Community Experience Distilled

Getting Started with Polymer

Explore the whole new world of web development and create responsive web apps using Polymer

Arshak Khachatrian

[PACKT] open source*
PUBLISHING
community experience distilled

Getting Started with Polymer

Explore the whole new world of web development and create responsive web apps using Polymer

Arshak Khachatrian

[PACKT] open source
PUBLISHING
community experience distilled

BIRMINGHAM - MUMBAI

Getting Started with Polymer

First published: June 2016

Production reference: 1270616

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78588-937-0

www.packtpub.com

Credits

Author Arshak Khachatrian	**Project Coordinator** Suzanne Coutinho
Reviewer Thibaut Gensollen	**Proofreader** Safis Editing
Commissioning Editor Neil Alexander	**Indexer** Hemangini Bari
Acquisition Editor Reshma Raman	**Production Coordinator** Arvindkumar Gupta
Content Development Editor Pooja Mhapsekar	**Cover Work** Arvindkumar Gupta
Technical Editor Taabish Khan	
Copy Editors Pranjali Chury Shruti Iyer	

About the Author

Arshak Khachatrian is a programmer who was born on May 16, 1997, in Yerevan, Armenia.

Since an early age, he has always been interested in computers, and after turning 10 years old, he had his first computer presented to him on his birthday by his father. A week later, Arshak was supporting all the computers in the entire town by installing the Windows operating system and solving various OS issues.

After completing primary school, he decided to study mathematics and physics in high school and enrolled at Polytechnic High School, where he deepened his knowledge of mathematics and physics. Following this, Arshak was accepted in the Tumo Center. At Tumo, he first encountered programming languages, using them to design sites and program robots and fountains. A year after graduating, Arshak decided to create his first big project, the solar system in the browser, and one week after that, he started writing a hard code that he published on a social network. Soon after, he was called to work with the X-Tech company as a JavaScript developer. It was the beginning of his career. Then, Arshak accepted an offer from the BetConstruct company and developed his knowledge in the programming sphere. Thereafter, he decided to support Google and change the world by joining the Google Developers Group Armenia in 2014 and then started to contribute to the Polymer team by writing articles and creating open source components and tools for customelements.io.

In November, 2015, Arshak had a Polymer Code Lab at GDG DevFest at TUMO. In 2016, he and his fellow designer Serge Navasardyan decided to found their startup based on a 360 website builder called POP360.

Finally, in March 2016, Arshak joined the famous programmer Rouben Meschian and started working with the Cambridge Semantics company in Boston.

Acknowledgments

Hey there! Thank you for choosing my book. When I was just starting to learn programming in school, I did not understand so well how big this world is. We have big opportunities and little time. Every day is a competition in which you win or lose. We get our "human" label according our victories, and when we lose, we try again the next day. This book is a victory for me. It pushes me to win in the upcoming days. I'm behind this book, and all the people who have supported me during the whole writing process are by my side. A very big thank you goes to my mother, Gohar Mikaelyan, who motivated me to write and finish this book; to my sister Nelly, who helped me with the translation of the book; and to Anna and Melania, as well. I want to thank to all my friends and, most importantly, my father, Karen Khachatryan, who gave me the chance to enter the digital world when I was 10 years old by giving me my first computer. I would like to express very warm thanks to the Packt Publishing staff—in particular, Reshma Raman, Pooja Mhapsekar, Taabish Khan, and Amey Varangaonkar—for helping and supporting me all this time.

About the Reviewer

Thibaut Gensollen is a 25-year-old French guy born in Bordeaux, passionate about the computer sciences and, especially, web development. He got a master's degree of science in electrical engineering and computer sciences from École Normale Supérieure, Cachan, with a major in machine learning. During his studies, Thibaut got some research experience at UC Berkeley and University of Michigan, and he also worked at Orange, San Francisco, and Deloitte, Paris, where he started learning Polymer.

Thibaut's passion for the Web started at the age of 12, when he designed and coded websites for people. Since then, he never stopped designing and coding websites, mobile applications, or databases. Thibaut was the co-founder of a designing community, graphistes-world, which was one of the best French communities at the time.

With time, he learned more and more about lots of fields, such as SEO, backends, functional programing, machine learning, and more. Thibaut is now really enthusiastic about all of these new upcoming technologies, such as Polymer.

He is currently thinking about doing a PhD in deep learning, and he is the C.T.O. of a mobile application called CHOOSE, which has thousands of daily active users, using libraries such as Polymer, Ionic, and AngularJS.

I would like to thank Packt Publishing for giving me this opportunity to review the book and my parents for letting me spend so much time on my computer when I was young.

Table of Contents

Preface

Web technologies are growing day by day, and people are trying to improve and create things that will facilitate and accelerate their lives.

A few years ago, it was very problematic to create a website that was nice and quick for all devices; we had to write a few hundred lines of CSS and HTML code in order to develop a responsive website for all.

Now, there is a tool that lets you easily create websites that work quickly and, at the same time, look very nice on all devices. This tool is called Polymer.

Polymer is a collection of components that you can use to create your own components and construct a website, much like Lego. In this book, you will learn how to use these components, how to create your own, and how to collect all this into a simple app.

What this book covers

Chapter 1, *Web Components*, introduces you to HTML imports, custom elements, Shadow DOM, and templates to easily create applications using new web development technologies.

Chapter 2, *Material Design*, teaches you Material Design concepts such as how to create material applications using Material Design animations, styles, and layouts.

Chapter 3, *Introduction to Polymer*, introduces you to Polymer and its features to make the creation of web components simpler and faster.

Chapter 4, *Polymer Elements*, presents a set of useful elements, such as iron elements, paper elements, Google web components, gold elements, neon elements, platinum elements, and molecules.

Chapter 5, *First Application with Polymer*, teaches you how to create a simple app with Polymer.

Chapter 6, *Polymer Designer Tool and Polymer Starter Kit*, familiarizes you with some tools to make your job faster.

Chapter 7, *Working with Polymer.dart*, provides you with a brief start on Polymer.dart.

Chapter 8, *Best Practices*, contains a lot of cool stuff about Polymer including how to write clean and awesome code that is high performance and with minimal bugs.

What you need for this book

The main thing we need is an editor. I recommend that you use the Atom and Sublime Text editors as they're easy to configure and get the packages in these editors. You need Node.js on your computer to run the npm commands and get the Bower, Grunt, and Polymer components.

Who this book is for

If you are a beginner-level web developer who wants to learn the concepts of web development using the Polymer library, then this is the book for you. Knowledge of JavaScript and HTML is expected.

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:

```
<style>
  :host > div {
    background-color: teal;
  }
</style>
```

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

```
some-behavior.html
<script>
    SomeBehavior = {
      properties: {
        isSelected: {
```

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

```
bower update
```

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: "It is a simple drawer panel, with four menu items: **My Music**, **Poly Music**, **Users**, and **About the app**."

[Warnings or important notes appear in a box like this.]

[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`, 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`.

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 for this book from your account at `http://www.packtpub.com`. 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.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

You can also download the code files by clicking on the **Code Files** button on the book's webpage at the Packt Publishing website. This page can be accessed by entering the book's name in the **Search** box. Please note that you need to be logged in to your Packt account.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the app in *Chapter 5*, *First Application with Polymer*, is hosted on GitHub at `https://github.com/AKHXtern/poly`. The code bundle for *Chapter 1*, *Web Components*, is hosted on GitHub at `https://github.com/AKHXtern/polymer-book-examples`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

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` 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`, and we will do our best to address the problem.

1
Web Components

Currently, web technologies are growing rapidly. Though most current websites use these technologies, we come across many with a bad, unresponsive UI design and awful performance. The only reason we should think about a responsive website is that users are now moving to the mobile web. 55% of web users use mobile phones because they are faster and more comfortable. This is why we need to provide mobile content in the simplest way possible. Everything is moving to minimalism, even the Web.

The new web standards are changing rapidly too. In this chapter, we will cover one of these new technologies, web components, and what they do. Using web components, you can easily create your web application by splitting it into parts/components.

We will cover the following topics:

- Introduction to web components
- Templates
- Shadow DOM
- Custom elements
- HTML imports
- Styles and selectors

These are the specifications of web components. Each one of these creates magic. Polymer is a library that uses the features of web components. Hence, we need to study web components in detail before we start working with it.

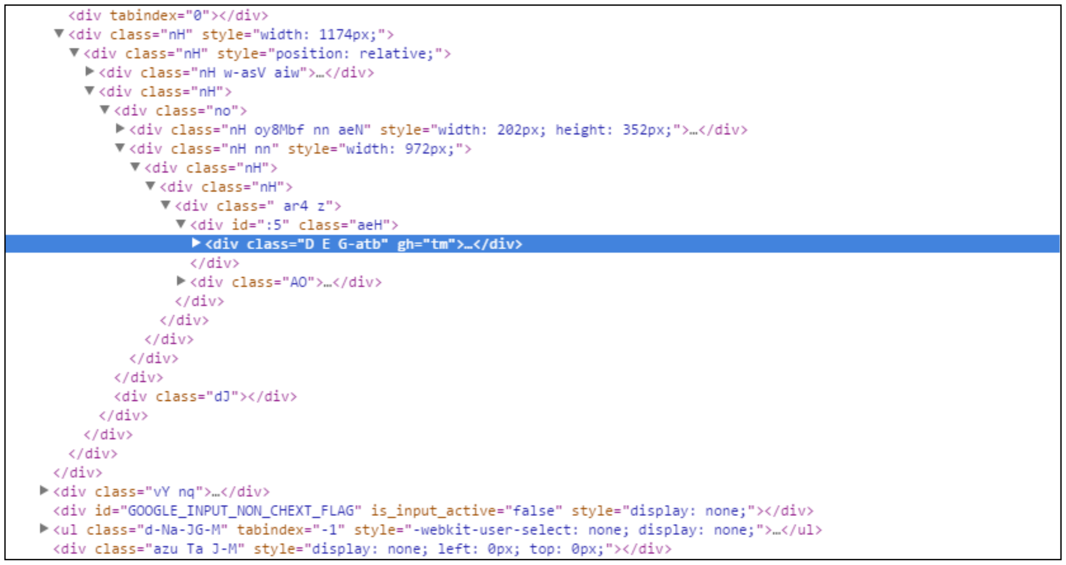
Introduction to Web Components

Let's talk about web components. What are they? Why should we use them on the Web and what kind of problems can they solve?

Imagine for a moment, what if you could write less code and make more things than now, what if you could just create a `<website>` component and give it some attributes and create your entire website in a second. You think it is possible only in the future? Nope, it's happening now! Web components are here and they will solve all your problems.

There is no issue with browser support because now (at the time of writing this chapter) all modern browsers support web component polyfills.

Look at the following screenshot of code blocks from Gmail:



```
    <div tabindex="0"></div>
  ▼<div class="nH" style="width: 1174px;">
    ▼<div class="nH" style="position: relative;">
      ▶<div class="nH w-asV aiw">…</div>
      ▼<div class="nH">
        ▼<div class="no">
          ▶<div class="nH oy8Mbf nn aeN" style="width: 202px; height: 352px;">…</div>
          ▼<div class="nH nn" style="width: 972px;">
            ▼<div class="nH">
              ▼<div class="nH">
                ▼<div class=" ar4 z">
                  ▼<div id=":5" class="aeH">
                    ▶<div class="D E G-atb" gh="tm">…</div>
                    </div>
                  ▶<div class="AO">…</div>
                  </div>
                </div>
              </div>
            </div>
          </div>
          <div class="dJ"></div>
        </div>
      </div>
    </div>
  </div>
 ▶<div class="vY nq">…</div>
  <div id="GOOGLE_INPUT_NON_CHEXT_FLAG" is_input_active="false" style="display: none;"></div>
 ▶<ul class="d-Na-JG-M" tabindex="-1" style="-webkit-user-select: none; display: none;">…</ul>
  <div class="azu Ta J-M" style="display: none; left: 0px; top: 0px;"></div>
```

If you can read and understand this code, the only thing I would say to you is that *you're a hero*, because I can't!

What if I showed you code similar to the following? (I like this one, by the way):

```
<iron-pages role="main" selected="[[page]]" attr-for-selected="name" selected-attribute="visible">
  <!-- home view -->
  <shop-home name="home" categories="[[categories]]"></shop-home>

  <!-- list view of items in a category -->
  <shop-list name="list" route="[[subroute]]" offline="[[offline]]"></shop-list>

  <!-- detail view of one item -->
  <shop-detail name="detail" route="[[subroute]]" offline="[[offline]]"></shop-detail>

  <!-- cart view -->
  <shop-cart name="cart" cart="[[cart]]" total="[[total]]"></shop-cart>

  <!-- checkout view -->
  <shop-checkout
      name="checkout"
      cart="[[cart]]"
      total="[[total]]"
      route="{{subroute}}"></shop-checkout>
</iron-pages>
```

This code is written using the features of web components and is way easier to read/understand/execute. Now, compare both the examples. Do you see any difference? I guess you already know which one is better.

Web components allow us to create complex UI widgets and applications. We can fill in the gaps with our own reusable components and then use them whenever we want.

There is a saying that:

> *"If you think that the Web was changed by HTML5, then wait and see what changes web components make."*

Web components are a new era of web development and, in this book, we will discuss the benefits you can get using Polymer from Google. You can create your own elements that contain templates, encapsulated styles, and logic (JS). They also take advantage of a rich collection of ready-made elements (take a look at `http://customelements.io`).

"The platform of the future" is a set of standards that allows us to describe new types of DOM elements with their properties and methods that encapsulate their DOM and styles.

This means that the styles you have in a document will always render as you intend them to and your HTML code is safe from other users of external JavaScript code.

Web components have four specifications and we will discuss them in this chapter. They are as follows:

- Templates
- Shadow DOM
- Custom elements
- HTML imports

Templates

In this section, we will discuss what we can do with templates. However, let's answer a few questions before this.

What are templates and why should we use them?

Templates are basically fragments of HTML, but let's call these fragments the "zombie" fragments of HTML as they are neither alive nor dead. What is meant by "neither alive nor dead"? Let me explain this with a real-life example.

Once, when I was working on the **ucraft.me** project (it's a website built with a lot of cool stuff in it), we faced some rather new challenges with the templates. We had a lot of form elements, but we didn't know where to save the form elements content. We didn't want to load the DOM of each form element, but what could we do? As always, we did some magic; we created a lot of div elements with the form elements and hid it with CSS. But the CSS `display: none` property did not render the element, it loaded the element. This was also a problem because there were a lot of form element templates and it affected the performance of the website.

I recommended to my team that they work with templates. Templates can contain HTML content, but they do not load the element nor render.

We call template elements "dead elements" because they do not load the content until you get their content with JavaScript. Let's move ahead and let me show you some examples of how you can create templates and do some stuff with their contents.

Imagine that you are working on a big project where you need to load some dynamic content without AJAX. If I had a task such as this, I would create a PHP file and get its content by calling the jQuery `.load()` function. However, now you can save your content inside the `<template>` element and get the content without any jQuery or AJAX, but instead with a single line of JavaScript code. Let's create a template.

In `index.html`, we have `<template>` and some content we want to get in the future, as shown in the following code block:

```
<template class="superman">
  <div>
   <img src="assets/img/superman.png" class="animated_superman" />
  </div>
</template>
```

The time has now come for JavaScript! Execute the following code:

```
<script>
  // selecting the template element with querySelector()

  var tmpl = document.querySelector('.superman');
  //getting the <template> content
  var content = tmpl.content;
  // making some changes in the content
  content.querySelector('.animated_superman').width = 200;

  // appending the template to the body
  document.body.appendChild(content);
</script>
```

So, that's it! Cool, right? The content will load only after you append the content to the document. So, do you realize that templates are a part of the future web? If you are using Chrome Canary, just turn on the flags of experimental web platform features and enable HTML imports and experimental JavaScript.

There are four ways to use templates, which are:

- Add templates with hidden elements in the document and just copy and paste the data when you need it, as follows:

  ```
  <div hidden data-template="superman">
    <div>
      <p>SuperMan Head</p>
      <img src="assets/img/superman.png"
        class="animated_superman" />
    </div>
  </div>
  ```

  However, the problem is that a browser will load all the content. It means that the browser will load but not render images, video, audio, and so on.

- Get the content of the template as a string (by requesting with AJAX or from `<script type="x-template">`).
- However, we might have some problems in working with the string. This can be dangerous for XSS attacks; we just need to pay some more attention to this:

  ```
  <script data-template="batman" type="x-template">
    <div>
      <p>Batman Head this time!</p>
      <img src="assets/img/superman.png"
        class="animated_superman" />
    </div>
  </div>
  ```

- Compiled templates such as Hogan.js (`http://twitter.github.io/hogan.js/`) work with strings. So they have the same flaw as the patterns of the second type.

Templates do not have these disadvantages. We will work with DOM and not with the strings. We will then decide when to run the code.

In conclusion:

- The `<template>` tag is not intended to replace the system of standardization. There are no tricky iteration operators or data bindings.
- Its main feature is to be able to insert "live" content along with scripts.
- Lastly, it does not require any libraries.

Shadow DOM

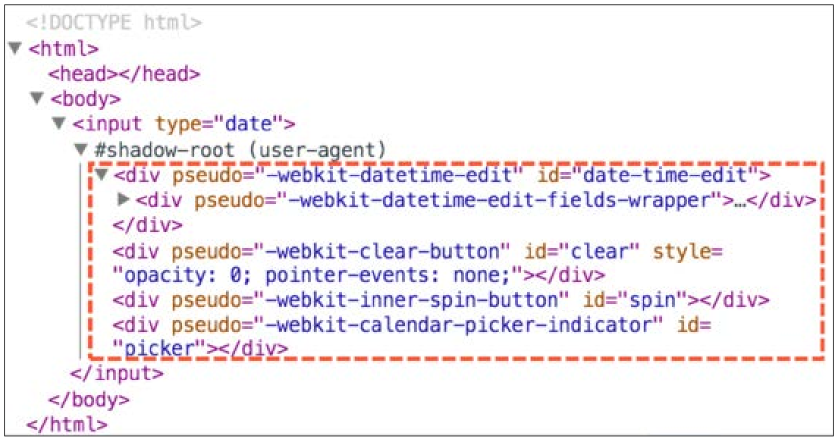
The Shadow DOM specification is a separate standard. A part of it is used for standard DOM elements, but it is also used to create web components. In this section, you will learn what the Shadow DOM is and how to use it.

The Shadow DOM is an internal DOM element that is separated from an external document. It can store your ID, styles, and so on. Most importantly, the Shadow DOM is not visible outside of its scope without the use of special techniques. Hence, there are no conflicts with the external world; it's like an iframe.

Inside the browser

The Shadow DOM concept has been used for a long time inside browsers themselves. When the browser shows complex controls, such as an `<input type = "range">` slider or a `<input type = "date">` calendar within itself, it constructs them out of the most ordinary styled `<div>`, `<span>`, and other elements.

They are invisible at first glance, but they can be easily seen if the checkbox in Chrome DevTools is set to display Shadow DOM:



```
<!DOCTYPE html>
▼ <html>
    <head></head>
  ▼ <body>
    ▼ <input type="date">
      ▼ #shadow-root (user-agent)
        ▼ <div pseudo="-webkit-datetime-edit" id="date-time-edit">
          ▶ <div pseudo="-webkit-datetime-edit-fields-wrapper">…</div>
          </div>
          <div pseudo="-webkit-clear-button" id="clear" style=
          "opacity: 0; pointer-events: none;"></div>
          <div pseudo="-webkit-inner-spin-button" id="spin"></div>
          <div pseudo="-webkit-calendar-picker-indicator" id=
          "picker"></div>
      </input>
    </body>
  </html>
```

In the preceding code, `#shadow-root` is the Shadow DOM.

Getting items from the Shadow DOM can only be done using special JavaScript calls or selectors. They are not children but a more powerful separation of content from the parent.

In the preceding Shadow DOM, you can see a useful pseudo attribute. It is nonstandard and is present for solely historical reasons. It can be styled via CSS with the help of subelements—for example, let's change the form input dates to red via the following code:

```
<style>
  input::-webkit-datetime-edit {
    background: red;
  }
</style>

<input type="date" />
```

Once again, make a note of the pseudo custom attribute. Speaking chronologically, in the beginning, browsers started to experiment with encapsulated DOM structure inside their scopes, then Shadow DOM appeared which allowed developers to do the same.

Now, let's work with the Shadow DOM from JavaScript or the standard Shadow DOM.

Creating a Shadow DOM

The Shadow DOM can create any element within the `elem.createShadowRoot()` call, as shown by the following code:

```
<div id="container">You know why?</div>

<script>
  var root = container.createShadowRoot();
  root.innerHTML = "Because I'm Batman!";
</script>
```

If you run this example, you will see that the contents of the `#container` element disappeared somewhere and it only shows "Because I'm Batman!". This is because the element has a Shadow DOM and ignores the previous content of the element.

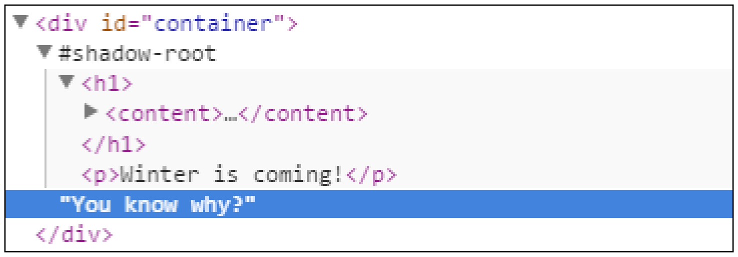
Because of the creation of the Shadow DOM, instead of the content, the browser has shown only the Shadow DOM.

