                <div className="clock">
                    <div className="hour">
{this.props.hour}
                    </div>
                    <div className="minute">
{this.props.minute}
                    </div>
                    <div className="second">
{this.props.second}
                    </div>
                );
        },
        updateClock: function() {
            var clock = new Date();
            this.setProps({
                hour : clock.getHours(),
                minute: clock.getMinutes(),
                second: clock.getSeconds()
            });
        },
    },
```

```

componentDidMount: function() {
    window.setInterval(this.updateClock, 1000);
}
});
React.render(<TsClock/>, document.body);
</script>

```

The details of the preceding code are as follows:

- This block defines the `<TsClock>` element using the `React.createElement` method.
- It has three default properties named `hour`, `minute`, and `second` with values `HH`, `MM`, and `SS` inside the `get defaultProps` method.
- The `render` method returns the ReactJS component tree with the digital clock template. The template displays the values of `hour`, `minute`, and `second` inside a ReactJS expression `{}` using the `this.props` object.
- The `componentDidMount` lifecycle callback method gets fired when the `<TsClock>` element is mounted on the DOM for the first time. This method has the `setInterval` method bound with the `updateClock` method, with a duration of 1 second.
- The `updateClock` method creates a new date object and modifies the value of `hour`, `minute`, and `second` properties by calling the `setProps` method provided by ReactJS. After updating the value of `hour`, `minute`, and `second`, the `setProps` method calls the `render` method in the background. This updates the DOM in the browser displaying the latest time.

## Step2 – defining CSS styles for the digital clock

The following CSS code is used to apply styles to a digital clock:

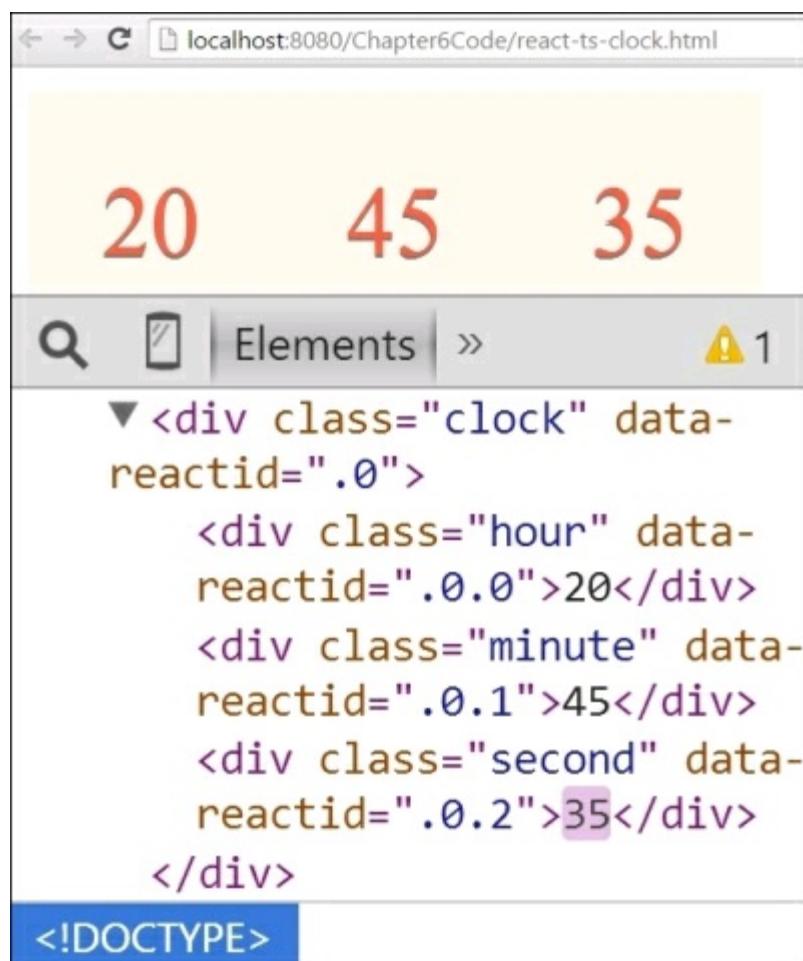
```

<style>
    .clock {
        display: inline-flex;
        justify-content: space-around;
        background: floralwhite;
        font-size: 2rem;
        font-family: serif;
    }
    .clock .hour,
    .clock .minute,
    .clock .second {
        color: tomato;
        padding: 1.5rem;
    }

```

```
        text-shadow: 0px 1px grey;  
    }  
</style>
```

The following screenshot shows the output of the preceding code with the digital clock containing hour, minute, and second properties:

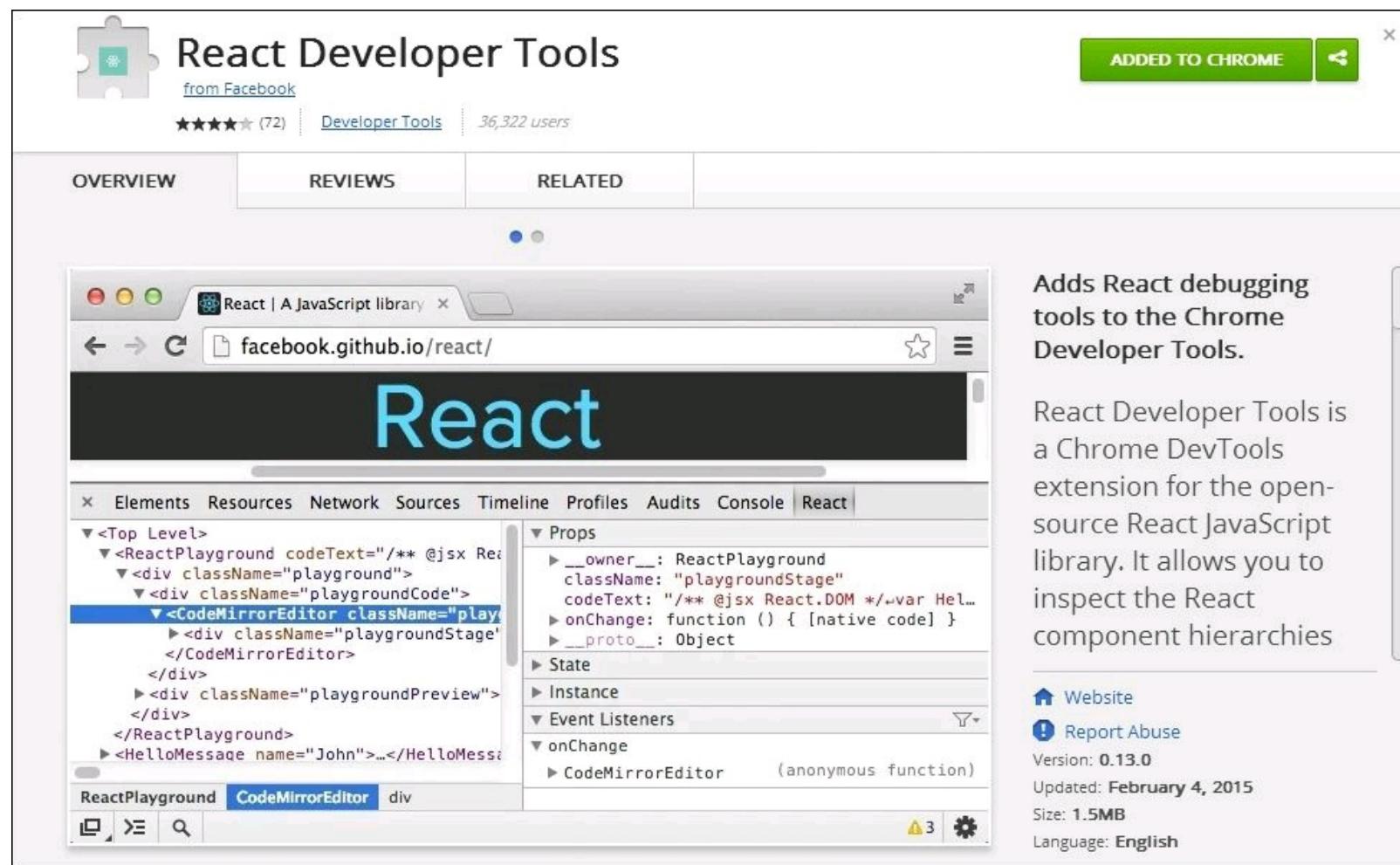


# Debugging ReactJS

ReactJS provides a debugging tool for the Chrome browser. It can be installed to the Chrome browser using the following link:

<https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadoplbjbjfkapdkoienihi>

The following screenshot shows the Chrome store with the ReactJS debugging tool:

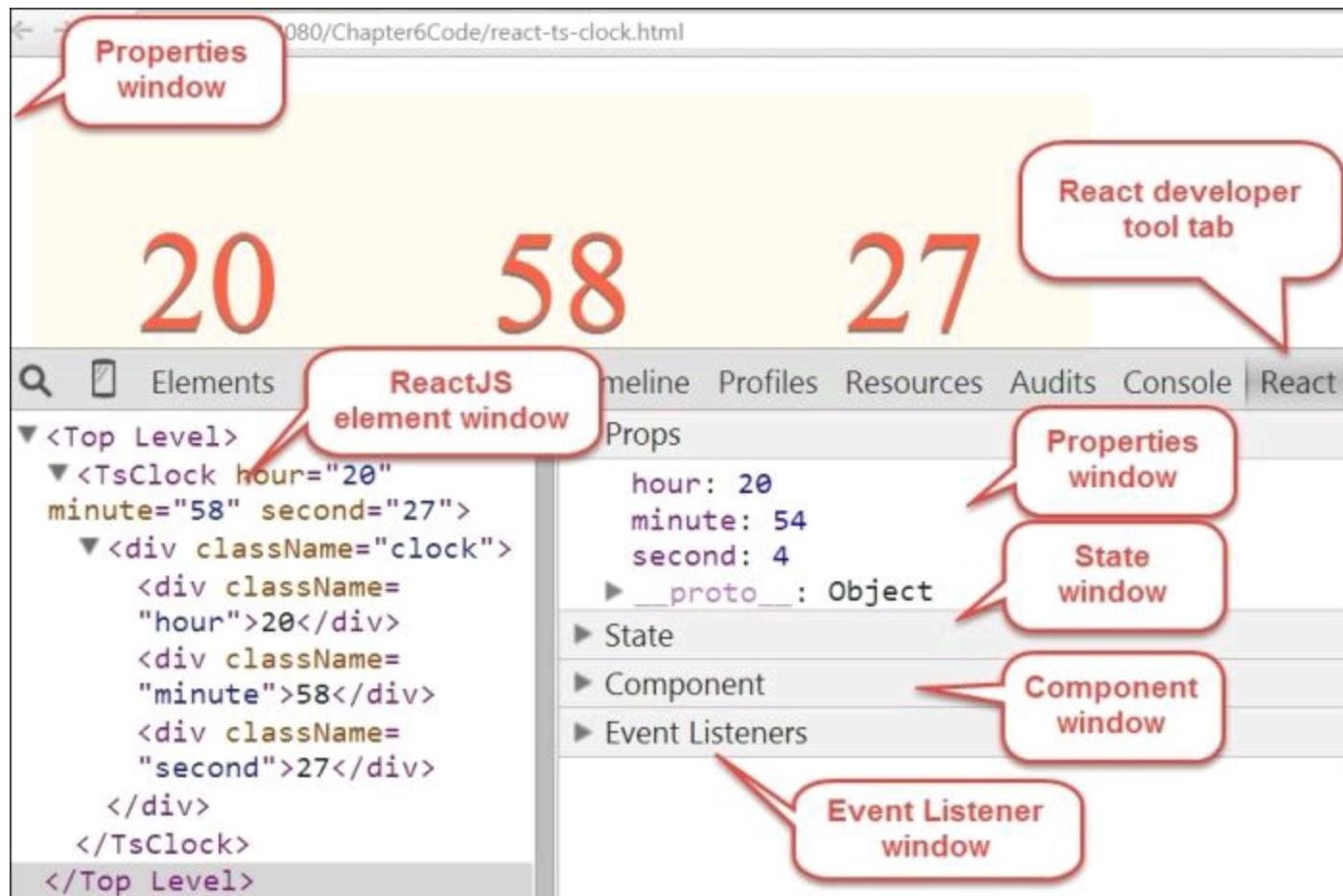


After successful installation of **React Developer Tools**, we can see a new tab name **React** added to the Chrome developer console. The **React Developer Tools** tab contains the following five different windows for debugging:

- **Elements:** This window shows the ReactJS element markup for the rendered page.
- **Props:** This window lists all the properties used by the `React` element.

- **State:** This window lists all the states used by the React element.
- **Component:** This window lists all the properties and methods used by the React element.
- **Event Listeners:** This window lists all the event listeners used by the React element.

The following screenshot shows the React developer tool for the digital clock `<TsClock>` element:



## Note

To find out more about the React developer tool for debugging use the following link:

<http://facebook.github.io/react/blog/2014/01/02/react-chrome-developer-tools.html>

# Summary

In this chapter, we learned about the ReactJS library and its different features, such as JSX, virtual DOM, and custom component development. We understood the flux architecture and how data flows in a ReactJS-based application. With this chapter, we have come to the end of this book. I hope you now understand the web component specification concepts. Happy coding!

# Appendix A. Web Component References

If some of these links do not work, they are usually replaced by a new version. You should locate the latest version as software changes are fairly rapid and common.