If you wish, you can put HTML content via JavaScript inside this Shadow DOM. To do this, you need to specify where it is to be done. The Shadow DOM is done through the "insertion point" and it is declared using the `<content>` tag; here's an example:

```
<div id="container">You know why?</div>

<script>
  var root = container.createShadowRoot();
  root.innerHTML = '<h1><content></content></h1><p>Winter is
    coming!</p>';
</script>
```

Now, you will see "You know why?" in the title followed by "Winter is coming!".

Here's a Shadow DOM example in Chrome DevTool:



```
▼<div id="container">
  ▼#shadow-root
    ▼<h1>
      ▶<content>…</content>
    </h1>
    <p>Winter is coming!</p>
  "You know why?"
</div>
```

The following are some important details about the Shadow DOM:

- The `<content>` tag affects only the display and it does not move the nodes physically. As you can see in the preceding picture, the node "You know why?" remained inside the `div#container`. It can even be obtained using `container.firstElementChild`.
- Inside the `<content>` tag, we have the content of the element itself. In this example, the string "You know why?".

With the `select` attribute of the `<content>` element, you can specify a particular selector content you want to transfer; for example, `<content select="p"></content>` will transfer only paragraphs.

Inside the Shadow DOM, you can use the `<content>` tag multiple times with different values of `select`, thus indicating where to place which part of the original content. However, it is impossible to duplicate nodes. If the node is shown in a `<content>` tag, then the next node will be missed.

For example, if there is a `<content select="h3.title">` tag and then `<content select= "h3">`, the first `<content>` will show the headers `<h3>` with the class `title`, while the second will show all the others, except for the ones already shown.

In the preceding example from DevTools, the `<content></content>` tag is empty. If we add some content in the `<content>` tag, it will show that if there are no other nodes.

Check out the following code:

```
<div id="container">
  <h3>Once upon a time, in Westeros</h3>
  <strong>Ruled a king by name Joffrey and he's dead!</strong>
</div>

<script>
```

```
  var root = container.createShadowRoot();

  root.innerHTML = '<content select='h3'></content> \
  <content select=".writer"> Jon Snow </content> \
  <content></content>';
</script>
```

When you run the JS code, you will see the following:

- The first `<content select='h3'>` tag will display the title
- The second `<content select = ".hero">` tag would show the hero name, but if there isn't element with this selector, it will take the default value: `<content select=".hero">`
- The third `<content>` tag displays the rest of the original contents of the elements without the header `<h3>`, which it had launched earlier

Once again, note that `<content>` moves nodes on the DOM physically.

Root shadowRoot

After the creation of a root in the internal DOM, the tree will be available as `container.shadowRoot`.

This is a special object that supports the basic methods of CSS requests and is described in detail in `ShadowRoot`.

You need to go through `container.shadowRoot` if you need to work with content in the Shadow DOM. You can create a new Shadow DOM tree of JavaScript; here's an example:

```
<div id="container">Polycasts</div>

<script>
  // create a new Shadow DOM tree for element

  var root = container.createShadowRoot();

  root.innerHTML = '<h1><content></content></h1> <strong>Hey
    googlers! Let\'s code today.</strong>';
</script>

<script>
```

```
      // read data from Shadow DOM for elem

      var root = container.shadowRoot;

      // Hey googlers! Let's code today.
      document.write('<br/><em>container: ' + root.
        querySelector('strong').innerHTML);
      // empty as physical nodes - is content
      document.write('<br/><em>content: ' + root.
        querySelector('content').innerHTML);
    </script>
```

To finish up, Shadow DOM is a tool to create a separate DOM tree inside the cell, which is not visible from outside without using special techniques:

- A lot of browser components with complex structures have Shadow DOM already.
- You can create Shadow DOM inside every element by calling `elem.createShadowRoot()`. In the future, it will be available as `elem.shadowRoot` root and you will be able to access it inside the Shadow DOM. It is not available for custom elements.
- Once the Shadow DOM appears in the element, the content of it is hidden. You can see just the Shadow DOM.
- The `<content>` element moves the contents of the original item in the Shadow DOM only visually. However, it remains in the same place in the DOM structure.

Detailed specifications are given at `http://w3c.github.io/webcomponents/spec/shadow/`.

Now, let's move to custom elements, which are also a part of web components. You can do lots of cool stuff with custom elements, so go ahead to the next topic and keep rocking the world!

Custom elements

In this section, we will discuss another great feature of web components.

A reader with a critical mind would say, "Why do we need more standard types of elements? I can create any element right now! In any of the modern browsers, I can create any HTML tag I want using custom tags (`<customtag>`) or create elements from JavaScript using `document.createElement('customtag')`."

However, the default element with a nonstandard name (for example, `<customtag>`) is seen by the browser as something vague and incomprehensible. It corresponds to the `HTMLUnknownElement` class and it does not have any special methods.

The standard of custom elements allows you to describe the elements of its new properties and methods, declare your DOM and construction similarity, and much more.

Running examples of this book

Since the specification is not final, it is recommended that you run the examples used in this book in Google Chrome, better yet, in the latest build of Chrome Canary, which tends to reflect the latest changes.

Let's look at the following examples.

New item

For the description of a new element, we will use `document.registerElement (name, {prototype: proto})`, which can be explained as follows:

- `name`: This is the name of the new tag, such as `super-tag`. It must contain a dash (`-`). The specification requires a dash to avoid future conflict with the standard elements of HTML. You cannot create an item or timer with the name `myTag`. The DOM will identify this element as an unknown element.
- `prototype`: This is the prototype object for the new element and it must be inherited from the `HTMLElement` property, standard properties, and methods of the element.

Let me show you some examples of how to create custom elements and play with them using JavaScript.

At first, we need to create the custom `<show-logo>` element that has two attributes (`id` and `logo`). Let's go ahead and take a look at the ways in which we can choose to create a custom element, as follows.

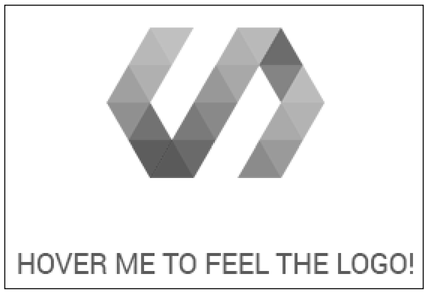
HTML

```
<!—Creating Framework Logos tag -->
<!-- You can use names like (polymer, angular-js, backbone, ember,
underscore, bower) -->
<show-logo id="showLogo" logo="polymer"></show-logo>
```

JavaScript

```
var logo = Object.create(HTMLElement.prototype);
logo.show = function(){
    // Creating info text
    var info = document.createElement('p');
    info.innerHTML = 'Hover me to feel the Logo!';

    var fwork = this.getAttribute('logo');

    var img = document.createElement('img');
    img.src = 'frameworks/logo/' + fwork + '.png';
    img.width = 300;
    img.height = 240;

    this.appendChild(img);
    this.appendChild(info);

    this.onmouseover = function(){
        this.audio.play();
    };
    this.onmouseout = function(){
        this.audio.pause();
    };
};
logo.audio = new Audio('logo/music/logo.ogg');

// Registering our Framework Logo element
document.registerElement('show-logo', {prototype: logo});
showLogo.show();
```

You can see the result in the following figure:



Here, we have our `<show-logo>` component, which contains an image that we have given as the `logo` attribute. The logo gets the image from the attribute value. You can try this live in the Examples page in the `Chapter 1` folder in the accompanying code bundle of this book, available on the Packt website or on GitHub.

Let's have a look at what we did here. We created and registered our `<show-logo>` component, which has the `.show()` method. The `.show()` method creates some information and an image and appends it to our custom element. We also have audio attached here that starts playing when we hover over the element.

In the end, we will call our component's `.show()` method and it will create our scene.

If your new element is not defined in HTML, you can register it through `registerElement()`. The browser has a special mode to "upgrade" the existing elements.

When the browser sees an element with an unknown name that has a dash (`-`) (these elements are called *unresolved*), then:

- It puts this special CSS pseudo class as `:unresolved` and CSS might show that it is still "not loaded"
- When we call `registerElement()`, the elements will be updated automatically to the correct class

In the following example, the registration element takes 3 seconds after the document is loaded, as shown here:

```
<style>
  / * Style for: unresolved element (prior to registration) * /
  say-hello:unresolved {
    color: white;
    background: #cc0000;
  }
  say-hello {
    transition: all 4s;
  }
</style>
<say-hello id="hello">Hey, guys!</say-hello>
<script>
  // Registration will take place in 3 seconds
  setTimeout(function() {
    document.registerElement("say-hello", {
      prototype: {
        __proto__: HTMLElement.prototype,
        sayHelloAgain: function() { alert('I said HEY!'); }
      }
    });

    // the new type of elements is a method sayHelloAgain
    hello.sayHelloAgain();
  }, 3000);
</script>
```

Now, you can create `<say-hello>` elements in JavaScript and call `createElement()`, as follows:

```
var time = document.createElement('say-hello');
```

The expansion of built-in elements

We discussed an example to create an element based on the `HTMLElement` base. But it is possible to expand more specific HTML elements.

To extend built-in elements, there is a `registerElement()` option called `extends` in which you can specify the tag you are inheriting from.

The following is an example of a button:

```
<script>
  var proto = Object.create(HTMLButtonElement.prototype);
  proto.count = function() {
    this.innerHTML++;
  };

  document.registerElement("timer-tag", {
    prototype: proto,
    extends: 'button'
  });
</script>

<button is="timer-tag" id="counter">0</button>

<script>
  setInterval(function() {
    counter.count();
  }, 1000);

  counter.onclick = function() {
    alert("Current value: " + this.innerHTML);
  };
</script>
```

Some points to check are as follows:

- The prototype is not inheriting from `HTMLElement` but from `HTMLButtonElement`.

  To expand an element, it is necessary to inherit the prototype of its class.

- In HTML, you can see the `is='…'` attribute.

  This is HTML's version of `extends` in the JavaScript prototype. It extends the type of the element and applies to its prototype. Now, `<timer-tag>` will not work, you need to implement the tag and use the `is` attribute.

- Work methods, styles, and button events.

  When you click on the button from the previous example, it won't identify whether it is a built-in element. By the way, it is built in and the method is `tick()`.

  When you use the new element in JS, you use `extends`; you must specify and include the source tag, as shown:

  ```
  var time = document.createElement("button", "timer-tag");
  ```

Life cycles

In the prototype of the element, we can define special methods that will be fired when they are created, attached, or detached from the DOM, as shown in the following table:

`createdCallback`	The element is created
`attachedCallback`	The element is added to the document
`detachedCallback`	The element is removed from the document
`attributeChangedCallback(name, pValue, nValue)`	The attribute is added, modified, or deleted

As you probably noticed, `createdCallback` is an inspired designer. It is called only when an item is created, so it makes sense to describe any additional initialization.

Let's use `createdCallback` to initialize the timer and `attachedCallback` to start the timer automatically when you append it into your document:

```
<script>
  var proto = Object.create(HTMLElement.prototype);

  proto.count = function() {
    this.counter++;
    this.innerHTML = this.counter;
  };

  proto.createdCallback = function() {
    this.counter = 0;
  };

  proto.attachedCallback = function() {
    setInterval(this.count.bind(this), 1000);
  };

  document.registerElement("timer-tag", {
    prototype: proto
  });
</script>

<timer-tag id="timer">0</timer-tag>
```

So, we have discussed how to create DOM elements using standard custom elements, and we will further explore new opportunities to work with web components. Next, we will move on to further exploring opportunities to work with web components, which are called HTML imports.

HTML imports

The new specification of HTML imports describes how to insert one document into another using the HTML tag `<link rel="import">`.

Yeah, you're right! There's an `<iframe>` element in the HTML code, so why should we use HTML imports instead of iframes?

With `<iframe>`, everything is okay. However, the meaning of `<iframe>` is a separate document:

- The `<iframe>` element is entirely another environment; it has its own window object and variables
- If the `<iframe>` element is loaded from a different domain, then the interaction with `<iframe>` is possible only through `postMessage`

It's useful when you want to show one page's content on another.

However, what if you want to build another document as a natural part of this? This can be done with a single scripting space and with the same style, but at the same time, it will be a new document.

For instance, it is necessary to load the external parts of the document (web components) from the outside. This is an excellent component because you will not to have to face origin problems with different domains. If we really want to connect an HTML page in one domain with another, we should be able to do it without having to "dance with a tambourine."

In other words, `<link rel="import">` is an analogue of `<script>` for the connection of full documents, templates, libraries, web components, and so on. Everything will become clear when we look at the details.

Let's consider the following example of insertion:

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

- Unlike the `<iframe>` tag, `<link rel="import">` can be anywhere in the document, even in `<head>`.
- When you insert via `<iframe>`, the document is displayed in a frame. In the case of `<link rel="import">`, it is not displayed, and the imported document does not appear at all.

HTML loaded via `<link rel="import">` has a separate document, but the scripts it contains are carried out in the general context of the page.

The file (`imported.html`) is loaded via `<link rel = "import">`, is processed, the scripts are run, and the DOM implementation is built. However, it is not shown and is recorded in the property `link.import`.

We will decide when and where to insert it.

In the following example, the `<link rel="import" href="import.html">` code imports the `import.html` document and, after downloading it, calls the `load()` function. This function selects interesting parts of the loaded document through `link.import.querySelector('custom-tag')` and appends them into the current `index.html` file, as shown in the following example code:

```
<script>
  function load() {
    var element = link.import.querySelector('custom-tag')
    document.body.appendChild(element);
  };
</script>
<link rel="import" id="link" onload="load()" href="import.html">
```

The `import.html` file has elements and script that "revives" `import.html`, as shown here:

```
<!DOCTYPE HTML>
<html>
<body>
  <custom-tag id="counter">0</custom-tag>
  <script>
    var document = document.currentScript.ownerDocument;
    var counter = localDocument.getElementById('counter');

    var counterId = setInterval(function() {
      counter.innerHTML++;
    }, 1000);
  </script>
</body>
</html>
```

Here are some important details:

- After downloading all of the scripts in the imported `import.html` file, our main HTML file will execute the imported script so that the timer and other variables would be the global variables of the page.
- The variable document is a document main page. For access to imported documents that are inside the current `import.html` document, you can get it as `document.currentScript.ownerDocument`.
- The timer function in the imported document begins immediately. The new document comes alive immediately after being loaded, though the transfer of nodes in the main document cannot be seen.

In this example, the main document is controlling the imported document, but the imported document can control itself and use `document.body.appendChild(timer)` to append itself inside the parent document. In this way, we don't need the `onload` event.

Web components

Imports are created as a part of the web components platform.

It is assumed that the main documents can import all the HTML, JS, and CSS elements and then use them.

Here's an example:

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

<paper-button>...</paper-button>
<paper-radio-button>...</paper-radio-button>
```

Reusing an imported document

Reimporting the same URL uses an existing document.

If the file (for example, `lib.html`) is imported twice, CSS and scripts are merged and executed exactly once.

This can be helpful in not loading the same file many times. We can use `lib.html` to manipulate imports, subimports, and so on and can connect without any fear many times.

Here's an example:

- The main `index.html` file connects documents
- The `paper-button.html` file connects `lib.html`
- The `paper-radio-button.html` file also uses `lib.html`

The `lib.html` file will then be connected only once. This allows you to not be afraid of too much duplication of libraries; it is used to describe a variety of components.

So, the `<link rel="import">` tag allows you to connect to the page of any document, in which:

- The scripts and styles of the page are shared.
- The imported DOM is available from the outside as `link.import`, so you can catch the imported DOM, but you can also get the owner document (the document that imports you) with `document.currentScript.ownerDocument`. So you have access from the main document to the imported document and vice versa.
- Imports can contain other imports.
- If a URL is reimported, it connects ready documents without reexecuting the scripts in them and it avoids the duplication of using a library in a variety of places.

Now, we are finished with features of web components . In the next subtopic, we will discuss styles and selectors in web components. Stay with me and change the world!

Styles and selectors

Shadow DOM uses the standard styling specifications of CSS scoping. You can check the specifications at `https://drafts.csswg.org/css-scoping/`.

Default styles in the Shadow DOM apply only to its contents.

Consider the following example:

```
<p>Once upon a time,</p>
<p id="text">we are learned about Web Components</p>

<template id="template">
  <style>
    p {
      color: blue;
    }
  </style>
  <h1><content></content></h1>
  <p>Hello, from Shadow Root!</p>
</template>

<script>
  var root = text.createShadowRoot();
  root.appendChild(template.content.cloneNode(true));
</script>
```

When you open the document, the blue color will apply to only the `<p>` element inside the template. Let me note that as the color of the element, which is located directly in the Shadow DOM, and the items that appear in the Shadow DOM use the `<content>` tag, the style will not work; they have their own styles on the outer page.

Exterior styling for Shadow DOM

Although a boundary between primary DOM and Shadow DOM exists, using special selectors, it is possible to pass it.

If you want to stylize the main page or select items within the Shadow DOM, you can use the following selectors:

- `:: shadow`: This selects the Shadow DOM root.

  The selected item does not create a CSS box itself, but it serves as a starting point for further sampling inside the Shadow DOM tree.

  For example, the `#text::shadow>` div selector will find all the first-level divs inside the Shadow DOM with the `#text` ID.

- `>>>`: This is a special kind of CSS selector for all elements of Shadow DOM that completely ignores the boundaries between DOM elements, including nested subelements, which can also be your Shadow DOM.

  For example, the `#text >>>` span finds all spans in Shadow DOM `#text`, but in addition, if there is sub `#text` that has its own Shadow DOM, then it will continue to search for it.

  Here's an example where we have a single `<input type="date">` tag in a Shadow DOM, which also has a Shadow DOM:

```
<style>
  #text::shadow span {
    /* to span within Shadow DOM #elem */
    border-bottom: 1px solid red;
  }
  #text >>> * {
  /* All elements within the Shadow DOM #elem continue in input
[type = date] */
  color: blue;
  }
</style>
<p id="text"></p>
<script>
  var root = text.createShadowRoot();
  root.innerHTML = "<span>Current time:</span> <input
type='date'>";
</script>
```

- In addition, the Shadow DOM has simple CSS inheritance if the property is supported by its defaults.

  In this example, CSS styles for the body inherited the internal elements, including Shadow DOM, as can be seen here:

  ```
  <style>
    body {
      color: blur;
      font-weight: bold;
    }
  </style>
  <p id="text"></p>
  <script>
    text.createShadowRoot().innerHTML = "<span>Hello,
      Yerevan!</span>";
  </script>
  ```

  The inner span becomes blue and bold.

Styling depending on the host

The following selections allow the Shadow DOM on the inside to select an external element (`member-owner`):

- `:host` selects a host in which the Shadow DOM lives.

  `:host` is selected in the context of the Shadow DOM. That is, the access is not an external element but rather to the root of the current Shadow DOM. After `:host`, we can specify the selectors and styles to be applied if the owner meets a particular condition:

  ```
  <style>
    :host > div {
      background-color: teal;
    }
  </style>
  ```

  This selector works for the first level `<p>` inside the Shadow DOM.

- The `:host` (the host selector) selects the host if it matches the selector.

  This selector is used to host styling from "within", depending on the classes and attributes. It is great to add a simple `: host`; here's an example:

  ```
  :host > div {
      background-color: teal;
  }
  :host(.underline) p {
      text-decoration: underline;
  }
  ```

  In this example, the divs will have a background color teal, but if the master component has an `underline` class, all the paragraphs will have an `underline` decoration.

- The `:host-context` (master selector) selects a host if any of the parents meets the selector; here's an example:

  ```
  :host-context(h1) strong {
    /* selector work for strong, and if the owner is inside
      the h3 */
  }
  ```

  This is used for advanced theming. This means you can give a style to the parent of the host element. In this example, it is `<h1>`.

Style to content

The `<content>` tag does not alter the HTML DOM. It describes how and where to show the content inside the Shadow DOM. Therefore, if an item was originally located in the host cell, the external document retains access to it.

It will take styles and selectors, as always.

To access the `<content>` tag from the styles, you can use the `::content` pseudo-class to select the content.

For example, from within the Shadow DOM, the selector `content [select = "h1"] ::content span` finds the element `<content select = "h1">` and its contents will find `<span>`.

In the following example, the selector `::content span` stylizes all `<span>` tags within all `<content>` tags:

```
<style>
  span { border: 1px solid black; }
</style>

<p id="text"><strong>Valar Marghulis!</strong></p>

<template id="template">
  <style>
    ::content strong { color: green; }
  </style>
  <h1><content></content></h1>
  <span>Valar Dohaeris!</span>
</template>