# Chapter 1

- W3C web component specification status:  
[http://www.w3.org/standards/techs/components#w3c\\_all](http://www.w3.org/standards/techs/components#w3c_all)
- W3C Shadow DOM specification:  
<http://w3c.github.io/webcomponents/spec/shadow/>
- W3C template specification: <http://www.w3.org/TR/2013/WD-components-intro-20130606/#template-section>
- W3C custom element specification: <http://www.w3.org/TR/2013/WD-components-intro-20130606/#custom-element-section>
- W3C HTML Import specification: <http://www.w3.org/TR/2013/WD-components-intro-20130606/#imports-section>
- Mozilla documentation on web component specification:  
[https://developer.mozilla.org/en-US/docs/Web/Web\\_Components](https://developer.mozilla.org/en-US/docs/Web/Web_Components)
- W3C web component specification repository:  
<https://github.com/w3c/webcomponents>
- Gallery of reusable components: <http://component.kitchen/components>
- *Ten Principles for Great General Purpose Web Components*:  
<https://github.com/basic-web-components/components-dev/wiki/Ten-Principles-for-Great-General-Purpose-Web-Components>
- Web component articles from HTML5 rocks:  
<http://www.html5rocks.com/en/search?q=web+component>
- Web components—the future of web development: <http://html5-demos.appspot.com/static/cds2013/index.html#28>
- Web component gallery: <http://customelements.io/>
- Article on custom element:  
<http://www.smashingmagazine.com/2014/03/04/introduction-to-custom-elements/>
- Article on *Web Components a tectonic shift for web development*:  
<http://www.webcomponentsshift.com/#2>
- Material design specification: <http://www.google.com/design/spec/material-design/introduction.html>

# Chapter 2

- The PolymerJS code repository: <https://github.com/polymer/polymer>
- The CDN link for the PolymerJS library: <http://cdnjs.com/libraries/polymer>
- Tutorials on the PolymerJS library:  
<http://www.tutorialsavvy.com/search/label/polymer>
- Twitter handle for the PolymerJS library: <https://twitter.com/polymer>
- Applications built with the PolymerJS library: <http://builtwithpolymer.org/>
- PolymerJS ready Chrome plugin to detect a web application is built in PolymerJS: <https://chrome.google.com/webstore/detail/polymer-ready/aaifiopbmiecbpladpjaoemohhfjcbdk>

# Chapter 3

- Vulcanize process details: <https://github.com/polymer/vulcanize>
- YouTube videos for the PolymerJS library: <http://www.youtube.com/playlist?list=PLm0Hvc2jKkpAfjR2xuBFUPD4s7n-fg023>
- Stack overflow community group for PolymerJS: <http://stackoverflow.com/questions/tagged/polymer>
- *Building Web Apps With Yeoman And Polymer*: <http://www.html5rocks.com/en/tutorials/webcomponents/yeoman/>
- *Concatenating Web Components with Vulcanize*: [https://www.polymer-project.org/articles\(concatenating-web-components.html](https://www.polymer-project.org/articles(concatenating-web-components.html)
- *Building Web Apps With Yeoman And Polymer*: <http://www.html5rocks.com/en/tutorials/webcomponents/yeoman/>
- Google Chrome development editor for PolymerJS-based application development: <https://chrome.google.com/webstore/detail/chrome-dev-editor-develop/pnoffddplpippgcfdhbmhkofpnalpg?hl=en>

# Chapter 4

- Twitter handle for the Bosonic library: [https://twitter.com/bosonic\\_project](https://twitter.com/bosonic_project)
- Bosonic built-in elements: <http://bosonic.github.io/elements.html>

# Chapter 5

- Mozilla documentation on the Brick library: [https://developer.mozilla.org/en-US/Apps/Tools\\_and\\_frameworks/Web\\_components](https://developer.mozilla.org/en-US/Apps/Tools_and_frameworks/Web_components)
- Twitter handler for Mozilla Brick library: <https://twitter.com/mozbrick>
- X-Tag library documentation: <http://www.x-tags.org/docs>
- TechCrunch article on Mozilla Brick:  
<http://techcrunch.com/2013/09/01/mozilla-gets-on-the-web-components-bandwagon-with-brick/>
- Mozilla Brick online designer tool: <https://apps.webmaker.org/designer>

# Chapter 6

- Stack overflow community group for ReactJS:  
<http://stackoverflow.com/questions/tagged/reactjs>
- Google group for ReactJS library:  
<https://groups.google.com/forum/#!forum/reactjs>
- ReactJS JSFiddle: <http://jsfiddle.net/reactjs/69z2wepo/>
- ReactJS JSFiddle without JSX: <http://jsfiddle.net/reactjs/5vjqabv3/>
- ReactJS starter kit: <http://facebook.github.io/react/downloads/react-0.12.2.zip>
- Tutorials on ReactJS library:  
<http://www.tutorialsvary.com/category/tutorials/javascript/reactjs/>
- ReactJS conference 2015 video talks: <http://www.youtube.com/playlist?list=PLb0IAmt7-GS1cbw4qonlQztYV1TAW0sCr>

# Index

## A

- asynchronous task execution / [Asynchronous task execution](#)
- auto-binding feature
  - used, for Polymer templating / [Polymer templating with auto-binding](#)
  - URL / [Polymer templating with auto-binding](#)

## B

- Bosonic
  - about / [What is Bosonic?](#)
  - supported browsers / [Browser support](#)
  - configuring / [Configuring Bosonic](#)
  - URL / [Configuring Bosonic](#)
  - packages / [Bosonic packages](#)
  - built-in elements / [Built-in elements](#)
  - lifecycle / [Bosonic lifecycle](#)
- Bosonic library
  - about / [Bosonic](#)
  - URL / [Bosonic](#)
- Bosonic packages
  - about / [Bosonic packages](#)
  - Bosonic platform / [Bosonic packages](#)
  - Grunt Bosonic / [Bosonic packages](#)
  - Bosonic transpiler / [Bosonic packages](#)
  - Yeoman Bosonic / [Bosonic packages](#)
- Bower
  - URL / [Using Bower, Installing Mozilla Brick](#)
- brick-action element
  - about / [The brick-action element](#)
  - URL / [The brick-action element](#)
- brick-calendar element
  - about / [The brick-calendar element](#)
- brick-deck element
  - about / [The brick-deck element](#)
  - URL / [The brick-deck element](#)

- brick-flipbox element
  - about / [The brick-flipbox element](#)
  - URL / [The brick-flipbox element](#)
- brick-menu element
  - about / [The brick-menu element](#)
- brick-tabbar element
  - about / [The brick-tabbar element](#)
  - URL / [The brick-tabbar element](#)
- built-in components, Mozilla Brick 1.0
  - about / [Built-in components](#)
  - brick-calendar element / [The brick-calendar element](#)
  - brick-flipbox element / [The brick-flipbox element](#)
  - brick-deck element / [The brick-deck element](#)
  - brick-tabbar element / [The brick-tabbar element](#)
  - brick-action element / [The brick-action element](#)
  - brick-menu element / [The brick-menu element](#)
- built-in elements
  - about / [Built-in elements](#)
  - URL / [Built-in elements](#)
  - b-sortable element / [The b-sortable element](#)
  - b-toggle-button element / [The b-toggle-button element](#)
- built-in filtering expression, Polymer expressions
  - about / [Built-in filtering expression](#)
  - TokenList filter / [The TokenList filter](#)
  - styleObject filter / [The styleObject filter](#)

## C

- Can I Use tool
  - URL / [Template feature detection](#)
- cloneNode method
  - using / [Cloning a node](#)
  - URL / [Cloning a node](#)
- component lifecycle, React.js
  - initialization phase / [ReactJS component lifecycle](#), [ReactJS initialization phase](#)
  - lifetime phase / [ReactJS component lifecycle](#), [ReactJS lifetime phase](#)
  - teardown phase / [ReactJS component lifecycle](#), [ReactJS teardown phase](#)

- example / [ReactJS lifecycle example](#)
- URL / [ReactJS lifecycle example](#)
- componentWillUnmount method / [ReactJS teardown phase](#)
- content
  - about / [Template element detail](#)
- content insertion point
  - about / [A content insertion point](#)
- core-elements, Polymer.js architecture
  - about / [Core elements](#)
  - URL / [Core elements](#)
  - core-input element / [The core-input element](#)
  - core-label element / [The core-label element](#)
  - core-tooltip element / [The core-tooltip element](#)
- core-input element
  - URL / [The core-input element](#)
- core range element properties
  - URL / [The paper-slider element](#)
- custom component
  - developing / [Developing custom component](#)
  - red-message element directory, creating / [Step 1 – creating the red-message element directory](#)
  - current directory, changing to red message / [Step 2 – changing the current directory to red-message](#)
  - skeleton, generating for <red-message> / [Step 3 – generating the skeleton for <red-message>](#)
  - directory structure, verifying / [Step 4 – verifying the directory structure](#)
  - <red-message> element code, defining / [Step 5 – defining code for the <red-message> element](#)
  - index.html demo file, modifying / [Step 6 – modifying the index.html demo file](#)
  - distribution files, generating with Grunt / [Step 7 – generating distribution files using Grunt](#)
  - index.html file, running / [Step 8 – running the index.html file](#)
- custom element
  - about / [Custom element](#)
  - feature detection / [Custom element feature detection](#)
  - developing / [Developing a custom element](#)

- new object, creating / [Creating a new object](#)
- object properties, defining / [Defining object properties](#)
- lifecycle methods, defining / [Defining lifecycle methods](#)
- new element, registering / [Registering a new element](#)
- element, extending / [Extending an element](#)
- example / [Example of a custom element](#)
- extending / [Extending a custom element](#)
- custom X-Tag element
  - creating / [X-Tag custom element development](#)
  - lifecycle property / [X-Tag custom element development](#)
  - methods property / [X-Tag custom element development](#)
  - events property / [X-Tag custom element development](#)
  - accessors sproperty / [X-Tag custom element development](#)

## D

- digital clock
  - developing / [Developing a digital clock](#), [Digital clock development](#)
  - developing, with X-Tag library / [Developing a digital clock using X-Tag](#)
  - React.js, using / [Developing a digital clock using ReactJS](#)
- digital clock, with React.js
  - about / [Developing a digital clock using ReactJS](#)
  - lifecycle script, defining / [Step1 – defining the digital clock lifecycle script](#)
  - CSS styles , defining / [Step2 – defining CSS styles for the digital clock](#)
- digital clock component
  - building / [Building a digital clock component](#)