<script>
  text.createShadowRoot().appendChild(
    template.content.cloneNode(true) );
</script>
```

The text inside `<h1>` is green and has a border at the same time, but it is stylized as `<strong>`, which is shown in `<content>`, and the other one, which has Shadow DOM, isn't.

Priority selectors are calculated by the usual rules of specificity. If the style of the page is the same style as the Shadow DOM, it will have more priority and will overwrite the CSS rules of the Shadow DOM, but you can always use `!important` to make the Shadow DOM styles dominant.

So, default styles and selectors of the DOM tree act only on their parents.

The border can be defeated easily. Of course, this is done from parent to Shadow DOM rather than vice versa, as follows:

- Outside of the Shadow DOM, you can select and style elements within the Shadow DOM using selectors `::shadow` and `>>>`
- Inside the Shadow DOM you can stylize not only the native content of the Shadow DOM but also the nodes that are displayed in `<content>`
- You can also set the style depending on the host using the `::host` and `::host-context` selectors, but you can't stylize the arbitrary tags inside the host

Detailed steps to download the code bundle are mentioned in the Preface of this book. Have a look.

The code bundle for the app in *Chapter 5*, *First Application with Polymer*, is hosted on GitHub at `https://github.com/AKHXtern/poly`. The code bundle for *Chapter 1*, *Web Components*, is hosted on GitHub at `https://github.com/AKHXtern/polymer-book-examples`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Summary

Here we are, finishing with the first chapter. As we haven't covered the concept of web components just yet, we will discuss Polymer in *Chapter 3*, *Introduction to Polymer*, the favorite framework of web components. It has a lot of components and features that use web components and even make it super powerful.

In the next chapter, we will cover the concepts of Material Design by Google and consider how to create an awesome app with these design concepts.

While we are here, let me mention two points about web components. First, you can make components, make elements with Shadow DOM in your page, import other HTML and make a lot of templates. When I started to write this book, modern browsers had problems supporting web components, but it's okay now! You can use them for sure. Second, please don't use any libraries to make web components. For example, jQuery has problems with the Shadow DOM components. It's always better to use JavaScript instead.

See you in the next chapter!

2
Material Design

In this chapter, you will learn about the concepts of Material Design. The main target of this chapter is to know more about creating an application using Material Design animations, styles, and layouts.

We will cover the following topics:

- Material Design: to the moon and back
- Tactile surface
- Publishing design
- Meaningful animation
- Adaptive design

Let's start by talking about what Material Design is and how it was born.

What is Material Design?

Material Design is a design language and style of Google released on June 25, 2014. It was presented at Google I/O by Google's vice president of design, Matias Duarte. Initially, it was called internally code-named **Quantum Paper**. The basic metaphor for Material Design is flat paper in three-dimensional space.

Matias Duarte, in an interview with The Verge, talked about the basic principles of the new design, called Material Design. This is a completely different approach that provides a unified set of rules, from the types of software to finishing functions. Though the design seems a little strange, you need to get used to it.

The design team at Google felt the need to come up with a uniform appearance and functionality of the software, which could be applied to all products: Android, Chrome OS, and web services. Rather than starting with the development of the color palette, they began with a question: "What is software?". Their goal was to create a visual language that synthesizes the classic principles of good design.

At the end of this chapter, we will take a look at Material Design Lite, created by Addy Osmani. Curiously, MDL—a standalone project—is not tied to Polymer. However, at the same time, Google says that the library is a modern interpretation of Polymer Paper Elements.

Why do we need Material Design?

Material Design serves two purposes: the unification of the many products of a company and of APIs for Android. Google decided to be more understandable and international; interface objects must have an analog metaphor in the real world. Paper was the metaphor.

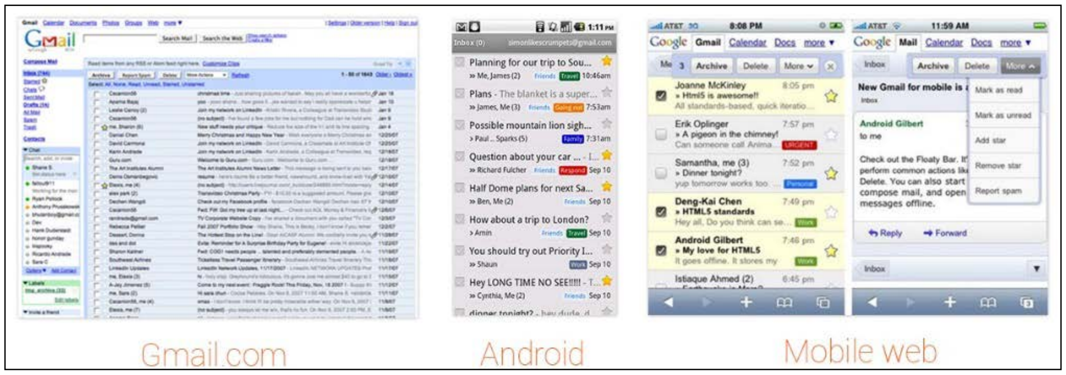
Thin and flat but located in three-dimensional space, this design style has shade, speed, and acceleration. The principles of Quantum Paper are not the same as real life; it obeys the laws of physics. However, it has magical properties. This helps the user show the principles of software as a transition from one to another state. Animations don't just enliven the interface but also show the user what is happening.

Material Design – to the moon and back!

Do you remember the first products of Google? They looked really bad! Not even a single product on their other platforms looked alike.

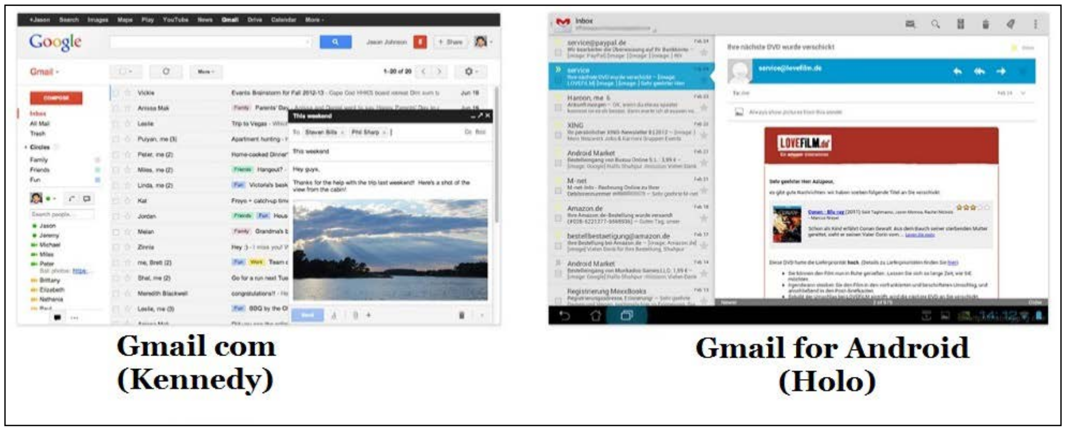
Everything began to change in 2011, when Google started to work hard on the unification of the visual part of the ecosystem of its products and called it Project Kennedy.

So, let me show you how the first Gmail design from Google looked. It was very difficult to use and painful to look at. Take a look at the following figure:



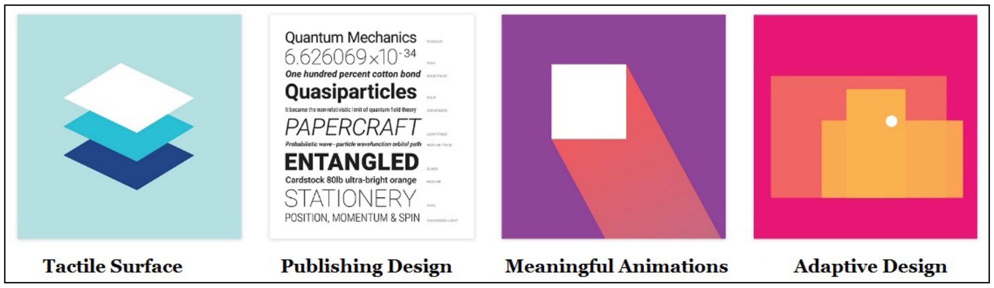
At first, they started to work on the web design of their products and then the mobile products. At the same time, there was a separate work on the design of Android, Holo, which replaced the interfaces of old Android.

However, there was one problem; Holo was still different from Project Kennedy, as shown by the following figure:



Every time Google changed the design, users had to adjust to the new interface switching and get used to the appearance, position controls, and so on. That's why the designers of Google decided to solve this problem once and for all.

Material Design has four principles, as shown by the following figure:

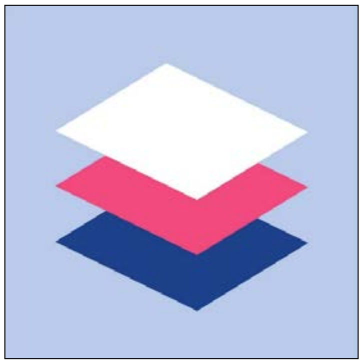


- **Tactile surface**: The Material Design interface consists of tangible layers of so-called "digital paper". These layers are arranged at different heights and cast shadows on each other, which helps users better understand the anatomy of the interface and the principle of interaction with them.
- **Publishing design**: If we assume that the layers are pieces of "digital paper", with regard to the "digital ink" (everything that is displayed on the "digital paper"), Material Design uses the traditional approach of graphic design, such as in magazines and posters.

- **Meaningful animation**: In the real world, things do not arise out of nowhere and disappear into nowhere; that only happens in the movies. Therefore, in Material Design, we always think about how to use animation in layers and "digital ink" to give users tips about the interface.
- **Adaptive design**: This is about how we apply the previous three concepts on different devices with different resolutions and screen sizes.

Tactile surface

Let's start with tactile surfaces. These are the pieces of "digital paper", which, unlike ordinary paper, have a superpower: the ability to stretch to connect and change their shape. Otherwise they act in full accordance with the laws of physics.

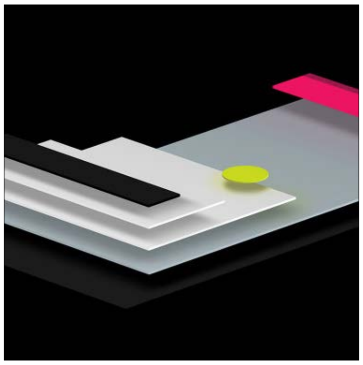


What is the surface? Basically it is a "container" with a shadow and nothing more. However, this is enough to distinguish one object from another and to show how they are positioned in relation to each other. The Material Design philosophy strives for simplicity and a "clean" design.

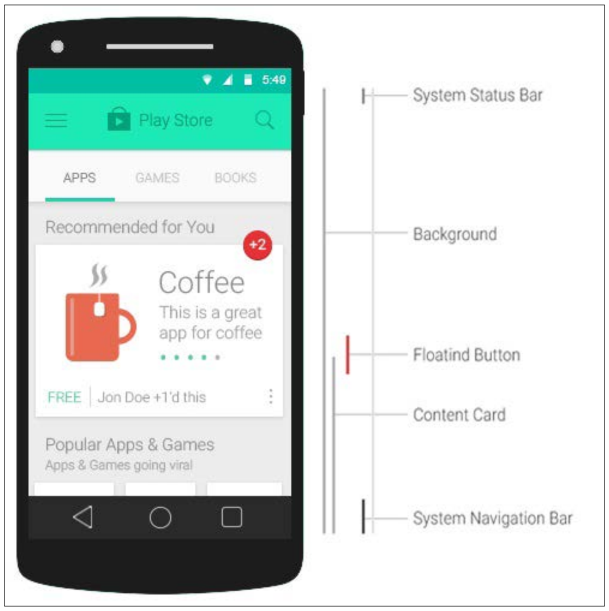
There is no need to go too far and use textures for the image gradients of light and shade. There is also no need to give it the visual properties of a skin like my grandmother's apartment door; a neat shadow can express a lot. But each surface has its surface height—which is the location on the *Z* axis—and each of the surfaces casts a shadow on the lower ones, as in the real world.

Depth

In the traditional "flat design", we avoid such shadows as this can bring manifestations of volume, but they perform an important function, that of noting structure and the hierarchy of elements on the screen. For example, if the rise of the element is greater, its shadow is longer. This increased depth of focus helps the user do the critically important things gracefully.



Depth also gives hints by interaction. In the following figure, as the user scrolls, a green panel is stretched at the top and adds some shadow on the "white paper". This shows that the panel is higher than the "white paper" because, during scrolling, it casts a shadow on it:



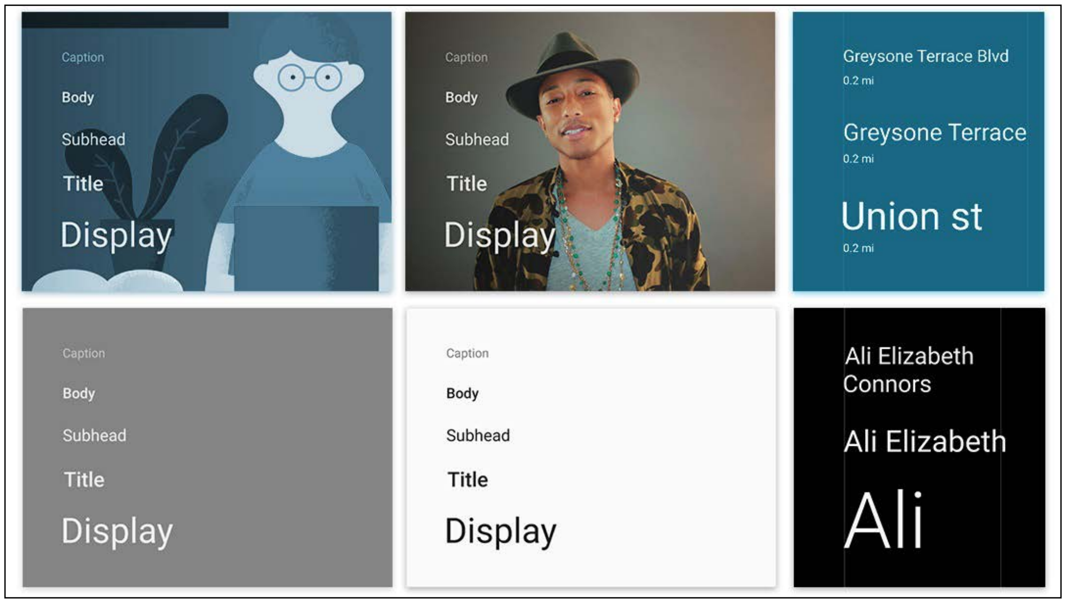
It is important to note that depth is the "bottom". It is assumed that it is limited by the thickness of the mobile device. It is important to note that the depth is the back end of your smartphone. It means that if your smartphone's width is about *X* cm, you should make all the cards to have a depth of *X* cms.

Here are some thoughts on the concept of depth in Material Design:

- **The depth should be meaningful**: Ask yourself, "Why is it so and not otherwise?" If the answer is no, it makes sense to look for another solution.
- **Take care of the logistics**: Floating buttons, toolbars, and dialog boxes are located at a certain height. Sometimes, they need to move along the *Z* axis to avoid collision when something happens. With this choreography, it's necessary to be very careful.
- **Do not overuse the button**: Floating buttons are a very distinctive element. Many believe that it is necessary to add a floating button to the interface to make it Material Design. However, it should only be used for key actions within your application. If you need to close some window or confirm an action, it is not necessary to use a floating button. To do this, there are other elements.
- **Not all elements should be on the card**: If any object has many forms and it contains a lot of different content, the card suits the object. If not, maybe it is better to use plain text or text list.
- **Use the disclosure of lists**: This is an underrated pattern, but it is quite necessary in Material Design.

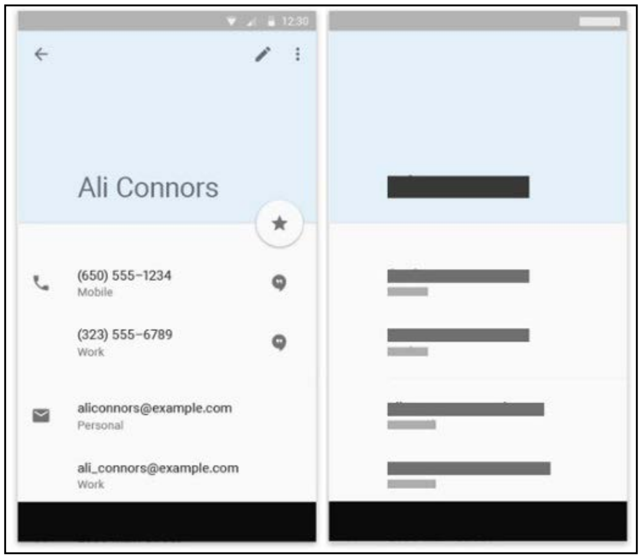
Publishing design

Every surface in Material Design is called "digital paper". Everything that is placed on it—text, images, and icons—is marked as "digital ink". Material Design uses classic print design principles in the design of interfaces.



Elegant typography

In print design, typography plays a crucial role. Pick up any magazine and you will notice that the typography performs two important functions. Firstly, the selection and composition of the font style is one of the important things for a brand. Secondly, typography defines the structure of the content.



Ali Connors

(650) 555-1234
Mobile

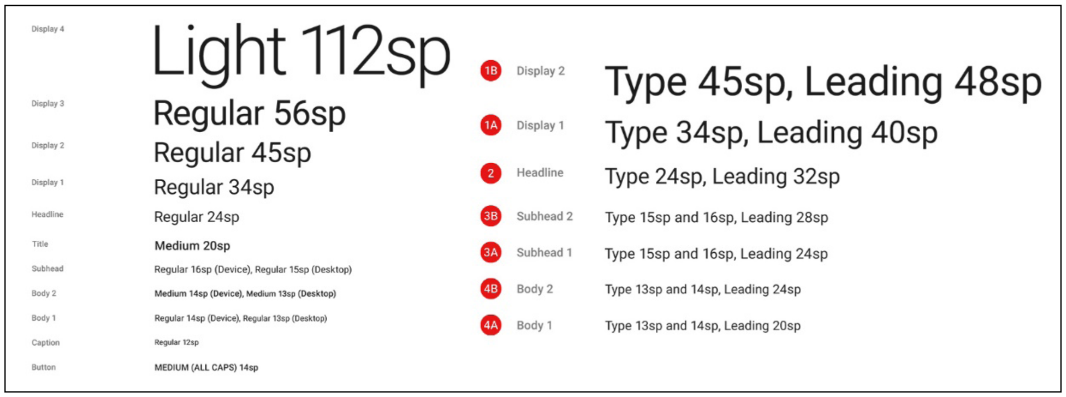
(323) 555-6789
Work

aliconnors@example.com
Personal

ali_connors@example.com
Work

Font size

The site `https://material.google.com/style/typography.html` has a standard palette of fonts that you can safely use. This palette uses the Roboto font, but you can replace your corporate font to support your brand. It is important to test everything carefully, as, on different devices, font rendering works in different ways. Usually OTF fonts work better than TTF.

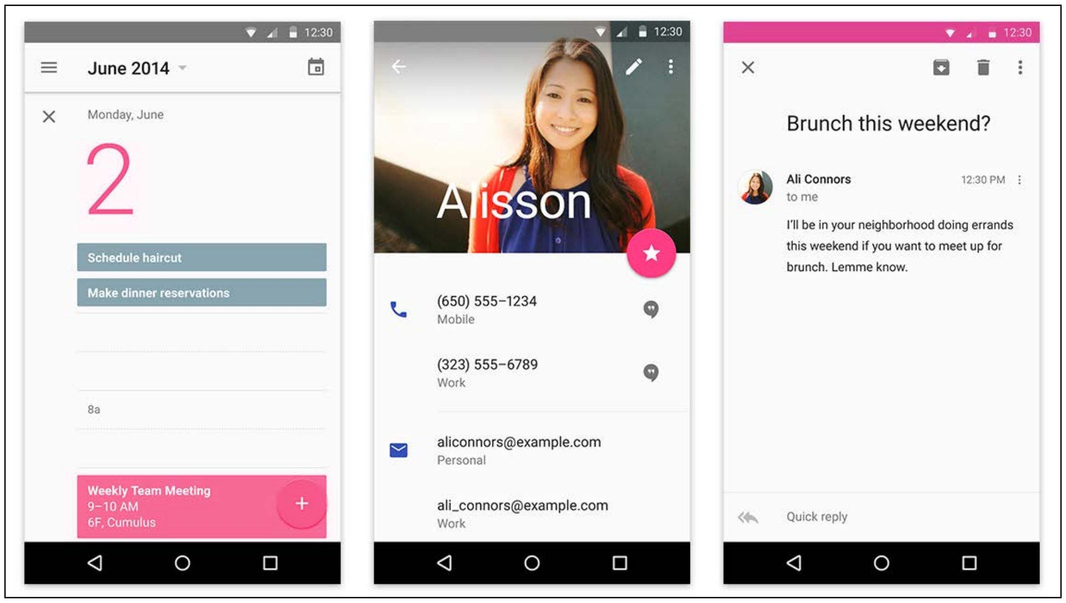


Display 4	Light 112sp
Display 3	Regular 56sp
Display 2	Regular 45sp
Display 1	Regular 34sp
Headline	Regular 24sp
Title	Medium 20sp
Subhead	Regular 16sp (Device), Regular 15sp (Desktop)
Body 2	Medium 14sp (Device), Medium 13sp (Desktop)
Body 1	Regular 14sp (Device), Regular 13sp (Desktop)
Caption	Regular 12sp
Button	MEDIUM (ALL CAPS) 14sp

1B	Display 2	Type 45sp, Leading 48sp
1A	Display 1	Type 34sp, Leading 40sp
2	Headline	Type 24sp, Leading 32sp
3B	Subhead 2	Type 15sp and 16sp, Leading 28sp
3A	Subhead 1	Type 15sp and 16sp, Leading 24sp
4B	Body 2	Type 13sp and 14sp, Leading 24sp
4A	Body 1	Type 13sp and 14sp, Leading 20sp

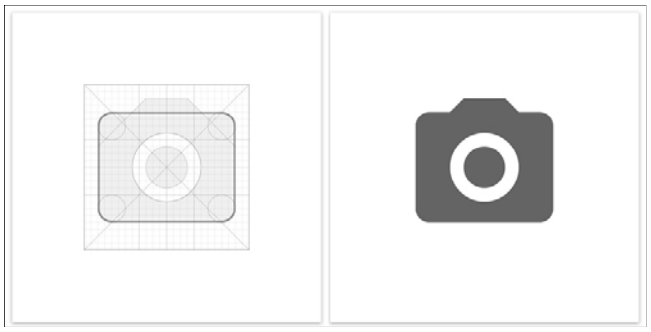
Contrast typography

Another principle from the world of printing that gets along well in Material Design typography is contrast; for example, a marked contrast between the size of the header and text. It's beautiful and highlights the point well, as can be seen from the following figure:



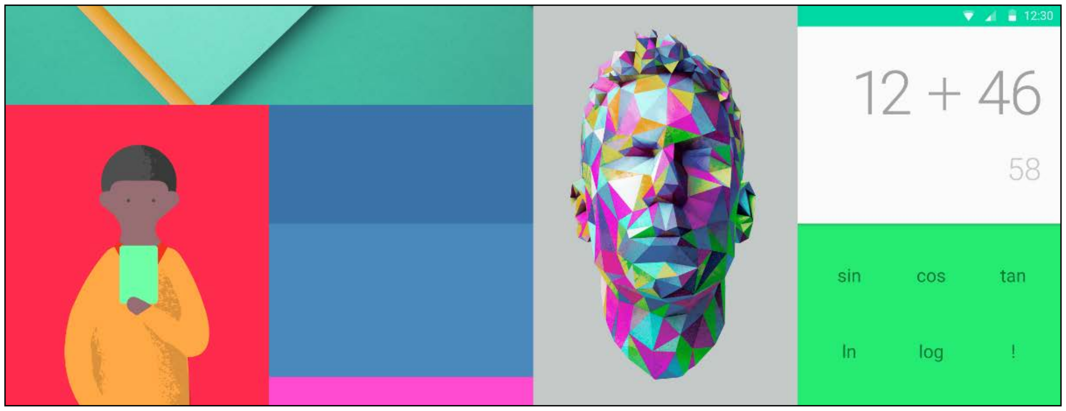
Geometric iconographies