  - digital clock template / [Clock template](#)
  - clock element registration script / [Clock element registration script](#)
  - using / [Using the clock component](#)
- document.registerElement method
  - URL / [Registering a new element](#)
- DocumentFargment
  - about / [Template element detail](#)
  - URL / [Template element detail](#)

## E

- elements, Polymer.js architecture
  - core-elements / [Core elements](#)

- paper elements / [Paper elements](#)

## F

- Flash of Unstyled Content (FOUC) / [PolymerJS ready event](#)
- flux architecture
  - dispatcher / [The flux architecture](#)
  - store / [The flux architecture](#)
  - action / [The flux architecture](#)
  - views / [The flux architecture](#)
  - application, working / [The flux architecture](#)
  - key features / [Flux key features](#)
  - URL / [Flux key features](#)
- forceReady method / [The Polymer forceReady method](#)

## G

- GitHub token
  - URL / [Getting a GitHub token](#)
- Google Material Design
  - URL / [Paper elements](#)
- Grunt
  - used, for generating distribution files / [Step 7 – generating distribution files using Grunt](#)
- grunt-vulcanize
  - URL / [Running vulcanize process](#)
- gulp-vulcanize
  - URL / [Running vulcanize process](#)

## H

- HTML <template> element
  - about / [Template element](#)
  - detail / [Template element detail](#)
  - interface definition language (IDL) definition / [Template element detail](#)
  - IDL language, using / [Template element detail](#)
  - template feature detection / [Template feature detection](#)
  - inert template / [Inert template](#)
  - activating / [Activating a template](#)
  - node, cloning / [Cloning a node](#)

- node, importing / [Importing a node](#)
- HTML import
  - about / [HTML Import](#)
  - feature detection / [HTML Import feature detection](#)
  - document, accessing / [Accessing the HTML Import document](#)
  - events / [HTML Import events](#)
  - load event / [HTML Import events](#)
  - error event / [HTML Import events](#)
  - handleSuccess method / [HTML Import events](#)
  - handleError method / [HTML Import events](#)
- HTMLElement
- about / [Template element detail](#)

## I

- IDL, shadow root element
  - about / [Shadow tree](#)
  - getElementById / [Shadow tree](#)
  - getElementsByClassName / [Shadow tree](#)
  - getElementsByTagName / [Shadow tree](#)
  - getElementsByTagNameNS / [Shadow tree](#)
  - getSelection / [Shadow tree](#)
  - elementFromPoint / [Shadow tree](#)
  - activeElement / [Shadow tree](#)
  - host / [Shadow tree](#)
  - olderShadowRoot / [Shadow tree](#)
  - innerHTML / [Shadow tree](#)
  - styleSheets / [Shadow tree](#)
- importNode method
  - using / [Importing a node](#)
- initialization phase, React.js
  - about / [ReactJS initialization phase](#)
  - getDefaultProps method / [ReactJS initialization phase](#)
  - getInitialState method / [ReactJS initialization phase](#)
  - componentWillMount method / [ReactJS initialization phase](#)

## J

- JSX

- about / [What is JSX](#)
- benefits / [What is JSX](#)
- custom component, using / [Custom components with JSX](#)
- inline style, adding in React.js / [ReactJS inline style](#)
- event handling, in React.js / [ReactJS event handling](#)
- non-DOM attributes / [Useful non-DOM attributes](#)

## L

- lifecycle, Bosonic
  - callback methods / [Bosonic lifecycle](#)
  - example / [Example of lifecycle](#)
- lifetime phase, React.js
  - about / [ReactJS lifetime phase](#)
  - componentWillReceiveProps method / [ReactJS lifetime phase](#)
  - shouldComponentUpdate method / [ReactJS lifetime phase](#)
  - componentWillUpdate method / [ReactJS lifetime phase](#)
  - render method / [ReactJS lifetime phase](#)
  - componentDidUpdate method / [ReactJS lifetime phase](#)

## M

- Material Design specification
  - URL / [Material design](#)
- Mozilla Brick
  - URL / [Mozilla Brick](#)
  - installing / [Installing Mozilla Brick](#)
  - configuring / [Configuring Mozilla Brick](#)
- Mozilla Brick 1.0
  - about / [Mozilla Brick 1.0](#)
  - built-in components / [Built-in components](#)
- Mozilla Brick 2.0
  - about / [Mozilla Brick 2.0](#)
- Mozilla Brick library
  - about / [Mozilla Brick, What is the Brick library?](#)
  - URL / [What is the Brick library?](#)
  - Mozilla Brick 1.0 / [Mozilla Brick 1.0](#)
  - Mozilla Brick 2.0 / [Mozilla Brick 2.0](#)
- Mutation observer

- URL / [Web components with polyfill](#)

## N

- node, HTML <template> element
  - cloning / [Cloning a node](#)
  - deep cloning / [Cloning a node](#)
  - shallow cloning / [Cloning a node](#)
  - importing / [Importing a node](#)
- node distribution
  - about / [Node distribution](#)
  - content insertion point / [A content insertion point](#)
  - shadow insertion point / [A shadow insertion point](#)
- node package manager (npm) / [Using Bower](#), [Vulcanize installation](#), [Bosonic packages](#)
- non-DOM attributes
  - key / [Useful non-DOM attributes](#)
  - ref / [Useful non-DOM attributes](#)
  - dangerouslySetInnerHTML / [Useful non-DOM attributes](#)

## O

- Object.create method
  - URL / [Creating a new object](#)
- Object.defineProperty method
  - URL / [Defining object properties](#)
- object lifecycle, custom element
  - created state / [Defining lifecycle methods](#)
  - attached state / [Defining lifecycle methods](#)
  - detached state / [Defining lifecycle methods](#)
  - attributeChanged state / [Defining lifecycle methods](#)
- object properties, custom element
  - targetObject / [Defining object properties](#)
  - propertyName / [Defining object properties](#)
  - propertySettings / [Defining object properties](#)
- operations, Polymer expressions
  - identifiers and paths / [Polymer expressions](#)
  - array access / [Polymer expressions](#)
  - logical not operator / [Polymer expressions](#)

- unary operators / [Polymer expressions](#)
- binary operators / [Polymer expressions](#)
- comparators / [Polymer expressions](#)
- logical comparators / [Polymer expressions](#)
- ternary operator / [Polymer expressions](#)
- parenthesis / [Polymer expressions](#)
- literal values / [Polymer expressions](#)
- array and object initializers / [Polymer expressions](#)
- function / [Polymer expressions](#)

## P

- paper-slider element
  - URL / [The paper-slider element](#)
- paper elements, Polymer.js architecture
  - about / [Paper elements](#)
  - URL / [Paper elements](#)
  - material design / [Material design](#)
  - paper-checkbox element / [The paper-checkbox element](#)
  - paper-slider element / [The paper-slider element](#)
  - paper-button element / [The paper-button element](#)
- placeholder / [A content insertion point](#)
- polyfill
  - URL / [Mozilla Brick 2.0](#)
- Polymer
  - about / [What is Polymer?](#)
  - installing / [Installing and configuring Polymer](#)
  - configuring / [Installing and configuring Polymer](#)
  - native layer / [Architecture of PolymerJS](#)
  - foundation layer / [Architecture of PolymerJS](#)
  - core layer / [Architecture of PolymerJS](#)
  - elements layer / [Architecture of PolymerJS](#)
  - Web Components, using with polyfill / [Web components with polyfill](#)
  - library / [The Polymer library](#)
  - expressions / [Polymer expressions](#)
  - templating, with auto-binding / [Polymer templating with auto-binding](#)
  - template attributes / [Polymer template attributes](#)
- Polymer-generator commands

- about / [The polymer-generator commands](#)
- polymer application generator / [The Polymer application generator](#)
- Polymer element generator / [The Polymer element generator](#)
- Polymer seed generator / [The Polymer seed generator](#)
- Polymer GitHub page generator / [The Polymer GitHub page generator](#)
- Polymer.import method / [The Polymer import method](#)
- Polymer.js
  - architecture / [Architecture of PolymerJS](#)
- Polymer.js architecture
  - about / [Architecture of PolymerJS](#)
  - elements / [Elements](#)
- Polymer.js ready event / [PolymerJS ready event](#)
- Polymer configuration
  - ZIP file, downloading / [Downloading ZIP file](#)
  - GIT clone, using / [Using GIT clone](#)
  - Bower, using / [Using Bower](#)
- Polymer custom element
  - developing / [Developing Polymer custom elements](#)
  - defining / [Defining a custom element](#)
  - attributes, defining / [Defining element attributes](#)
  - default attributes defining / [Defining default attributes](#)
  - public properties, defining / [Defining public properties and methods](#)
  - methods, defining / [Defining public properties and methods](#)
  - properties, publishing / [Publishing properties](#)
  - lifecycle method, defining / [Defining a lifecycle method](#)
  - registering / [Registering a custom element](#)
- Polymer designer tool
  - about / [Polymer designer tool](#)
  - URL / [Polymer designer tool](#)
  - used, for developing / [Developing with the designer tool](#)
  - GitHub token, obtaining / [Getting a GitHub token](#)
  - e-mail subscription form, developing / [Developing an e-mail subscription form](#)
- polymer directory
  - polymer.js file / [The Polymer library](#)
  - polymer-min.js file / [The Polymer library](#)
  - layout.html file / [The Polymer library](#)

- polymer.html file / [The Polymer library](#)
- Polymer element generator
  - URL / [The Polymer element generator](#)
- Polymer expressions
  - about / [Polymer expressions](#)
  - operations / [Polymer expressions](#)
  - Polymer templating / [Polymer templating with auto-binding](#)
  - filtering / [Filtering expression](#)
  - built-in filtering expression / [Built-in filtering expression](#)
  - custom filtering expression / [Custom filtering expression](#)
  - global filtering expression / [Global filtering expression](#)
- Polymer GitHub page generator
  - URL / [The Polymer GitHub page generator](#)
- Polymer installation
  - ways / [Installing and configuring Polymer](#)
  - ZIP file, downloading / [Downloading ZIP file](#)
  - GIT clone, using / [Using GIT clone](#)
  - Bower, using / [Using Bower](#)
- Polymer library
  - about / [Polymer](#)
  - URL / [Polymer](#)
- Polymer methods
  - about / [The Polymer mixin method](#)
  - mixin method / [The Polymer mixin method](#)
  - import method / [The Polymer import method](#)