If we talk about iconographies, simple icons have been used on Android for some time, but, in Material Design, they are even easier and friendlier. At an informal resource (`http://materialdesignicons.com/`), designers can find icons for their own purposes and also the opportunity to contribute.



Colors

The design of the interface color is an important means of expression. In the earlier Android versions, color was somewhat secondary, but now it plays a primary role. The standard palette color of Material Design consists of main and accent colors.



The main color is used for large areas, such as the action and status bars, and it is painted darker than the accent color. Brighter accentual color is used in controls, buttons, strips, indicators, and so on. Accent color is designed to attract users' attention to key elements such as floating buttons.

Beautiful photos

Finally, as in printed design, in Material Design, using photographs and images in all their glory is encouraged. Pictures mainly placed without frames often "bleed". Even the status bar is especially made transparent so as not to interfere. In addition, each drop of "digital ink" has a feature and there is nothing just for decoration.

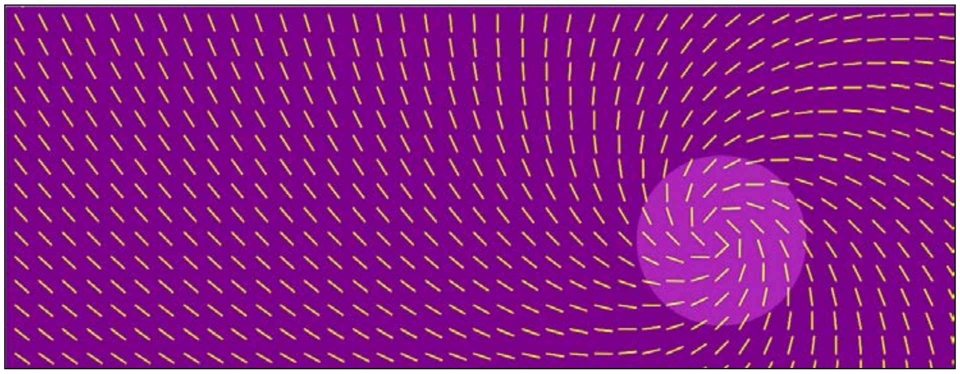
I would like to recommend the kind of images for you to use in your design:

- **Create your own with pleasure**: Google has a strong position here. It allows you to choose bright photos with bright colors for the brand, but don't hesitate to create your own brand. You can use your own colors and images. Colors can be chosen from the corporate brand book or from the logo.
- **Do not forget about paddings and spaces**: The basic grid with 8 dp margin from the top and 72 dp margin from the left is almost a rule. So use this rule to make the content feel good and free.
- **Expressive photos make the mood**: Photos and illustrations as a means of expression are our choice.

Meaningful animations

In the real world, objects cannot just appear out of nowhere and disappear into nowhere. It would cause confusion and put people on a dead end. Therefore, in Material Design, meaningful animation is used to show what just happened.

For a user, the main attention is focused on actions. The interaction with the design controls the user and not vice versa. All the action takes place in a single environment with interactive objects without interrupting the sequence of transition from one medium to another.

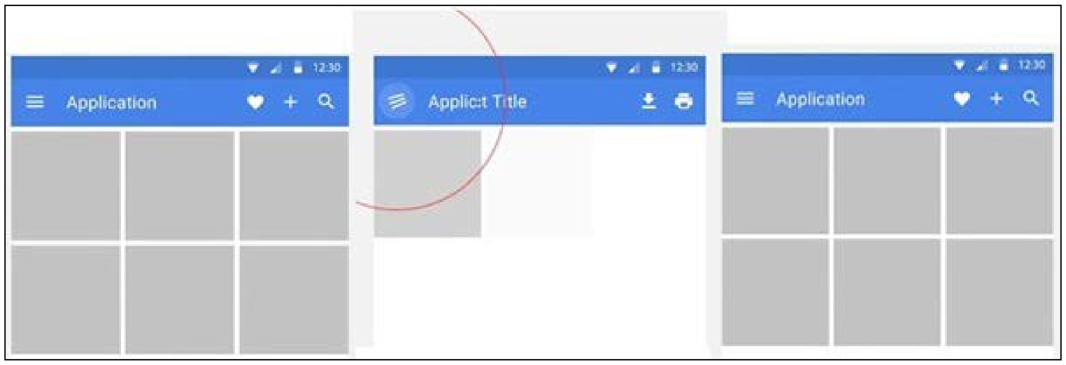


Asymmetry

Since the depth of the interface is limited by the thickness of the device, all the transformation facilities need to be produced on paper. It brings asymmetric transformations to the animation, which means that adjusting the object's width and height should be apart from one another. Otherwise, there will be an illusion of zoom in and zoom out for the viewer, with a very large distance.

Reaction

Another very important principle of animation in Material Design is the reaction of the user to actions. The epicenter of the changes in the interface should be touching the screen of the device. An example is the wave that comes and goes from the contact point of a finger. This effect is realized without difficulty in Android L, as shown in the following figure:



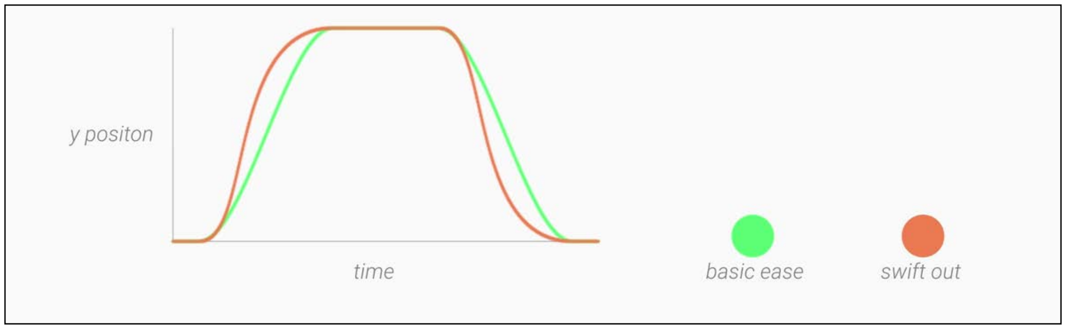
Micro animations

In the applications, we can animate each element, whether it is the transitions between the content or actions of small icons. Every detail of the application is important and micro animations add to the application's detail and responsiveness to every user action.

Clarity and sharpness

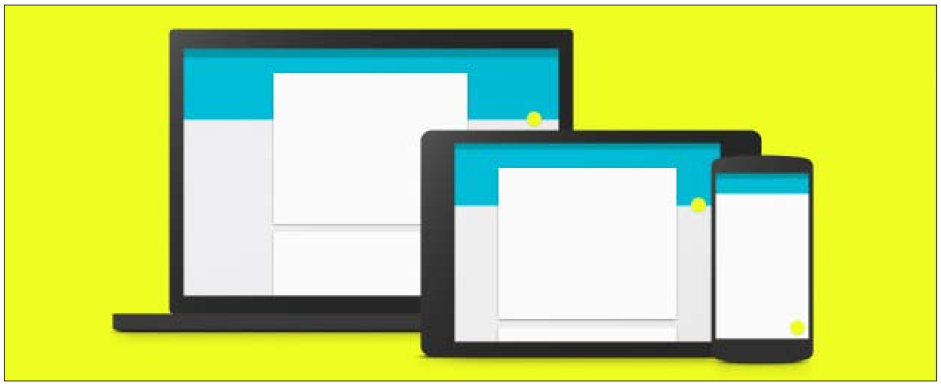
The last key principle of the animation is that the movements must be rapid and clear. In contrast to the banal acceleration at the beginning and deceleration at the end, the curve in Material Design animation is more natural and interesting. Objects react faster and reach the target state sharper. As a result, users need to wait less (this "less" is annoying too). At the same time, where the object is already out of the scope of the interest of users, it allows itself to behave more naturally. Here are some tips:

- **Do not leave animations until the end**: Do not leave the animation until the very end; it can serve as a key factor in user experience and it is necessary to think in advance.
- **Know the measure**: Too much animation is bad. Remember that the animations must be meaningful.



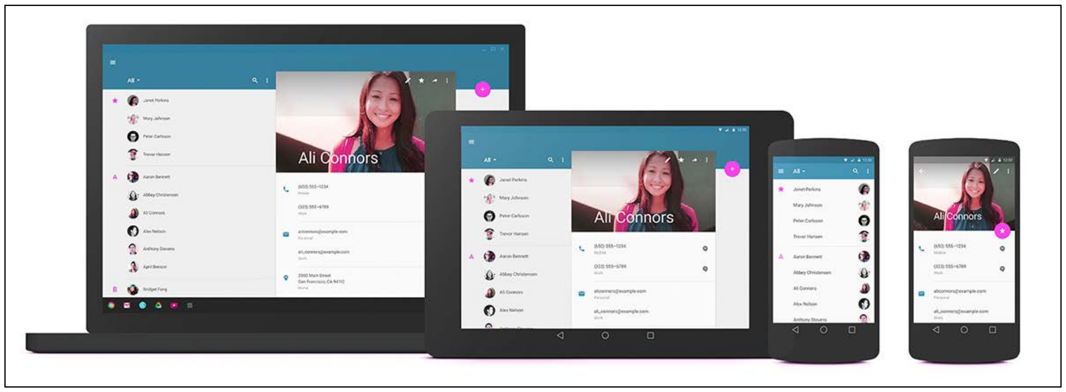
Adaptive design

The last major aspect of Material Design is the concept of adaptive design. It allows us to apply all three of the concepts mentioned before on different devices and screens with different form factors.



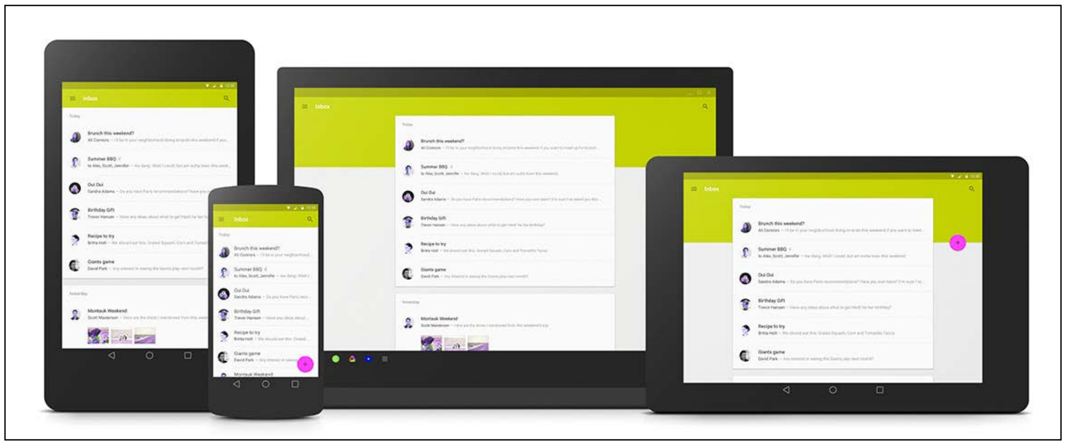
From the general to the particular

The most common technique is reducing the amount of information displayed onscreen along with a reduction of the screen. If, on the big screen, we can afford to show a list and detailed information on the selected item, then on smartphone displays, a separate screen may be needed for the list and details. In the case of tablets, the app bar can sometimes be increased to cope with a little excess space.



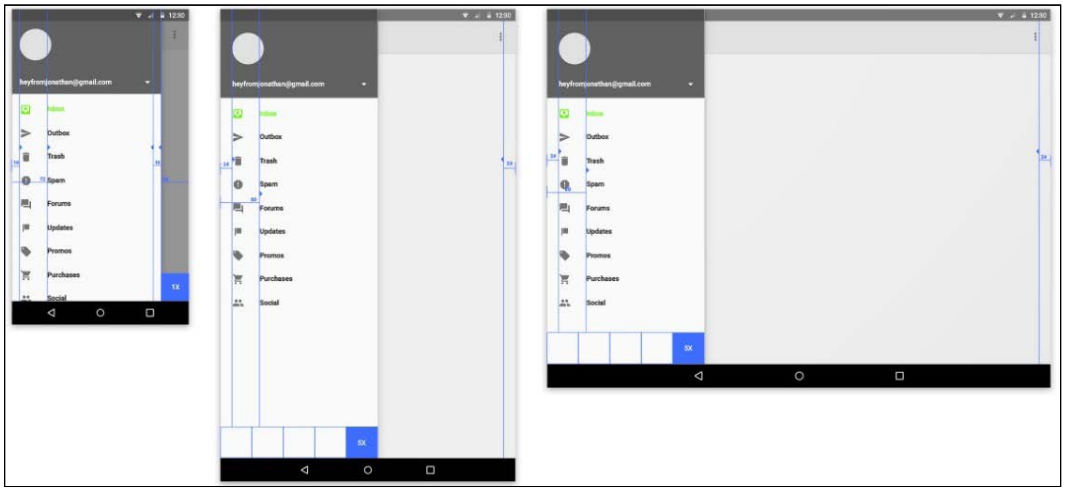
Padding

Placing the content with units makes it much easier to work with the free space on big screens. We know the content of each unit; we need to understand how broad it can be so as not to lose readability and how narrow it can be so that it doesn't look too crowded. On a wide screen, units are stretched to their limit of readability and then padding is added on the edges, which can be quite large. They can be filled with floating colored buttons and drawers.



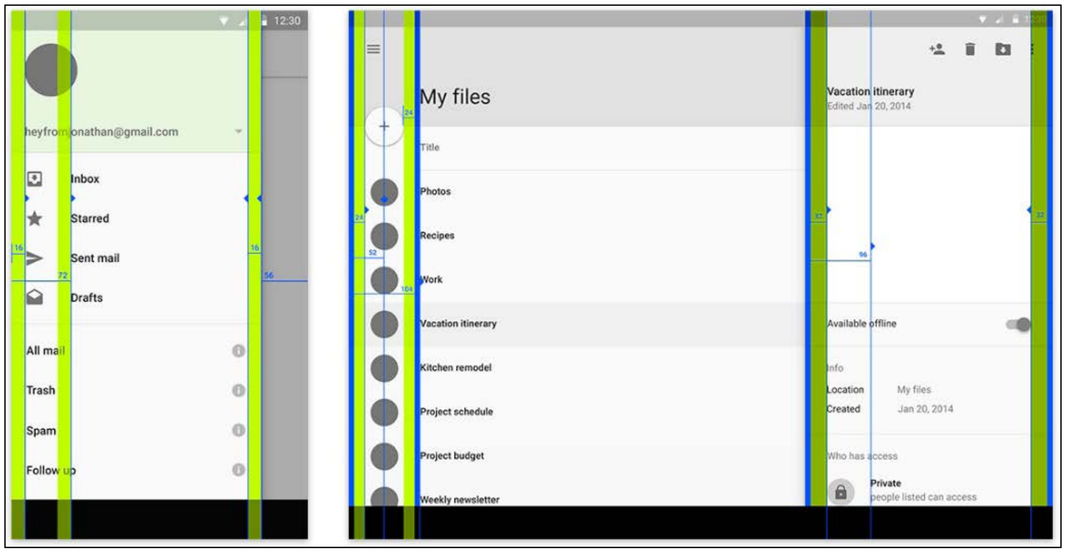
Wireframes

You can find ideas for the organization of space and padding for different screens at `https://material.google.com/layout/structure.html#structure-whiteframes`. This is a great place to understand the overall meaning and then go on with your own experiments.



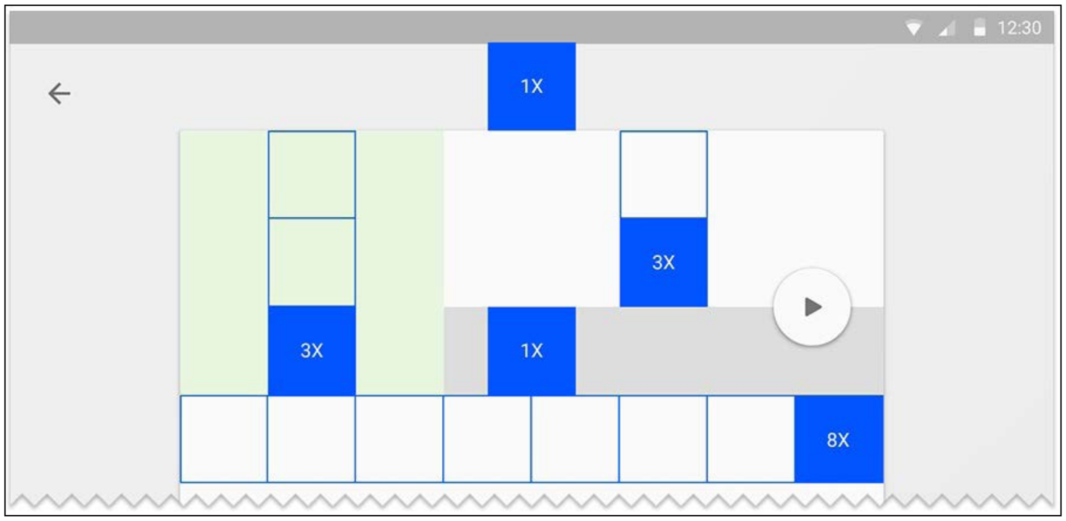
Guides

The Material Design guide asks us to indent for "ink" on separate sheets of "paper." On a smartphone, we have one sheet and one big left padding and, on the tablet, we have two of them; in both cases, there is padding. It is important that the indent of these two form factors be different. On the tablet, the indent is 80 dp and on smartphone it is 72 dp. Padding from the edges of the screen is also different, as shown by the following figure:



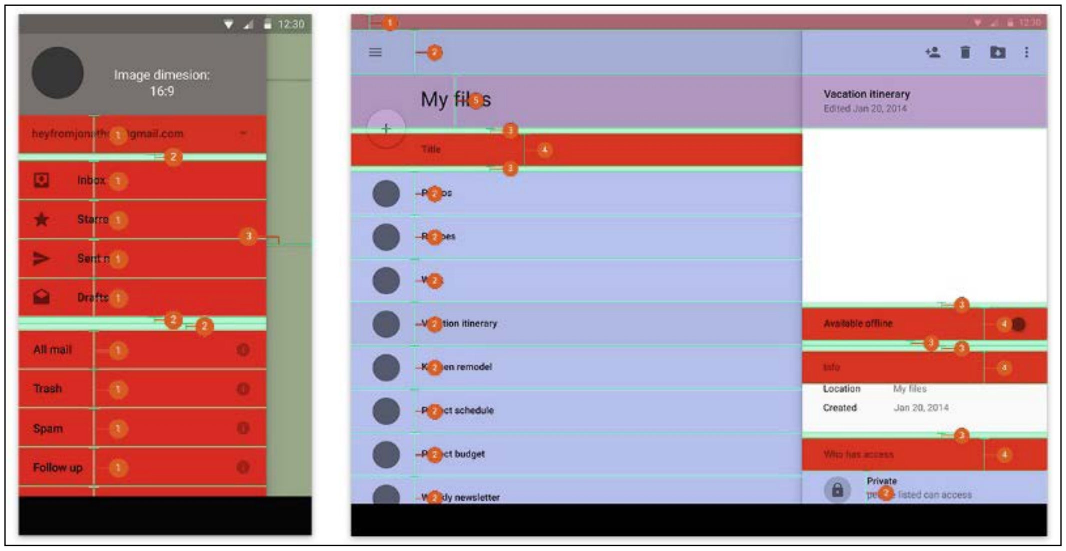
Sizes

It is recommended that you take multiple proportions of all the elements. In particular, selecting the size of the app bar is much more convenient if you do it in multiples: 1x, 2x, 3x, and so on. On smartphones and tablets, this size is different, but the app adapts with no problem.



Blocks

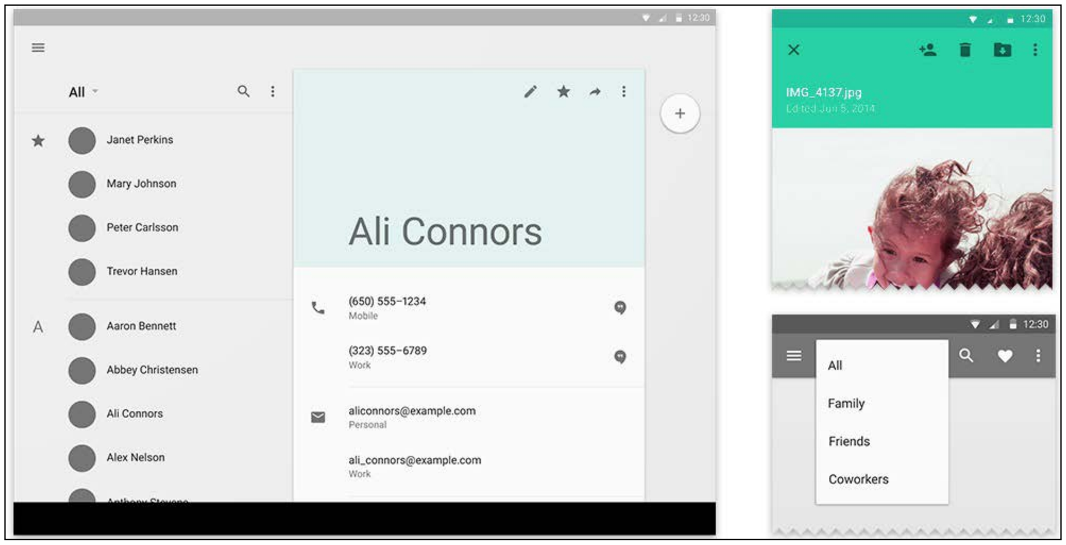
Thinking about all the blocks can be helpful. If you specify a module with blocks of height about 8 dp, it creates a great visual rhythm and it will be easier to make decisions when there is a need to add some details in the blocks. Visit `https://material.google.com/resources/layout-templates.html#layout-templates-desktop` to find material on whiteframes.



Toolbars

The action bar is one of the most important parts of the interface. In it are placed a header and the navigation and action buttons. The Android Lollipop action bar is transformed from a constrained strip on top to a full widget—a beautiful and functional control unit application. It made putting a lot of functional elements on the toolbar possible, which we couldn't even dream about previously. They are:

- Input fields and forms
- The floating button of the main action
- The advanced navigation toolbar, which is hidden; however, in the following screenshots, we can see quite a few functional widgets
- Toolbars that can be conveniently controlled as necessary



Here are some tips that can help you in the future:

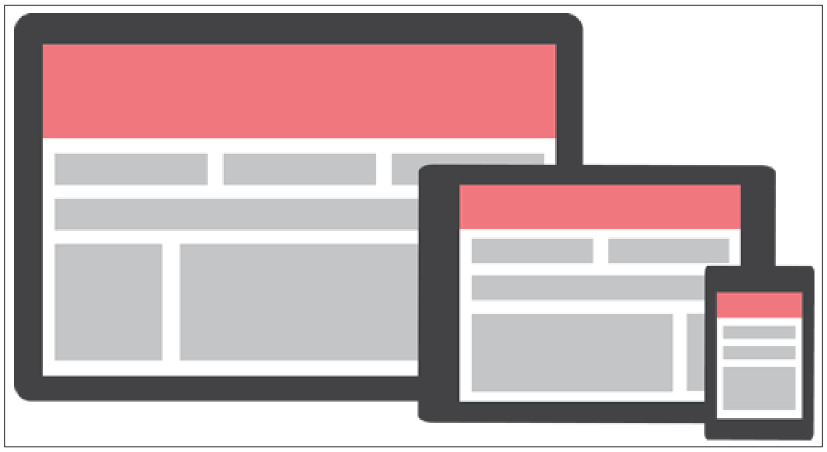
- **You don't always need a navigation drawer**: Very often, Google uses sliding navigation in their applications, as you can note in different examples. However, Google has a lot of problems that can be solved with the drawer's help: the help button, login/logout, user information, and so on. If you have similar problems, you can use a drawer, but if you are making a simple tool, it is not necessary.
- **Go ahead and try witty toolbars**: The ability to dynamically resize the toolbar, making it double and triple its size, is very cool and comfortable. Most designers are afraid to try this and choose one size once and for all, but you can be a bit bolder!
- **Do not keep the lower corner of the ghetto for the floating button**: Floating buttons can be anywhere: at the bottom, at the top, to the right, or to the left. Of course, it is more convenient to reach when it is in a corner, but this is not the only option. The button can be moved from place to place depending on your objective.
- **Material Design for smartphones and tablets for vertical and horizontal views**: The resolution of Android devices is large and it is not easy to deal with it. But the truth is that there are more than 1 billion Android users and it's important to provide them with a good and beautiful product. Material Design solves the resolution problems.

[This is Material Design! Do not be afraid of experimenting and making mistakes. Do not get hung up on copying existing solutions. Try harder!]

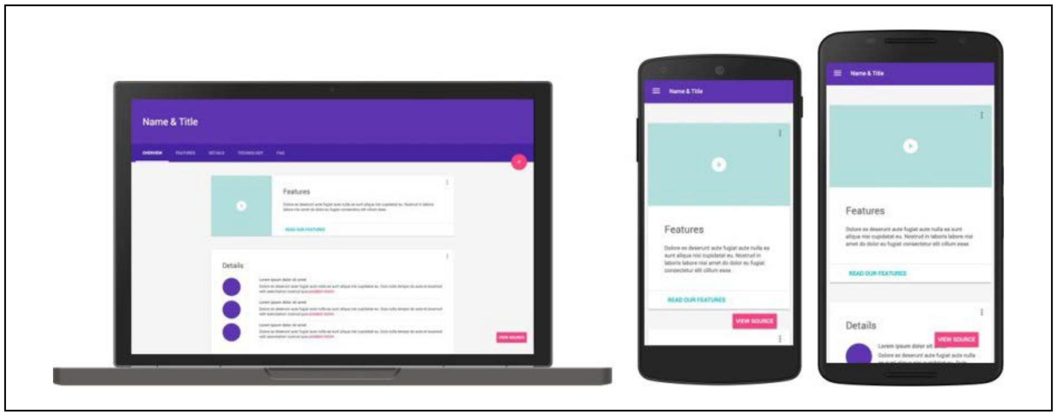
Other tools

Before finishing this chapter and moving ahead with the next one, I would like to introduce you to other great tools:

- **Materialize CSS** (`http://materializecss.com/`): This is an adaptive grid, which is made in the style of Material Design and the framework also consists of a few cool things.



- **Material Design Lite** (`https://getmdl.io/`): Google has reached a new level in the promotion of Material Design. An open source project named Material Design Lite has been created by Addy Osmani from Google and released at version 1.0.0. Curiously, MDL, a standalone project, is not tied to the framework's Polymer. At the same time, Google has said that the library is a modern interpretation of Polymer Paper Elements.



Summary

We have come to the end of this chapter, so let's review it. We discussed how Material Design was born and discussed the aspects of Material Design. We took a look at how to create beautiful applications with meaningful animation and adaptive design.

This was a very important chapter for us because, in this chapter, you learned how to create beautiful applications using Material Design principles (this will help you a lot in the future).

Let's now move ahead to the next chapter, in which we will talk about all the aspects of Polymer, from its installation to its elements and how they work.

3
Introduction to Polymer

In the previous chapters, we covered web components and Material Design fundamental concepts in depth. We talked about custom elements and discussed how to create a custom tag or template, how to use Shadow DOM and manipulate it, and how to import HTML documents into your document. You also learned about Material Design concepts, which you will use in your future apps. Finally, we can now move ahead and get introduced to Polymer and its features.

In this chapter, we will discuss the following topics:

- What is Polymer?
- Getting the code
- Working with Polymer
- Registering elements and the life cycle
- Declared properties
- Local DOM
- Data binding
- Behaviors
- Events
- Styling

What is Polymer?

Polymer is a library for the creation and use of web components. Not all browsers support the features of web components (at this time), but with the help of the Polymer library from Google, this problem can be fixed. Polymer provides a set of polyfills, which let us use web components with all browsers in the world.

- You can create your custom HTML element, give it a name, and share it with other developers
- Every single element has its own template
- It lets you use UI elements already created for your projects

You may ask if Polymer is a web component or maybe a collection of elements? My answer is no! Polymer is created according to the standards of web components and it provides some Material Design elements and lets you create your own.

Now, if you're ready for adventures with Polymer, press the red **FAB** (**floating action button**) in the lower-right corner of the page. Oops! I forgot that this is a book. Ok, let's move ahead and get Polymer code from the Internet.

Downloading the code

There are three ways to download the code.

Bower

This is the most recommended way to install Polymer. Bower is the package manager used to manage scripting dependencies. You can install it from the official website at `http://bower.io/`.

If you're not familiar with Bower, you can read the documentation from the official website. If you have worked with it, you will probably know that we need the `bower.json` file. Just run the following command:

```
bower init
```

After some questions (you can ignore them), it will create a `bower.json` file for your project and then allow you to install your dependencies. For Polymer, we need to run the following command:

```
bower install --save Polymer/polymer
```

At the time of writing this book, the version of Polymer is 1.1.0, but it could change by the time you read this book; so, just change the version that it belongs to.

The `--save` keyword adds packages as a dependency in the `bower.json` file. It is useful when you need to upgrade your packages.

By the way, Bower creates a `bower_components/` folder and saves all the dependencies you installed there.

Downloading the ZIP file

You can download the ZIP file from the official Polymer website at `http://polymer-project.org`.

The bad bit about installing with `.zip` is that you can't update your dependencies later. These are static files and nothing will be changed from there automatically. However, with Bower, you can update your dependencies by running the following command:

```
bower update
```

The new version of polyfills is available now; you can take a look at it by installing the web components dependency from Bower. It will download the web-components-lite version as well, but this doesn't contain Shadow DOM, which supports polyfill.

This is much faster and has better performance, but I recommend you use the minified version of web components. This is because, with this version, all browsers in the world are starting to support web components and it is more important for our developers to work with multiplatform devices and software. However, on the other hand, the web-components-lite version is more important for designers as they want to transform the world into a simpler, faster place and, in my case, this is very important when creating a product in which UX would be awesome, faster, and beautiful. It is very important for UX that the user takes a very short time to perform an action.

In the next chapter, I will talk about the Polymer Starter Kit, which is the "getting started" template for your applications. It includes application layouts and lots of tools for testing and deployment. We will create an example with this kit and take a look at the progress in installing the kit.

The Polymer team has a repository on GitHub in which you can contribute code for the Polymer project. I have been contributing since December, 2014, and, in my case, it is great to know that I have a part in this project. Just hack something special for the project or submit a pull request (and for contribution, we recommend using Bower to install).

Cloning from Git

You can get the code from the Polymer repo. This is also not the recommended way and the reason is the same; you can't update your dependencies from anywhere. So, why choose the hard way instead of the easy and fast one?

Working with Polymer

Once you have Polymer code, we can move ahead to understanding how Polymer works and how to create something with it. But first, we need to create the `index.html` file in which we will create our Polymer app.

Just create an `index.html` file with a skeleton in the project folder (in the folder with `bower.json`). Then, you just need to include polyfills, this way:

```
<script src="bower_components/webcomponentsjs/
  webcomponents.min.js" ></script>
```

Then, we need to create our first custom Polymer element.

The main objective of Polymer is to make the creation of web components simpler and faster. With Polymer, you can create Polymer custom elements, which is the easier way to create an element than with just WC. Let's take a look at what features we will have by using Polymer custom elements:

- Registering elements
- Life cycle callbacks
- Property observation
- The local DOM template
- Data binding
- Styling
- Events

We will talk about each feature in the future.

/ --- The future starts from here --- */*

Registering elements

In this section, we will register a new element in the browser by calling the Polymer function. Registering a new element is similar to custom element registration in web components. The registering element associates a name with a proto object and you can add anything there, such as properties and methods for your Polymer custom element, but it must contain a dash (`-`) in it—for example, `<polymer-dash>`. I will take a similar example from `https://www.polymer-project.org/1.0/`, but we will talk about it more than the website. So, we will create another `.html` file named `proto-element.html` here, which contains the following:

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

<script>
  // register a new element called proto-element
  Polymer({
    is: "proto-element",
    // add a callback to the element's prototype
    ready: function() {
      this.textContent = "I'm a proto-element. Check out my
prototype!"
    }
  });
</script>
```

Here, we have to import `polymer.html` to register the Polymer element. In the first property, `is`, we will give the element a name and describe what it should look like when you call it. Then, you will see the `ready` method, which is the main function for all Polymer elements, it will run when the element is ready. So, it is similar to jQuery's `$(document).ready`, but for Polymer. Inside the `ready` method, we have an exception, which defines the element content text. So, when we have the rendered version, it will look like just simple text in a view.

Then, we just need to import our `proto-element.html` file into `index.html` using HTML imports from web components. It will look similar to this:

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

This is awesome because we just created our first Polymer custom element! Now, we can use this element whenever and wherever we want to, similarly to this:

```
<!DOCTYPE html>
<html>
  <head>
    <script src="bower_components/webcomponentsjs/webcomponents-lite.
min.js"></script>
    <link rel="import" href="proto-element.html">
  </head>
  <body>
    <proto-element></proto-element>
  </body>
</html>
```

When you create a custom element, Polymer tells the browser to register a new element and returns a constructor of it. Here's an example:

```
// here we are registering the element
CustomElement = Polymer({

  is: 'my-tag',

  // Here is lifecycle callback, we will talk later
  created: function() {
    this.textContent = 'My yummy-dummy custom element!';
  }
});

// Creating an element with function createElement
var el1 = document.createElement('my-tag');

// or with the Polymer constructor:
var el2 = new CustomElement();
```

In this example, we created a Polymer custom element whose name is `my-tag` in two ways. The first way was creating an instance with the `createElement` function and the second one was using a Polymer constructor to create an element. So, you can use either one to register and create an element.

Then, the Polymer function chains the prototype of your custom element in Polymer's base proto. If you want to change something from here, you can't; instead, you need to use behaviors to share some code between elements. However, while we are here, let's take a look at what callbacks Polymer's base life cycle has.

It has standard life cycle callbacks for custom elements to make your life easy. There are four life cycle callbacks and they have the same functions as the custom elements of web components, but with a different (short) name; let's take a look at each one:

- `created`
- `attached`
- `detached`
- `attributeChanged`

There is also a `ready` callback of the Polymer custom element, which is invoked when the element is created and loaded with all its content elements. This is similar for life cycle callbacks, for example, the `created` callback is invoked when the Polymer element is created and so on. Let's consider this in the following example:

```
CustomTag = Polymer({
  is: 'custom-element',
  created: function() {
    console.log(this.localName + ' was created');
  },

  attached: function() {
    console.log(this.localName + ' was attached');
  },

  detached: function() {
    console.log(this.localName + ' was detached');
  },

  attributeChanged: function(name, type) {
    console.log(this.localName + ' attribute ' + name +
      ' is now ' + this.getAttribute(name));
  }

});
```

Here is the order of callback initialization:

1. The `created` callback (the element is created).
2. The local DOM is ready.
3. The `ready` callback.
4. The `factoryImpl` callback.
5. The `attached` callback.

The callbacks are created specially for you. With these callbacks, you can handle the initialization steps of the custom element. Let's move ahead to one more interesting subtopic about declaring the properties of an element.

Declaring element properties

In Polymer, you can set the properties of your custom element. There is a `properties` object created for your element properties. You may ask, what properties will the custom element have? With `properties`, you can configure the element's properties to specify some markup.

Here's an example from the Polymer website about how to set the properties of a custom element:

```
Polymer({
  is: 'x-tag',

  properties: {
    user: String,
    isJuly: Boolean,
    count: {
      type: Number,
      readOnly: true,
      notify: true
    }
  },

  ready: function() {
    this.textContent = 'Hello World, I am a Custom Element!';
  }
});
```

Yeah! You can give properties to an element that will effect it in future. Let's take a look at all default properties that Polymer functions have:

- `type` (Boolean, date, number, string, array, and object): This attribute is used to deserialize from an attribute
- `value` (Boolean, number, string, and function): This attribute is the default value for the property
- `reflectToAttribute` (Boolean): If you set this to `true`, it will reflect on the host node property's value changes
- `readOnly` (Boolean): By setting this to `true`, the property can't be set by data binding or by assignment

- `notify` (Boolean): If the value of the attribute is `true`, two-way data binding is available for this property
- `computed` (string): This method is invoked to calculate the value whenever any of the argument values changes
- `observer` (string): This value is invoked when the property value changes

You can configure default property values here. The values can be a specific value or a function that returns some data for you. Here's an example of how to configure default values:

```
Polymer({

  is: 'user-custom',

  properties: {

    name: {
      type: String,
      value: 'auto'
    },

    hobbies: {
      type: Object,
      notify: true,
      value: function() {}
    }

  }

});
```

I created some default values for Polymer properties here. The `name` property has a specific value `auto`, but the next one, `hobbies`, has a value that returns some data (object or array). Now, let's move ahead to property observers to take a look at how to observe data in the custom element.

Property change observers

With property observers, you can observe the data in the custom element. This means that you can handle the changes of the data and create some functionality for it. For example, let's assume you have a `status` variable in your Polymer proto and you need to handle the status of a user—to know whether he/she is online or not. Let me show you a little code to observe the change with the function, which has the old and new values as arguments, as follows:

```
Polymer({

  is: 'x-tag',

  properties: {
    status: {
  type: Boolean,
      observer: '_statusChanged'
    },
  },

  _statusChanged: function(newValue, oldValue) {
      if(newValue > oldValue) console.log("User is now online!");
  this.toggleClass('isOnline', newValue);
      this.online = newValue;
  }

});
```

In this example, we have the `status` variable, which is `true` if the user is online and `false` otherwise. We observed the data with the function called `_statusChanged`, which has old and new values as arguments, and are using it in the function. But how you can observe the multiple data in the property? Now, I will show you the way to observe the changes of multiple properties using the `observe` array in the Polymer proto, as follows:

```
Polymer({

  is: 'x-video',

  properties: {
    poster: Boolean,
    src: String,
    volume: String
  },

  observers: [
    'updateVideo(poster, src, volume)'
  ],

  updateVideo: function(poster, src, volume) {
    if(volume > 0.5) alert('Be careful, the boom is boom!');
  }

});
```

Next, let's consider how we can add local Polymer DOM elements to our page.

Local Polymer DOM elements

Polymer uses local elements for its UI Material elements because this is easier than the registering version. Yes, we are lazy people!