  - waitingFor method / [The Polymer waitingFor method](#)
  - forceReady method / [The Polymer forceReady method](#)
- Polymer seed generator
  - URL / [The Polymer seed generator](#)
- precompiled JSX
  - using, for production / [Precompiled JSX for production](#)
  - JSX file watcher / [JSX file watcher](#)
- private scope / [Benefits and challenges of web components](#)
- propertySettings, custom element
  - configurable / [Defining object properties](#)
  - enumerable / [Defining object properties](#)
  - value / [Defining object properties](#)

- writable / [Defining object properties](#)
- get / [Defining object properties](#)
- set / [Defining object properties](#)

- pseudo selectors

- about / [Styling web components](#)
- Unresolved pseudo selector / [Styling web components](#)
- Host pseudo selector / [Styling web components](#)
- Shadow pseudo selector / [Styling web components](#)
- Content pseudo selector / [Styling web components](#)

# R

- React.js

- approach / [The reactive approach](#)
- installing / [Installing ReactJS](#)
- core library / [Installing ReactJS](#)
- addon / [Installing ReactJS](#)
- JSX transformer library / [Installing ReactJS](#)
- configuring / [Configuring ReactJS](#)
- using / [Using ReactJS](#)
- inline style / [ReactJS inline style](#)
- event handling / [ReactJS event handling](#)
- component lifecycle / [ReactJS component lifecycle](#)
- using, for digital clock / [Developing a digital clock using ReactJS](#)
- debugging / [Debugging ReactJS](#)
- debugging tool, URL / [Debugging ReactJS](#)

- React.js library

- about / [ReactJS](#)
- URL / [ReactJS](#)

- React developer tool

- URL / [Debugging ReactJS](#)

- reprojection / [A content insertion point](#)

# S

- sample custom element

- developing / [Developing a sample custom element](#)

- Shadow DOM

- about / [Shadow DOM](#)

- enabling / [Shadow DOM](#)
- feature detection / [Shadow DOM feature detection](#)
- Shadow DOM tree
  - about / [Shadow tree](#)
  - document tree / [Shadow tree](#)
  - shadow tree / [Shadow tree](#)
  - composed tree / [Shadow tree](#)
  - host element / [Shadow tree](#)
  - shadow host / [Shadow tree](#)
  - sample DOM tree / [Shadow tree](#)
  - IDL, of shadow root element / [Shadow tree](#)
  - example / [Shadow tree](#)
- shadow insertion point
  - about / [A shadow insertion point](#)  
/ [Extending a custom element](#)
- stateful custom component / [Stateful custom component](#)
- supported browsers, Bosonic / [Browser support](#)

## T

- template attributes, Polymer
  - about / [Polymer template attributes](#)
  - bind / [Polymer template attributes](#)
  - repeat / [Polymer template attributes](#)
  - if / [Polymer template attributes](#)
  - ref / [Polymer template attributes](#)
- TemplateBinding, Polymer
  - URL / [Polymer template attributes](#)
- toggle method / [The brick-flipbox element](#)
- transpiler / [What is Bosonic?](#)

## U

- UIKit / [What is the Brick library?](#)

## V

- vulcanize
  - used, for preparing production / [Preparing for production using vulcanize](#)
  - installation / [Vulcanize installation](#)

- process, running / [Running vulcanize process](#)
- URL / [Running vulcanize process](#)

## W

- waitingFor method / [The Polymer waitingFor method](#)
- WeakMap
  - URL / [Web components with polyfill](#)
- Web Component
  - references / [Chapter 1](#), [Chapter 3](#), [Chapter 6](#)
- Web Component polyfill
  - URL / [Web components with polyfill](#)
- Web Components
  - about / [What are web components?](#)
  - benefits / [Benefits and challenges of web components](#)
  - challenges / [Benefits and challenges of web components](#)
  - architecture / [The web component architecture](#)
  - styling / [Styling web components](#)

## X

- X-Tag
  - about / [X-Tag](#)
  - URL / [X-Tag](#)
  - lifecycle / [X-Tag element lifecycle](#)
  - created event / [X-Tag element lifecycle](#)
  - inserted event / [X-Tag element lifecycle](#)
  - removed event / [X-Tag element lifecycle](#)
  - attributeChanged event / [X-Tag element lifecycle](#)
  - custom element development / [X-Tag custom element development](#)
  - Polymer library / [Polymer](#)
  - Mozilla Brick library / [Mozilla Brick](#)
  - React.js library / [ReactJS](#)
  - Bosonic library / [Bosonic](#)
- X-Tag core library
  - URL / [X-Tag custom element development](#)
- X-Tag library
  - about / [The X-Tag library](#)
  - URL / [The X-Tag library](#)

- files / [The X-Tag library](#)
- used, for developing digital clock / [Developing a digital clock using X-Tag](#)

## Y

- Yeoman
  - about / [Yeoman Polymer generator](#)
  - working with / [Working with Yeoman](#)
  - element generator / [Yeoman element generator](#)
  - seed generator / [Yeoman seed generator](#)
  - GitHub page generator / [Yeoman GitHub page generator](#)
  - GitHub page generator, URL / [Yeoman GitHub page generator](#)
- Yeoman Polymer generator
  - about / [Yeoman Polymer generator](#)
  - commands / [The polymer-generator commands](#)

## Z

- ZIP file
  - URL, for downloading / [Downloading ZIP file](#)