As in the previous example, we need to create an element `.html` file—for example, `dom-element.html`. Let's take a look at what we have here:

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

<dom-module id="dom-element">

  <template>
    <p>I'm a DOM element. This is my local DOM!</p>
  </template>

  <script>
    Polymer({
      is: "dom-element"
    });
  </script>

</dom-module>
```

You have a new thing here to learn. First of all is the use of the `<dom-module>` element. The `id` attribute of this element is the same as we had in the previous example—the `is` property—which is the name of the new custom element. Polymer clones the content of the template and registers a new Polymer custom element automatically.

In this example, we have declarative and imperative parts or portions. The declarative portion is the `<dom-module>` element, which has `id`, the same value that is Polymer's `is` property, and we have an imperative portion, which is `Polymer({…})`. These two portions can be in the same file or in separate ones. We can say the same about the `<script>` tag; it can be either outside or inside our `<dom-module>` element.

Manipulating nodes inside Polymer elements

When we create a Polymer element, it automatically creates a map of nodes inside its local DOM so that we can manipulate the content and nodes we want with ease. All nodes with IDs are stored on the `this.$` hash by their IDs. This means if you have any `#bombs` element inside your local DOM, you can get the node by calling `this.$.bombs` and then do some cool stuff with it. Nodes use data binding, so the manipulation gets simpler than ever. Let me show you an example from the Polymer website, as follows:

```
<dom-module id="x-custom">
  <template>
    Hello World from <span id="name"></span>!
  </template>
  <script>
    Polymer({
      is: 'x-custom',
      ready: function() {
        this.$.name.textContent = this.name;
      }
    });
  </script>
</dom-module>
```

There is a new feature you can see in the `ready` method; it sets the text content in the span with an ID, `name`. That's it! You can do whatever you want with any node you have in your local DOM.

Hey, did you know that we have our own DOM API? Yes! Polymer provides a custom API for a lot of things. Appending, removing children, adding some attributes, and others are some examples. Let's consider this API.

Here are some APIs to add and remove elements:

- `Polymer.dom(parent).appendChild(node)`
- `Polymer.dom(parent).insertBefore(node, beforeNode)`
- `Polymer.dom(parent).removeChild(node)`
- `Polymer.dom.flush()`

With these calls, you can append/remove some elements from the parent. Here are some other API calls for you.

Polymer DOM selectors:

- `Polymer.dom(parent).childNodes`
- `Polymer.dom(node).parentNode`
- `Polymer.dom(node).firstChild`
- `Polymer.dom(node).lastChild`
- `Polymer.dom(node).firstElementChild`
- `Polymer.dom(node).lastElementChild`
- `Polymer.dom(node).previousSibling`
- `Polymer.dom(node).nextSibling`
- `Polymer.dom(node).textContent`
- `Polymer.dom(node).innerHTML`
- `Polymer.dom(parent).querySelector(selector)`
- `Polymer.dom(parent).querySelectorAll(selector)`

Polymer DOM manipulations:

- `Polymer.dom(contentElement).getDistributedNodes()`
- `Polymer.dom(node).getDestinationInsertionPoints()`

Node mutation APIs:

- `Polymer.dom(node).setAttribute(attribute, value)`
- `Polymer.dom(node).removeAttribute(attribute)`
- `Polymer.dom(node).classList`

Let me show you some examples of how to use `Polymer.dom()`. For example, if you have these lines of the custom element:

```
<template>
  <div id="content">
     <div id="header"></div>
  </div>
</template>
```

You can do something similar to this:

```
var insert = document.createElement('div');
Polymer.dom(this.$.content).insertBefore(insert, this.$.header);
```

This will insert the div element before the header node.

API calls are very useful for us; while working with AngularJS and Polymer, I started to forget about jQuery and others because modern libraries such as Polymer provide almost anything you need for your application.

Data binding in Polymer

Of course Polymer uses data binding. One of the best pluses of Polymer is that you can bind the data from your model to your view. It is the most awesome way to propagate data changes in your element. The binding is similar to that with AngularJS; you just need to write `double-mustache` in a place you want to bind something. Let me show you a simple example of how to bind data in Polymer. I created this local DOM element:

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

<dom-module id="book-tag">

  <template>
    <!-- bind to the "bookName" property -->
    The name of this book is <b>{{bookName}}</b>!
  </template>

  <script>
  Polymer({
    is: "book-tag",
    ready: function() {
      // set the name of the book
```

```
        this.bookName = "Getting started with Polymer";
      }
    });
    </script>

  </dom-module>
```

In this example, we have a property, `bookName`, which binds to the view when the `ready` function calls. Let's move ahead and take a look at binding annotations.

Binding annotations

There are two bracket styles in Polymer data binding, which are:

- Curly brackets (`{{}}`) create two-way data binding. Data flow is not downward, which means you can modify a property from the host.
- Square brackets (`[[]]`) create one-way data binding. It means the data flow is downward, the binding works in a host-to-child way and this binding can never modify a property from the host.

To work with binding, you should specify the attribute for the property you want to bind. In Polymer, data binding works inside the `<template>` tag because all the content of the element goes inside it. Take a look at the following code:

```
<template>
    <h2>{{header}}</h2>
</template>
```

In this example, I've just bound a property with the name `header` using two-way data binding; but wait! I should show you the other part (that is, the JavaScript part) of the binding. When you use a property in brackets, JavaScript checks for the availability of this property. If there's no result, it creates the property for itself without any issues, as follows:

```
<template>
    <h2 is-clickable="{{checker}}">{{header}}</h2>
    <!–- In the JS <view>.isClickable = this.checker; -->
</template>
```

[If you want to bind the CamelCase properties of an element, you should use a dash-case name for the attribute.]

Okay! Now that we know how to define the properties in Polymer, let's define the `header` property in the properties of the Polymer function as follows:

```
<script>
  Polymer({
    is: "view",
    properties: {
      header: String,
      isClickable: Boolean
    }
  });
    </script>
    ...
    <h2 is-clickable="false">{{header}}</h2>
```

Yes, in this example, JavaScript creates a property for the Polymer element and applies it in the view. This means that when you change the value of `header` or `isClickable` in the model of your Polymer function, it will also be changed in the view. This is the best part about data binding; by changing the model, it is updating the view of your application, which is great for the future ☺.

By the way, data binding is also available in the new ECMA 2015; you can get more information about this on the Web.

Let me show you a great example of how the one-way and two-way data bindings work in a real example. Let's create our first element for Gravatar. In this element, we will need to write the Gravatar account's e-mail address and the Gravatar image should appear at the bottom.

What do we need for this task? Here's what:

- An MD5 decoder for JavaScript because Gravatar uses the decoding hash of the e-mail address
- The Gravatar account
- A Polymer paper element called `paper-input`

Here's the code:

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

<dom-module id="img-gravatar">
  <template>
```

```
    <paper-input value="{{email}}" placeholder="Email" id="email"></
paper-input>
    <img src="{{gCode(email)}}" />
  </template>
  <script>
    Polymer({
      is: 'img-gravatar',
      gCode: function(email){
          return "http://gravatar.com/avatar/" + md5(email);
      }
    });
  </script>
</dom-module>

<img-gravatar email="akhxtern@gmail.com"></img-gravatar>
```

Let me explain the code. We just created a Polymer element with the name `img-gravatar`, which has only one parameter to bind and to get the element's attributes. We also have a `gCode` function, which decodes the e-mail address with an MD5 cryptographic hash function added at the top of the element.

Here is the link to md5.js: `https://cdnjs.cloudflare.com/ajax/libs/crypto-js/3.1.2/components/md5-min.js`.

The `gCode` function returns the decoded e-mail address that we bound into the `src` of the `img` tag in the template. I also added `paper-input` before the `img` tag. This changes the `email` parameter here so that you can type your Gravatar e-mail address and see what happens. This is how binding works. Let me show you another example with some lists. Imagine you have a list of videos and want to show them. For this, we have to create an array with a lot of objects and use the binding of Polymer.

Let's execute the following code:

```
<dom-module id="video-list">
  <template>
    <div> Video list: </div>
    <template is="dom-repeat" items="{{videos}}">
        <div># <span>{{index}}</span></div>
        <div>Video title: <span>{{item.name}}</span></div>
        <div><video src="{{item.src}}"></video><div>
    </template>
  </template>
  <script>
    Polymer({
      is: 'video-list',
      ready: function() {
        this.videos = [
            {name: 'Polycasts #1', src: 'assets/videos/vid1.mp4'},
            {name: 'Polycasts #2', src: 'assets/videos/vid2.mp4'}
        ];
      }
    });
  </script>
</dom-module>
```

In this example, we have an object called `videos`, which has two videos in its list. In the view, we created a template for repeating a list. The `items` attribute gives the name of the object we are going to bind from, it refers to `videos` here.

The result of this example is a view with a list of videos. JavaScript binds the model property to the view and renders it. It is like appending an element to a div with the JS `for` loop.

Now, let's talk about behaviors in Polymer.

Behaviors in Polymer

With behaviors, you can extend the prototypes of some elements and use them for other elements. You can create mixins for your element, which look like a module. You can define properties, attributes, observers, and listeners using behavior. To add a behavior to the element, you should add a `behaviors` array in the Polymer prototype, as follows:

```
Polymer({
  is: 'custom-element',
  behaviors: [SomeBehavior]
});
```

However, wait! We don't have any behavior with the name `SomeBehavior`. So, let's create one. To define a behavior, just create a JavaScript object and give it the properties of an element. You can create another file for behaviors—for our example, `some-behavior.html`—as follows:

```
some-behavior.html
<script>
    SomeBehavior = {
      properties: {
        isSelected: {
          type: Boolean,
          value: false,
          notify: true,
          observer: '_changedSelect'
        }
      },
      listeners: {
        click: '_toggleSelect'
      },
  created: function() {
        console.log('Selected: ', this);
      },
      _toggleSelect: function() {
        this.isSelected = !this.isSelected;
      },
      _changedSelect: function(value) {
        this.toggleClass('selected', value);
      }
    };
</script>
```

Your behavior will work for your element. By the way, you can add multiple behaviors for your element, similarly to this:

```
  Polymer({
    is: 'custom-element',
    behaviors: [SomeBehavior1, SomeBehavior2, SomeBehavior3, ...]
  });
```

To avoid collisions in the future, you should think for behavior namespaces when creating a behavior by adding the namespace to the `window` global object, as follows:

```
  window.behaviors = window.behaviors || {};
  behaviors.SomeBehavior = { /* your element behavior here */ }
```

So, you can use a behavior such as `behaviors.SomeBehavior;`.

You can also extend from other behaviors and use the new behavior for your element. For example, it will look like this:

```
  <link rel="import" href="some-behavior.html">
  <script>
    NewSomeBehaviorImpl = {
    /* your new behavior here */
    }
    NewSomeBehavior = [ SomeBehavior, NewSomeBehaviorImpl ]
  </script>
```

`NewSomeBehavior` is your new behavior, use it with pleasure!

Events

You can add default events or create yours in Polymer. Default events are the same, nothing different to JS events. In this topic, we will take a look at how to set up an event handler for our Polymer elements. You can add any event you want on any element using the `this.$` object and the syntax `elementID.eventName`. You should use the `listeners` object to add event handlers to the elements, as follows:

```
<dom-module id="my-tag">
  <template>
    <button>Default Click Event</button>
    <div id="thank">Thanks a lot!</div>
  </template>
  <script>
    Polymer({
      is: 'my-tag',
      listeners: {
        'tap': 'thankYouMessage',
        'thank.tap': 'specialThankMessage'
      },
      thankYouMessage: function(e) {
        alert("Default click event!");
      },
      specialThankMessage: function(e) {
        alert("Thank you for clicking me, my dear!");
      }
    });
  </script>
</dom-module>
```

As you can see, I have a `listeners` object with the events I need to handle in the application. The first event is `tap`, which means that all the elements in the `dom-module` will handle the event when I tap on it. The second one is `thank.tap`, which handles only the element with the `thank` identifier. That's it! You can handle any event you want (mouseover, mouseout, keydown, keyup, and more) and for any element you want.

By the way, we also have other ways to handle the events without using the `listeners` object. You can specify the event handler function for the element itself, similarly to this:

```
<dom-module id="my-tag">
  <template>
    <button on-mouseover="handleMouse">Click me now!</button>
    <p>Hover count: <span>{{mouseTrack}}</span></p>
  </template>
  <script>
    Polymer({
      is: 'my-tag',
      properties: {
          mouseTrack: {
              type: Number,
              value: 0
          }
      },
      handleMouse: function() {
        this.mouseTrack += 1;
      }
    });
  </script>
</dom-module>
```

Here, we used the default `mouseover` event in the attribute of the element. You just need to add the event name after the `on-` keyword (for example, `on-click`, `on-tap`, and so on).

Now, let's take a look at how to create our first custom event in Polymer. To fire a custom event, you should use a `fire` method and, in the second parameter of this method, you can pass any data you want; take a look at this example:

```
<dom-module id="my-tag">
  <template>
    <button on-click="happyEvent">Congratulate me now!</button>
  </template>
  <script>
    Polymer({
      is: 'my-tag',
      happyEvent: function(e, detail) {
        this.fire('congrats', {age: 18});
      }
    });
  </script>
</dom-module>
<my-tag></my-tag>

<script>
    document.querySelector('my-tag').addEventListener('congrats',
function (e) {
        alert('Happy' + e.detail.age + ' years, buddy!');
    });
</script>
```

In this example, I created my custom `congrats` event, which congratulates you by showing you an alert and printing your age. As you can note, in the body of the `addEventListener` method at the bottom, I used the `e.detail` property, which is the data object I passed at the top of the custom event firing. It's kind of awesome, right?

Styling

Polymer elements are Shadow DOM. Shadow DOM has its own rules for styling its content, which we will discover in this topic. Styling? Yeah! It is the same `<style>` tag but inside the `<template>` tag. Let me show you some code from styling Shadow DOM, as follows:

```
<template>
<style>
      :host {
        display: flex;
        width: 100vw;
        background: grey;
      }
      #header {
        color: red;
      }
  .container > ::content .p {
        color: green;
      }
    </style>

    <div id="header">Some header!</div>
    <div class="container"><content></content></div>
</template>
```

What's going on here? What are the `:host` and `::content` pseudo classes? Are they something new? Take a look:

- `:host`: This is the wrapper element that we will use (the Polymer element)
- `::content`: This is the content of Shadow DOM (the `<content>` tag)

As we know, the `<content>` tag doesn't appear in the DOM tree and it is important to have some element on the left-hand side before the `::content` pseudo class, because it doesn't know which Shadow DOM content to apply the styles to.

In Polymer, you can create your custom CSS properties, which you can use multiple times in code. It is the future of W3C CSS standards. This way, you define a CSS variable (as in SCSS or LESS); take a look:

```
<dom-module id="my-tag">
  <template>
    <style>
      :host {
        padding: 10px;
        background: teal;
      }
      .title {
        color: var(--my-tag-nav-color);
      }
    </style>
    <div class="nav">{{nav}}</span>
  </template>
  <script>
    Polymer({
      is: 'my-tag',
      properties: {
        nav: String
      },
    });
  </script>
</dom-module>
```

Now, the user can give the variable value outside of the shadow tree; here's an example:

```
<my-tag></my-tag>
<style>
  my-tag {
  color: white;
}
</style>
```

CSS gets the parameter from the outside and applies it to its tree. You can also assign a default value for the variable inside your template, as follows:

```
<style>
      :host {
        --my-tag-nav-color: black;
      }
</style>
```

Mixins in Polymer

If you are familiar with a CSS preprocessor, you must know about mixins. Mixins allow a user to define the patterns of property value pairs, which you can use anywhere in the document. Polymer also has a mixins feature in it. To apply a mixin, you should use `@apply` in your document, as follows:

```
@apply( --mixin-name );
```

To define a mixin, you should write the name of it and the properties it should have, as shown in the following code:

```
selector {
  --mixin-name: {
    /* properties */
  };
}
```

In the example, it will look similar to this:

```
<template>
    <style>
      /* Apply custom theme to toolbars */
      :host {
        --my-tag-theme: {
          background: teal;
        };
        --my-tag-title-theme: {
          color: red;
        };
      }
    </style>

    <my-tag title="Red title!"></my-tag>
  </template>
```

Summary

Here we are! We've covered all the features of Polymer in this chapter. You know now what Polymer is and have learned about its great features.

Polymer is growing so fast that I think, while you were reading this chapter, the guys from Google were rocking with Polymer and making a lot of new stuff, but the idea of it is the same. It's an awesome tool for all web developers in the world to create faster and more responsive apps.

You know now how Polymer works; about its life cycles, events, and binding; and how to create a custom polymer component with Polymer and use it in other applications.

In the next chapter, I will show you the Polymer elements collection created by the Polymer team at Google. Let's move on!

4
Polymer Elements

Polymer has a collection of useful elements created by the Polymer team from Google. In Polymer version 0.5, there were two sets of elements: Core and Paper elements, but now in version 1.0, there are a lot.

In this chapter, we will look at the following types of elements:

- App elements
- Iron elements
- Paper elements
- Google web components
- Gold elements
- Neon elements
- Platinum elements
- Molecules

To use our own markup elements, we usually perform the following steps:

1. Download the element package through Bower.
2. Import the appropriate `.html` file.
3. Use the imported elements anywhere in the document.

With this, let's go through the different Polymer elements and take a look at their power.

App elements

These are the elements we need to start our Polymer application, so the Polymer team has created this list of elements for us to make the application layouts faster (`https://elements.polymer-project.org/browse?package=app-elements`).

app-layout

The `<app-drawer-layout>` is the main wrapper of the app. There is a drawer panel to the left and a content area to the right.

Here's an example:

```
<app-drawer-layout>
  <app-drawer>
    drawer content
  </app-drawer>
  <div>
    main content
  </div>
</app-drawer-layout>
```

It has the following properties:

- `drawer`: This is the `<app-drawer>` element
- `forceNarrow`: If this is true, it ignores `responsiveWidth` and forces a narrow layout
- `responsiveWidth`: This is the minimum value of width close to the drawer

app-route

The `<app-route>` is a component that enables declarative routing for the web app.

Here's an example:

```
<app-location route="{{route}}"></app-location>
<app-route
    route="{{route}}"
    pattern="/:page"
    data="{{data}}"
    tail="{{tail}}">
</app-route>
```

It has the following properties:

- `data`: These are the values that are extracted from the route as described in `pattern`
- `pattern`: This is the pattern of slash-separated segments to match the path against
- `route`: This is the URL component managed by the element
- `tail`: This is the part of the path *not* consumed in `pattern`

Iron elements

These are the basic built-in blocks to create an application with visual and nonvisual elements. For example, `iron-icons` is a set of material icons and `iron-ajax` is used to request AJAX calls and parse the response. I will present some of them here (a collection that will be very useful for you), but you can see a list of the elements on the Polymer website at `https://elements.polymer-project.org/browse?package=iron-elements`.

iron-a11y-keys

The `iron-a11y-keys` element provides an interface to process keyboard commands in your application. It uses an expressive syntax to filter key presses. For example, if you want to call some function in some key event, you can use this element to handle it.

Here's an example:

```
<iron-a11y-keys target="[[target]]" keys="space" on-keys-
pressed="onSpace"></iron-a11y-keys>

...
onSpace: function() {
  // Do something
}
...
```

You can also use this element to combine some hotkeys—for example, *Ctrl* + *F*—similarly to this:

```
<iron-a11y-keys target="[[target]]" keys="ctrl+f" on-keys-
pressed="decrement"></iron-a11y-keys>
```

iron-ajax

You can easily make simple AJAX calls and get the response of them with just one custom element.

Here's an example:

```
<iron-ajax>
    url="http://example.com/api.php"
    params='{"action":"signin", "email":"akhxtern@gmail.com",
"password": "123456"}'
    handle-as="json"
    on-response="loginResponse"
</iron-ajax>
```

This has the following properties:

- `url`: This is the URL to request
- `params`: These are the query parameters of the request
- `handle-as`: This is the expected datatype (JSON, text, XML, `arrayBuffer`, blob, or document)
- `on-response`: This is the response handler function

Then, in JavaScript, you just need to handle its response. The `on-response` attribute is responsible for this, while the `loginResponse` function is the response handler for the call. The `handle-as` attribute is for the specification of the response data and you can use XML, text, document, and so on. This part of JavaScript will look similar to this:

```
...
loginResponse: function(data) {
  if(data.result) {
     // Do some login functionality
  } else {
     // Something wrong in the username or password
  }
}
...
```

iron-collapse

With this element, you can create collapsible blocks of content. A collection of this element is called an accordion.

Here's an example:

```
<button on-tap="collapseToggle">Show / Hide the bar</button>

<iron-collapse id="bar">
  Hello ceeceeque!
</iron-collapse>

...

collapseToggle: function() {
  this.$.bar.toggle();
}
```

As you can see in the code, we have a button that toggles the collapsible element with the `bar` ID and, in the JavaScript code, we have the `on-click` function to do the toggling.

iron-image

The `iron-image` is an element to show an image; you can change the sizing options, its resolution, and more.

Here's an example:

```
<iron-image src="http://lorempixel.com/600/400" style="width:200px;
height:500px; background-color: blue;" preload sizing="cover"></iron-
image>
```

This has the following properties:

- `alt`: This is an alternative text for the image
- `fade`: When the image loads, this fades into its place (when preload is set to `true`)
- `width`: This is the width of the image
- `height`: This is the height of the image
- `src`: This is the URL of the image
- `position`: These are the sizing options of the image (cover or contain)

This element has his own attributes for configuration; you can set some style for the image and change its size, which works similar to the CSS `object-fit` property for the image.

iron-dropdown

This element is responsible for creating a view similar to the `select` element in HTML, but the difference is that you can hide some content in the dropdown area; for example you can put buttons, images, or any other HTML tags you want in the dropdown container. It is a very useful element when you want to show some hidden content. There are a lot of attributes you can use for its configuration, but I will show you some of them now.

Here's an example:

```
<iron-dropdown horizontal-align="right" vertical-align="top">
  <div class="dropdown-content">I'm here!</div>
</iron-dropdown>
```

It has the following properties:

- `alwaysOnTop`: The *z* index of this drop-down menu is always at the top
- `horizontalAlign`: This is the horizontal direction of the animation (left or right)
- `verticalAlign`: This is the vertical direction of the animation (top or bottom)
- `noAnimations`: This is used to open and close the drop-down menu without animations
- `withBackdrop`: If this is true, it displays the backdrop behind the drop-down menu

iron-flex-layout

`iron-flex-layout` element provides you the opportunity to use CSS flex layouts. This means that you can use mixins and default classes to use flexbox. There are two ways to do this, which are as follows:

- **With layout classes**: This is a simple set of built-in stylesheet classes so that you can set the class you want to the elements in the markup
- **Custom CSS mixins**: This is a custom collection of mixins that can be used in your CSS rules using the `@apply` function

For example, if you want to create some horizontal flex tabs and wrap the container, you can use the `horizontal` and `wrap` classes, as follows:

```
<div class="layout horizontal wrap">
```

So the `<div>` container will do what you want.

And we have the mixins version, right? So, this is simple; as in this example, just apply the mixins in your CSS, this way:

```
<div class="container"></div>
<style is="custom-style">
    .container {
      @apply(--layout-horizontal);
      @apply(--layout-wrap);
    }
</style>
```

You can refer to the full documentation of the Polymer flexbox at `https://elements.polymer-project.org/guides/flex-layout`.

iron-form

The `iron-form` element is similar to the HTML `form` tag. It can validate the input data and submit it to an action URL with the GET or POST methods and use the `iron-ajax` element to request. For input elements, you should take a look at paper elements because the most visual elements of Polymer are in the paper collection.

Here's an example:

```
<form is="iron-form" id="search_form" method="post" action="/api/
users">
  <paper-input name="username" label="username"></paper-input>
  <input type="email" name="email">
  <paper-button onclick="searchUser()" raised>Search</paper-button>
</form>

function searchUser() {
  document.getElementById('search_form').submit();
}
```

There are some methods which you can call for certain reasons. To submit data, you should use `submit()`; to validate the form data, you should use the `validate()` method; and to serialize the data, use `serialize()`.

Of course, you can use some built-in events for the `form` element, such as `iron-form-error`, `iron-form-invalid`, `iron-form-presubmit`, `iron-form-reset`, `iron-form-response`, and `iron-form-submit`.

iron-icon

This element is used to display a Material Design icon from the Polymer icon set. There are a few groups of icons, such as default, AV (audio/video), communication, device, editor, hardware, image, maps, notification, social, and places icons.

To add a specific group of icons, you just need to import the specific icon HTML code inside `bower-components/iron-icons/`.

Here's an example:

```
<iron-icon icon="icons:bat_icon"></iron-icon>
```

As you can note in the example, `<iron-icon>` has an `icon` attribute, which is the name of the icon from the Polymer icon set.

The default size of the icon is 24px, but `<iron-icon>` has its own CSS properties for resolutions and colors. Take a look at the following code:

```
<style is="custom-style">
  .bat_icon {
    --iron-icon-height: 50px;
    --iron-icon-width: 50px;
    --iron-icon-fill-color: black;
    --iron-icon-stroke-color: yellow;
  }
</style>
```

You can also use a set of icons using `<iron-icon>`; you just need to import the icon set and use `<iron-icon>`, this way:

```
<iron-icon icon="iconset_name:icon_name"></iron-icon>
```

iron-swipeable-container

The `iron-swipeable-container` is a container that users can swipe away (with its children); for example, you can use this element if you want to alert or inform the user about something. There are default transition styles, curved and horizontal, but you can customize the duration and properties.

You can write native HTML elements inside this container or use Polymer paper elements; the decision is yours.

Here's an example:

```
<iron-swipeable-container>
  <paper-card heading="Me too!"></paper-card>
  <div>Please swipe me! :)</div>
  <div class="disable-swipe">You cannot swipe me!</div>
</iron-swipeable-container>
```

If you want to disable the swiping of some children, just add the `disable-swipe` class to the element.

Paper elements

Yes! My favorite part of elements is the paper. They are the visual collection of elements in Polymer created according to Material Design concepts, so you can add a lot of cards, inputs, checkboxes, and buttons and have a lot of fun with ripple effects. In this section, we will cover these elements in detail (`https://elements.polymer-project.org/browse?package=paper-elements`).

paper-badge

The `<paper-badge>` element is used to represent a status or notification in the upper-right corner of an element. I know you have noticed the notification status in the Facebook website or app, the red circle with white text at its head; this element is to create a view like that. You can use text or an icon in the badge. Here's an example of how to add it on the markup:

```
<div>
  <paper-button id="msgs">Messages</paper-button>
  <paper-badge icon="communication:email" for="msgs" label="favorite
icon"></paper-badge>
</div>

<div>
  <paper-icon-button id="notification-box" icon="account-box"
alt="notification-box"></paper-icon-button>
  <paper-badge icon="social:mood" for="notification-box" label="mood
icon"></paper-badge>
</div>
```

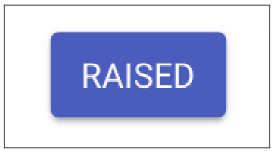
In this example, we have two types of buttons: one with text and one with icons. Each one has its own badge of status. This is simple and beautiful! You can even change the color of badges or margins with CSS rules, as follows:

```
paper-badge {
   --paper-badge-margin-left: 10px;
   --paper-badge-margin-bottom: 0px;
   --paper-badge-background: var(--paper-light-green-100);
}
```

paper-button

This is the simplest button in the world. When the user clicks on this button, it rises with a ripple effect and casts a shadow. You can configure the button with attributes, and the main attribute is that it can be flat or raised. If the button has no attribute, it is flat; add the `raised` attribute to change the type and add the `noink` attribute to disable the ripple effect.

Here's the view:



Here's an example:

```
<paper-button>Tapak</paper-button>
<paper-button raised>Ceeceeque</paper-button>
<paper-button noink>Simple button</paper-button>
```

This has the following properties:

- `active`: If true, the button is currently active
- `disabled`: If true, the button is in disabled state
- `elevation`: This is the *z* depth of the element—that is, the shadow level
- `noink`: If this is true, there's no ripple effect on pressing the button
- `raised`: If this is true, then the button has a shadow

If you add an `<iron-icon>` element inside `<paper-button>`, you might get a button with an icon and with some text if you want. Consider the following example:

```
<paper-button>
  <iron-icon icon="star"></iron-icon>
  Like
</paper-button>
```

Now, we have a button with an icon and text.

paper-card

This element is a Material Design container with a shadow. We met this element in the `<icon-swipeable-container>` element's example.

Here's an example:

```
<paper-card heading="Material Card">
  <div class="card-content">Mr. Echo</div>
  <div class="card-actions">
    <paper-button noink raised>Hello</paper-button>
  </div>
</paper-card>
```

This simple example is for a card with a header and content, but you can add a header with an image just by adding the `image` attribute in the `<paper-card>` tag and by providing the link to the image in the value!

paper-checkbox

This element is a checkbox; it's the same element as the checkbox in HTML, but it has Material Design animations and styles. It can be checked or unchecked and, no, it is not an ON/OFF switcher; we have another element for this.

Here's an example:

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

You can add the `checked` attribute to check and remove it to uncheck the checkbox.

paper-drawer-panel

This element is a navigation drawer for your application. The `<paper-drawer-panel>` element contains two side-by-side panels: `drawer` and `main`. When the browser window is of a small size, the `drawer` panel is hidden.

Here's an example:

```
<paper-drawer-panel>
  <div drawer> drawer part </div>
  <div main> content part </div>
</paper-drawer-panel>
```

You should add the `drawer` and `main` attributes to specify the panels. The `drawer` part is usually useful for the menu panel.

paper-dropdown-menu

The `<paper-dropdown-menu>` element is the same as the `<select>` element in native HTML, but here you should use one more element to make the view similar to the native one.

Here's an example:

```
<paper-dropdown-menu label="Your favorite musician?">
  <paper-menu class="dropdown-content">
    <paper-item>Notorious B.I.G.</paper-item>
    <paper-item>Woodkid</paper-item>
    <paper-item>Gesaffelstein</paper-item>
    <paper-item>The Beatles</paper-item>
  </paper-menu>
</paper-dropdown-menu>
```

You should add the `<paper-menu>` element inside it and the result will be the beautiful Material Design element.

paper-fab

This is the popular floating action button in Material Design. It contains an icon in the center styled with a shadow and could be of two sizes: regular and small. To use the FAB button, you should import `iron-icon` for the icon inside.

Here's an example:

```
<link href="bower_components/iron-icons/iron-icons.html" rel="import">

<paper-fab icon="edit"></paper-fab>
<paper-fab mini icon="star"></paper-fab>
```

To use the smaller version, you should add the `mini` attribute inside the element tag.

paper-icon-button

The `<paper-icon-button>` element is a simple button with an icon at the center of it. The ripple effect comes from the center of the button. This element uses the default icon set, so, if you want to set an icon, just add the `icon` attribute and add the icon name (of course you need to import `iron-icon`) and the same attributes, as in `<paper-button>`.

Here's an example:

```
<paper-icon-button noink icon="add"></paper-icon-button>
```

The styling of this element is also with built-in custom parameters:

- `--paper-icon-button-disabled-text`: This is the color of the text of the disabled button
- `--paper-icon-button-ink-color`: This is the ripple color

paper-input

This is the text input area of Material Design. With this element, you can create beautiful form fields. You can add a maximum length of character in the value and any validations you want.

Take a look at the following example:

```
<paper-input label="Text" char-counter maxlength="10" ></paper-input>
```

In this example, we used `char-counter`; if it's set to true, the character counter is kept visible and we give the maximum length of characters.

paper-listbox

The `<paper-listbox>` element is a Material Design listbox controller. It is similar to the right-click options bar in your operating system. If you select an item from this list, the item becomes bolder, and the focused items are highlighted when hovered over.

Here's an example:

```
<paper-listbox>
  <paper-item>POP.xyz</paper-item>
  <paper-item>JAZZ</paper-item>
</paper-listbox>
```

If you want to make it a multiselect listbox, you just need to add the `multi` attribute inside it.

paper-material

This is the container to index the Materials layers with shadow levels. To change the material level, just change the `elevation` attribute.

Here's an example:

```
<paper-material elevation="2">
  …I'm higher than you! …
</paper-material>
```

You can also use the `animated` attribute to add an animation on shadow changes.

paper-menu

This is the built-in menu control with Material Design styling.

Here's an example:

```
<paper-menu selected="0">
  <paper-item>Edit</paper-item>
  <paper-item>Delete</paper-item>
  <paper-item>Share</paper-item>
</paper-menu>
```

You can use the `multi` attribute, as in `<paper-listbox>`, to enable multiselection.

paper-progress

The progress bar is also an important element in Material Design. You can show the progress of some action or the buffer level during the streaming of a video. Just change the `value` attribute to change the progress position.

Here's an example:

```
<paper-progress value="10"></paper-progress>
```

You can change the color of the progress bar or the timing parameters from the CSS properties of the element, as follows:

```
paper-progress {
  --paper-progress-active-color: #e91e63;
  --paper-progress-transition-duration: 0.03s;
  --paper-progress-transition-timing-function: ease-in-out;
  --paper-progress-transition-transition-delay: 0s;
}
```

paper-radio-button

This element is similar to the native HTML radio button but with Material Design styling. A user can click on it to check or uncheck it.

Here's an example:

```
<paper-radio-button>I love meat</paper-radio-button>
<paper-radio-button>I love music</paper-radio-button>
```

To use a group of radio buttons in Polymer, you can use the `<paper-radio-group>` element, as follows:

```
<paper-radio-group selected="male">
  <paper-radio-button name="secret">Secret</paper-radio-button>
  <paper-radio-button name="male">Male</paper-radio-button>
  <paper-radio-button name="female">Female</paper-radio-button>
</paper-radio-group>
```

paper-ripple

The `<paper-ripple>` element is used to add a ripple effect animation to an element. For example, if you use this element inside a div, it will respond to your clicks with ripples. Notice that this element adds relative position to its parent, because `<paper-ripple>` is positioned with an absolute method I can go outside of its parent.

Here's an example:

```
<div style="position:relative">
  <paper-ripple class="circle"></paper-ripple>
</div>
```

In this example I have used the `<paper-ripple>` element inside a div tag. It will ripple when you click on it and note that the ripple will be a circle if you add the `circle` class. You can also add the `center` attribute to make the rippling animation start from the center.

paper-slider

This element is similar to the native HTML range input element. It gives the user an opportunity to select a value from the range. You can add minimum and maximum values of the range and, of course, you can provide the value of the slider.

Here's an example:

```
<paper-slider min="0" max="100" value="69"></paper-slider>
```

By adding the `step` attribute, you can split the slider into steps. Take a look at its full documentation at `https://elements.polymer-project.org/elements/paper-slider`.

paper-spinner

This element provides a beautiful Material Design loading icon.

Here's an example:

```
<paper-spinner active></paper-spinner>
```

You can change the colorful spinner to some specific color with CSS properties, such as `--paper-spinner-layer-1-color`, `--paper-spinner-layer-2-color`, `--paper-spinner-layer-3-color`, and `--paper-spinner-layer-4-color`. The `active` attribute shows the spinner; just remove the attribute to hide it.

paper-tabs

This element provides a menu panel with tabs. It comes with awesome animations that will make the UI of your website the best! When the user selects an item, the border at the bottom of the current menu item animates to the item you have selected and the tabs use the ripple effect animation.

Here's an example:

```
<paper-tabs selected="0">
  <paper-tab>Home</paper-tab>
  <paper-tab>About us</paper-tab>
  <paper-tab>Portfolio</paper-tab>
</paper-tabs>
```

You can use this paper element with `<iron-pages>` and create a simple website in a second. Take a look at the following example:

```
  <paper-tabs selected="{{selected}}">
  <paper-tab>Home</paper-tab>
  <paper-tab>About us</paper-tab>
  <paper-tab>Portfolio</paper-tab>
</paper-tabs>

<iron-pages selected="{{selected}}">
  <div>This page is for homepage</div>
  <div>This page is about us</div>
  <div>This page is for my portfolio</div>
</iron-pages>
```

That's it! Your simple one-page application is ready. How does it work? The `selected` variable binds in the `selected` attribute and shows the correct tab/page.

paper-toast

These are notification toasts for your application. With this element, you can show error messages or show notifications (as with Twitter or Facebook).

Here's an example:

```
<paper-toast id="hello" text="Get the new app from here!">
  <a href="http://play.google.com">Play Store</a>
</paper-toast>
<button onclick="document.querySelector('#hello)
  .open()">Login</button>
```

In this example we have a button that opens the toast when you click on it!

paper-toggle-button

This is a button to switch ON/OFF, which a user can toggle by clicking or dragging.

Here's an example:

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

Google web components

This collection of elements is created to make it easier to use Google APIs. For example, until now, you added a Google Map on your website by writing some markup and calling Google Maps API methods with JavaScript to create the map. However, now, using Polymer, you can add the map with one simple tag. That's not all, you can add any Google component you want with simple code.

The following subsections list some useful Google web components that will come in handy for you (`https://elements.polymer-project.org/browse?package=google-web-components`).

google-analytics-query

This element is responsible for querying the Google Analytics Core Reporting API.

Here's an example:

```
<google-analytics-query
  ids="ga:1174"
  metrics="ga:sessions"
  dimensions="ga:country"
  sort="-ga:sessions"
  max-results="5">
</google-analytics-query>
```

Here, `ids` is your user ID for requesting your account information.

google-client-loader

With this element, you can load a specific Google API with JavaScript.

Here's an example:

```
<google-client-loader id="shortener"
  name="urlshortener"
```

```
    version="v1"></google-client-loader>

  <script>
    var shortener = document.getElementById('shortener');
    shortener.addEventListener('google-api-load', function(event) {
      var request = shortener.api.url.get({
         shortUrl: 'http://goo.gl/fbsS'
      });
      request.execute(function(resp) {
        console.log(resp);
      });
    });
  </script>
```

google-chart

The `<google-chart>` element easily visualizes data with charts. Any known chart is included in this element: line, circular, map, and more. Here's a list of chart types that you can use in code: area, bar, bubble, candlestick, column, combo, geo, histogram, line, pie, scatter, stepped area, and treemap.

Here's an example:

```
  <google-chart
    type='pie'
    options='{"title": "Distribution of days in 2001Q1"}'
    cols='[{"label":"Month", "type":"string"}, {"label":"Days",
      "type":"number"}]'
    rows='[["May", 16],["Nov", 9],["Aug", 29]]'>
  </google-chart>
```

google-hangout-button

This is a simple Hangout button for your application. Now, you can start calling anytime and anywhere by adding this element.

Here's an example:

```
  <google-hangout-button></google-hangout-button>
```

google-map

This is my favorite element in this collection. I remember having to add tons of JavaScript code and markup for one simple Google Maps code, but now, this can be done using just a tag. Yeah, it's magic!

Here's an example:

```
<style>
  google-map {
    height: 400px;
  }
</style>
<google-map latitude="40.1553425" longitude="44.5166002" zoom="18"></
google-map>
```

All you need to do, is add the center's latitude and longitude and the zoom of the map. You can also add markers on the map! Again, you can do this with a simple element.

Here's an example:

```
<google-map latitude="40.1553425" longitude="44.5166002" fit-to-
  markers>
  <google-map-marker latitude="37.779" longitude="-122.3892"
      draggable="true" title="Go Giants!"></google-map-marker>
  <google-map-marker latitude="37.777" longitude="-
    122.38911"></google-map-marker>
</google-map>
```

The `draggable` attribute in the `<google-map-marker>` element enables the drag and drop of the marker. Now, let's add directions with Google Maps, as follows:

```
<google-map map="{{map}}"></google-map>
<google-map-directions map="{{map}}"
    start-address="Yerevan" end-address="Gyumri">
</google-map-directions>
```

google-signin

This element is used to authenticate a Google user account, which allows you to interact with Google APIs such as Google+ or Drive.

Here's an example:

```
<google-signin client-id="..." scopes="https://www.googleapis.com/
auth/drive"></google-signin>
```

Here, `client-id` is your unique application ID, which you can get from the Google Developers Console (`https://console.developers.google.com`).

google-streetview-pano

This is an element to show a specific street view from Google Maps Street View Panorama. All you need to do is paste the pano ID and configure some default settings.

Here's an example:

```
<google-streetview-pano
  pano-id="PANO ID"
  heading="230"
  pitch="-3"
  zoom="0.5"
  disable-default-ui>
</google-streetview-pano>
```

Go grab a panorama and look at the URL in the address bar. An example would be `google.com/maps/views/view/123012930218409142190/photo/PANO_ID_IS_HERE`.

google-youtube

You can use this element to show YouTube videos on your page without any iframes; just add this element.

Here's an example:

```
<google-youtube
  video-id="VIDEO_ID"
  height="640px"
  width="360px"
  rel="0"
  start="2"
  autoplay="0">
</google-youtube>
```

Include a `VIDEO_ID` of your YouTube video and feel the magic!

Gold elements

This collection of elements is built for e-commerce use cases, such as checkout flows. Obviously, it is a collection of elements connected with money (`https://elements.polymer-project.org/browse?package=gold-elements`).

gold-cc-cvc-input

This is a Material Design styled input area to enter a credit card's verification code.

Here's an example:

```
<gold-cc-cvc-input card-type="amex"></gold-cc-cvc-input>
```

gold-cc-input

This is a Material Design styled input area to enter a credit card number. As the user types, the number is formatted by adding a space after every four digits.

Here's an example:

```
<gold-cc-input label="Master Card"></gold-cc-input>
```

gold-email-input

This element is a simple text field, especially for e-mail addresses, styled in Material Design.

Here's an example:

```
<gold-email-input label="SUBSCRIBE"></gold-email-input>
```

For validation, the element has a `validate()` method, which returns `true` if it is valid and otherwise if it is not valid.

gold-phone-input

This element is a simple text field, especially for phone numbers, styled in Material Design.

Here's an example:

```
<gold-phone-input
    country-code="374"
    phone-number-pattern=XX-XX-XX-XX">
</gold-phone-input>
```

For validation, the element has a `validate()` method, which returns `true` if it is valid and false otherwise.

Neon elements

This collection has only one element so far. It fills the dark space of Material Design on the Web: meaningful transitions. With this element, you can easily make cool and fast transitions for the web and mobile and implement pluggable animated transitions using web animations (`https://elements.polymer-project.org/browse?package=neon-elements`).

You should implement the `Polymer.NeonAnimatableBehavior` behavior or `Polymer.NeonAnimationRunnerBehavior` for element animation. Let me show you one example of neon animation and how it works via the following code:

```
Polymer({
  is: 'popup-box',
  behaviors: [
    Polymer.NeonAnimationRunnerBehavior
  ],
  properties: {
    checker: {
      type: Boolean
    },
    animationConfig: {
      value: function() {
        return {
          'in': {
            name: 'scale-up-animation',
            node: this
          },
          'out': {
            name: 'scale-down-animation',
            node: this
          }
        }
      }
    }
  },
  listeners: {
    'neon-animation-finish': '_onNeonAnimationFinish'
  },
  show: function() {
    this.checker = true;
    this.style.display = 'inline-block';
    this.playAnimation('in');
  },
  hide: function() {
    this.checker = false;
    this.playAnimation('out');
  },
  _onNeonAnimationFinish: function() {
    if (!this.checker) {
      this.style.display = 'none';
    }
  }
});
```

In this example, we have a neon animation of a box that scales up and then down. We have the `animationConfig` property for our animations, which has two transitions: `in` and `out`. One of these properties is scaling the box, the other one is hiding. To play the animation, we have the `playAnimation()` method, which calls the specific animation you are calling. There are a few animations with `neon-animation` and you can take a look at them at `https://elements.polymer-project.org/elements/neon-animation`.

Platinum elements

This is a set of Polymer elements to turn your web app into a mobile app. You can use Bluetooth in Polymer, push messages, and handle notifications (`https://elements.polymer-project.org/browse?package=platinum-elements`).

platinum-bluetooth

This element allows you to search for nearby Bluetooth devices using the Web Bluetooth API. But first, you need to enable this feature in Chrome. Just go to `chrome://flags/#enable-web-bluetooth` and enable the flag item.

Here's an example:

```
<platinum-bluetooth-device
    services-filter='["battery_service"]'>
</platinum-bluetooth-device>

…

document.querySelector('platinum-bluetooth-device').request()
.then(function(device) { console.log(device.name); })
.catch(function(error) { console.error(error); });
```

platinum-push-messaging

The `<platinum-push-messaging>` element sets up push messaging.

Here's an example:

```
<platinum-push-messaging
  title="Application downloaded"
  message="The application was successfully downloaded"
  icon-url="pacman.apk"
  click-url="download.html">
</platinum-push-messaging>
```

Molecules

Molecules are elements that wrap your app with other JavaScript libraries. They are useful when you want to connect a bunch of plugins to your Polymer app (`https://elements.polymer-project.org/browse?package=molecules`).

marked-element

The `<marked-element>` element gets the markdown value and renders it to a child element with the `markdown-html` class. If you haven't created the `markdown-html` element, markdown will still be rendered, but inside Shadow DOM.

Here's an example:

```
<marked-element markdown="YAYAYA! `Markdown` is here!">
  <div class="markdown-html"></div>
</marked-element>
```

You can also use markdown in another way. Just create a `<script>` tag inside `<marked-element>` and give it `type="text/markdown"`. It will render the text inside the `markdown-html` div tag, as follows:

```
<marked-element>
  <div class="markdown-html"></div>
  <script type="text/markdown">
    <em>Please welcome the markdown!</em>
  </script>
</marked-element>
```

That's it! These are the Polymer elements. We just got all the important Polymer elements we should use in the future. With these collections, you can create a lot of complex websites in a hurry. The paper elements create the Material face of your app, and the iron elements are the soul of your project.

You can follow the updates of Polymer elements on the official website of Polymer, so stay with me in the next chapter, in which we will create our first Polymer application.

Summary

In this chapter, we considered a lot of cool, reusable elements created by Google's Polymer team. This means that we have all the stuff to turn on development mode in this book.

In the next chapter, we will create a simple app using these components.

5
First Application with Polymer

Hey again!

In the previous chapters, we learned about Material Design, Polymer, and web components. Now it is time to create our first app using these concepts and technologies.

We will create our first application using Material Design and Polymer knowledge, and create our own custom reusable elements. Moreover, we will use the built-in elements of Polymer and follow the concepts of Material Design to make a fast, beautiful, and multidevice app.

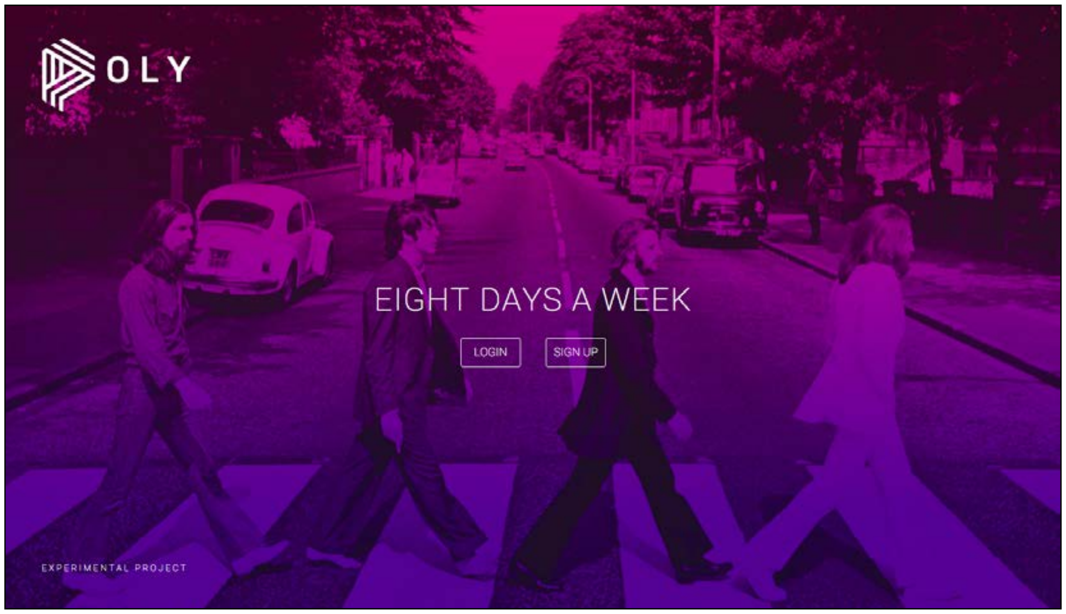
A lot of code, logic, and Polymer will be used here. So, let's go ahead, the most interesting part begins now.

Here are the topics that we will cover in this chapter:

- Setting up Gulp
- Developing the login page of the app
- Connecting the backend part with the app
- Home page
- Creating our custom elements for songs and player
- Showing all the artists

Meet P O L Y

In this chapter, we will develop an app named *P O L Y*—a music platform—where you can sign up, listen to some cool music, add it to your personal playlist, and see what other users are listening to.



How the idea came

While writing, a nice little idea struck me. How about we create an experimental music app for desktop and a mobile web with Polymer? I think it would be a great experience for our readers to work on a real project, and develop the app from the core.

What is the project structure?

The skills required for this project are as follows:

- Polymer
- SASS
- JavaScript

From JavaScript runners, we will use Gulp, because it's more convenient to configure and because I just love it.

Here's the project structure that I have thought of:

- `assets`
- `dist` (this folder is for minified `.css` and `.js`)
- `img`
- `js`
- `scss`
- `elements` (our own elements for the app)
- `songs` (the app songs)
- `bower_components` (for Polymer components)
- `index.html` (the login page)
- `home.html` (the app)
- `elements.html` (for the import of elements)
- `.htaccess` (for some Apache configurations)

What about the backend?

The backend is already written. I've used PHP for it, and it communicates with JSON. There's nothing else you need to know about the backend. I will share the code in GitHub with the name *P O L Y* when we finish the project.

Why do we need SASS?

As for any project in the world, we need some custom styles for the app. There are some issues with the inline custom stylesheet imports in Polymer, so I thought that it would be better to use SASS, and collect all styles in one compact file.

As this is an experimental project, we will use songs in the `.mp3` format.

Setups

First of all, we need to set up the Gulp, so follow the steps listed next:

1. Install `npm`:
   ```
   $ npm init
   ```
2. Install Bower:
   ```
   $ npm install bower -g
   ```

3. Install Gulp:

   ```
   $ npm install gulp
   ```

   This last command will install Gulp locally to the project.

   Now, we need to install the Gulp plugins to achieve the tasks locally.

4. Install all necessary Gulp plugins:

   ```
   $ npm install gulp-ruby-sass gulp-cssnano gulp-jshint gulp-concat
   gulp-uglify gulp-notify gulp-cache gulp-rename del --save-dev
   ```

5. Create `gulpfile.js`, and load the plugins:

   ```
   var gulp = require('gulp'),
       sass = require('gulp-ruby-sass'),
       cssnano = require('gulp-cssnano'),
       jshint = require('gulp-jshint'),
       uglify = require('gulp-uglify'),
       concat = require('gulp-concat'),
       notify = require('gulp-notify'),
       rename = require('gulp-rename'),
       cache = require('gulp-cache'),
       del = require('del');
   ```

6. Create Gulp tasks for SASS and JS:

   ```
   gulp.task('styles', function() {
     return sass('assets/scss/main.scss', { style: 'expanded' })
       .pipe(gulp.dest('assets/dist/css'))
       .pipe(rename({suffix: '.min'}))
       .pipe(cssnano())
       .pipe(gulp.dest('assets/dist/css'))
       .pipe(notify({ message:'SCSS files compiled!' }));
   });

   gulp.task('scripts', function() {
     return gulp.src('assets/js/**/*.js')
       .pipe(jshint())
       .pipe(jshint.reporter('default'))
       .pipe(concat('main.js')) // merging all js files into one
   main.js file
       .pipe(gulp.dest('assets/dist/js'))
       .pipe(rename({ suffix: '.min' })) //creating minified version
       .pipe(uglify())
       .pipe(gulp.dest('assets/dist/js'))
       .pipe(notify({ message: 'JS files minified!' }));
   });
   ```

7. Now Gulp will compile the `.scss` files into `.css`, will concat, uglify, and minify the `.js` files, and will move all of them into the `assets/dist/` folder. Then we just need to add the `watch` and `clear` tasks as follows:

   ```
   gulp.task('clean', function() {
     return del(['assets/dist/css', 'assets/dist/js']);
   });

   gulp.task('watch', function() {
     gulp.watch('assets/scss/**/*.scss', ['styles']);
     gulp.watch('assets/js/**/*.js', ['scripts']);
   });

   gulp.task('default', ['clean'], function() {
     gulp.start('styles', 'scripts');
   });
   ```

Starting with the app development

We will start first with the development of the login page (`index.html`), as it is the first page that the user will see.

The login page

The login page has multiple elements that we should implement in `index.html`:

- Logo at the top
- Tagline in the middle of the page that says "Eight Days a Week"
- **Login** and **Sign up** buttons

As seen in the design shown earlier, there are some grid alignments and structure, so I think we should use `<iron-flex-layout>` to build a grid, then add components.

Just run the following command in the command line:

```
$ bower install --save PolymerElements/iron-flex-layout
```

Once we have installed the element, let's start with the main HTML elements that we should create at the beginning of every project:

```
<html>
<head>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>P O L Y</title>
  <link rel="stylesheet" is="custom-style" href="assets/dist/css/main.
min.css">

  <!-- Importing the elements file -->
  <link rel="import" href="elements.html">
</head>
<body unresolved>
  …
</body>
</html>
```

I have added our minified `.css`, and imported `elements.html`, which is a file for all other imported elements. Yeah, I love clear code!

The unresolved attribute in the `<body>` tag ensures that no Polymer custom elements are displayed before Polymer is ready.

Let's import `<iron-flex-layout>`.

The elements.html file

The following line of code imports `<iron-flex-layout>`:

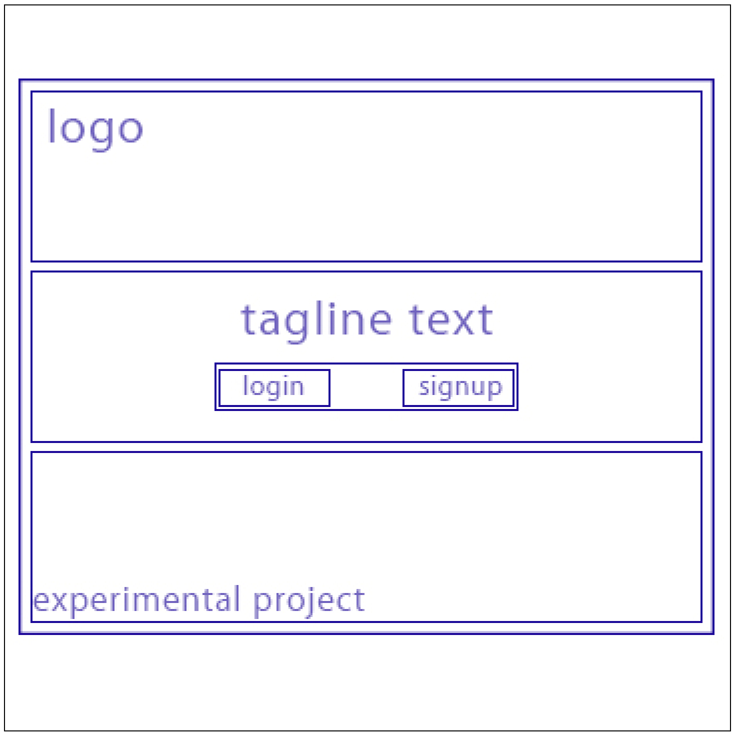
```
…
<link rel="import" href="bower_components/iron-flex-layout/classes/
iron-flex-layout.html">
…
```

Now, since we have imported the flex classes, let's create our first div: the main div of the login page, which will be the container of all elements—one container, three vertical flex children—as follows:

```
<div id="container" class="layout vertical">
  <div class="flex">
    …
  </div>
  <div class="flex">
  …
  </div>
```

```
    <div class="flex">
    …
    </div>
  </div>
```

The layout should be like this following image:



The main grid system is ready, but we need two more Polymer elements here: `<paper-button>` for the buttons in the middle, and `<iron-image>` for the logo. Those are created as follows:

```
  <div class="flex">
    <iron-image src="assets/img/logo.svg"></iron-image>
  </div>
  <div class="flex-10 layout vertical center center-justified">
    <p class="big_text">Eight days a week</p>
    <div></div>
  </div>
  <div class="flex vertical end-justified">
    <p>Experimental project</p>
  </div>
```

The classes of `<iron-flex-layout>` are awesome; you can create a grid in seconds, as in this example. The second child is much bigger than the others, and that's why I've given him the `flex-10` class—it is ten times bigger than a normal `flex`.

Seems like everything is done, but no! We have buttons here. For the buttons functionality (AJAX requesting, opening a dialog, and so on), it's better if we create our own custom elements: `<poly-login>` and `<poly-signup>`.

<poly-login>

The functionality of `<poly-login>` is to open a dialog, where we would have two inputs, for the e-mail and the password fields.

I have created a `poly-login` folder in the `assets/elements/` directory, and added `poly-login.html` inside it. Now, let's write its content:

1. We should have a button to open the dialog, so let's install `<paper-button>` and `<paper-dialog>`, using the following commands:

   ```
   $ bower install --save PolymerElements/paper-button
   $ bower install --save PolymerElements/paper-dialog
   ```

2. Now let's open `poly-login.html`, and create the component:

   ```
   <dom-module id="poly-login">
     <template>
       <paper-button raised>Login</paper-button>

       <paper-dialog id="dialog" role="dialog" entry-
   animation="scale-up-animation" exit-animation="fade-out-animation"
   with-backdrop>
         <h2>Login</h2>
         <paper-dialog-scrollable>
         </paper-dialog-scrollable>
         <div class="buttons">
           <paper-button>Login</paper-button>
         </div>
       </paper-dialog>

     </template>
     <script>
       Polymer({
         is: 'poly-login',
       });
     </script>
   </dom-module>
   ```

In the preceding code, I have created the `<dom-module>` for our `<poly-login>` element, and added the button and dialog inside it.

> Notice that I have used `<neon-animation>`, so you should install that too, and then import the animations you want.

3. Now let's add an event listener for `paper-button`, to open the dialog when the user clicks on it:

```
..
<paper-button on-click="openDialog" raised>Login</paper-button>
..
```

And in the JavaScript, it is given as follows:

```
Polymer({
  is: 'poly-login',
  openDialog: function(){
    this.$.dialog.open();
  }
});
```

4. The dialog will open when the user clicks on the button. Now let's add the inputs, and create the part for the AJAX calls.

To install `<paper-input>`, just run:

```
$ bower install --save PolymerElements/paper-input
```

Add in the dialog's `<paper-dialog-scrollable>` as follows:

```
<paper-input
  name="email"
  value="{{user.email}}"
  error-message="Please write email address"
  type="email" required
  label="Email"
  id="inputWithButton">
    <iron-icon icon="account-box" prefix></iron-icon>
</paper-input>

<paper-input
  name="password"
  value="{{user.password}}"
  error-message="Please write your password"
  required
  type="password"
  label="Password"
  id="inputWithButton">
    <iron-icon icon="lock" prefix></iron-icon>
</paper-input>
```

Add validation attributes for the inputs and the values of the `user` property as follows:

```
properties: {
  user: {
    type: Object,
    value: {}
  }
},
```

5. We need the `user` property in the AJAX request parameters. Now is the time to add a login button functionality inside the `buttons` div of the dialog:

```
  <div class="buttons">
    <paper-button on-click="login" >Login</paper-button>
  </div>
```

6. To make AJAX requests, we need to install the `<iron-ajax>` element and import that element here, somewhere in the template of our element:

```
  <template>
    <iron-ajax
    id="ajax"
    method="POST"
    url=""
    handle-as="json"
    on-response="handleResponse"></iron-ajax>
    ...
```

7. If we write some URL in the `url` attribute and give it the `auto` attribute, it will call a request when the element is loaded, but we don't need this. We need to call the request when the user clicks on the login button. The `on-response` attribute handles the response event, so we need to also create a `handleResponse` function for it, but now let's get back to our `login` event listener:

```
login: function(){
    var inputs = document.querySelectorAll('#dialog input'),
    inputs_length = inputs.length,
    is_valid = false;

    for( var i = 0; i < inputs_length; i++ ) {
      is_valid = inputs[i].validate(); // validating
      inputs[i].focus();
    }

    if(is_valid) {
      this.$.ajax.url = "api/api.php?action=login";
      this.$.ajax.params = this.user; // our user parameter

      this.$.ajax.generateRequest(); // calling the request
    }
},
```

This function will go through every input element in the dialog box, check its validity, and then perform a request to the server.

The server should answer with the user details if they exist:

```
{
  date: "2016-01-25 00:00:00"
  email: "akhxtern@gmail.com"
  id: "1"
  name: "Arshak Khachatrian"
  token: "6f0ecb9769b5a9380719b389dfd294f5"
  username: "mr_echo"
}
```

8. We need the token and user ID for all requests in the app so that we can save this user data into `sessionStorage`. So the `handleResponse` function will look like the following:

```
…
handleResponse: function(res) {
  var res = res.detail.response; // getting the response data

  if(res) {
    sessionStorage.setItem('id', res.id);
    sessionStorage.setItem('email', res.email);
    sessionStorage.setItem('token', res.token);

    window.location = 'makesomenoice'; // redirecting to the app
itself
  }
});
```

And that's it! The login functionality of *P O L Y* is ready! Now it's time for `<poly-signup>`. The functionality of the signup part is also the same, we just need to add some new inputs and names, and that's it—it is ready to use.

Don't forget to import every element in `elements.html`.

<poly-signup>

The structure of the `<poly-signup>` element is the same as `<poly-login>`. The only difference is the requesting link to the backend, and the content of the dialog (as we need four input elements):

```
<dom-module id="poly-signup">
  <template>
    <iron-ajax id="ajax" method="POST" url="" handle-as="json" on-
response="handleResponse"></iron-ajax>

    <paper-button on-click="openDialog" raised>Sign Up</paper-button>

    <paper-dialog id="dialog" role="dialog" entry-animation="scale-up-
animation" exit-animation="fade-out-animation" with-backdrop>

      <h2>Sign Up</h2>
      <paper-dialog-scrollable>

        <paper-input name="name" value="{{user.name}}" error-
message="we need your name" required label="Name">
          <iron-icon icon="face" prefix></iron-icon>
        </paper-input>

        <paper-input name="username" value="{{user.username}}"
error-message="username should be from 6 to 20 characters" required
minlength="6" maxlength="20" label="Username">
          <iron-icon icon="icons:accessibility" prefix></iron-icon>
        </paper-input>

        <paper-input name="email" value="{{user.email}}" error-
message="invalid email address" type="email" required label="Email">
          <iron-icon icon="icons:drafts" prefix></iron-icon>
        </paper-input>

        <paper-input name="password" value="{{user.password}}"
error-message="password should be from 6 to 20 characters" required
minlength="6" maxlength="20" type="password" label="Password">
          <iron-icon icon="icons:https" prefix></iron-icon>
        </paper-input>

      </paper-dialog-scrollable>

      <div class="buttons">
        <paper-button on-click="signup">Sign Up</paper-button>
      </div>
    </paper-dialog>

  </template>
  <script>
    Polymer({
      is: 'poly-signup',
      properties: {
        user: {
          type: Object,
          value: {}
        }
      },
      openDialog: function(){
        this.$.signup.open();
      },
      signup: function(){

       var inputs = this.querySelectorAll('input'),
      inputs_length = inputs.length,
      is_valid = false;

    for( var i = 0; i < inputs_length; i++ ) {
      is_valid = inputs[i].validate();
      inputs[i].focus();
    }

    if(is_valid) {
         this.$.ajax.url = "api/api.php?action=signup";
```

```
            this.$.ajax.params = this.user;

             this.$.ajax.generateRequest();
        }
          },
          handleResponse: function(res) {
            var res = res.detail.response;
            if(res) {
              sessionStorage.setItem('id', res.id);
              sessionStorage.setItem('email', res.email);
              sessionStorage.setItem('token', res.token);

              location.href = 'makesomenoice';
            }
          }
        });
      </script>
    </dom-module>
```

Done! Nothing special.

Don't forget to import all elements in the `elements.html` file, because if you do not import some of these elements, the app will not work.

Now my `elements.html` looks like this:

```
<!-- The collection of elements -->
<link rel="import" href="bower_components/iron-flex-layout/classes/
iron-flex-layout.html">
<link rel="import" href="bower_components/iron-image/iron-image.html">
<link rel="import" href="bower_components/iron-icon/iron-icon.html">
<link rel="import" href="bower_components/paper-button/paper-button.
html">
<link rel="import" href="bower_components/paper-dialog/paper-dialog.
html">
<link rel="import" href="bower_components/paper-input/paper-input.
html">
<link rel="import" href="bower_components/iron-ajax/iron-ajax.html">

<!-- Neon Animations -->
<link rel="import" href="bower_components/neon-animation/animations/
scale-up-animation.html">
<link rel="import" href="bower_components/neon-animation/animations/
scale-down-animation.html">
```

```
<link rel="import" href="bower_components/neon-animation/animations/
fade-in-animation.html">
<link rel="import" href="bower_components/neon-animation/animations/
fade-out-animation.html">

<!-- Material Icons -->
<link rel="import" href="bower_components/iron-icons/av-icons.html">
<link rel="import" href="bower_components/iron-icons/social-icons.
html">

<!-- P O L Y custom elements -->
<link rel="import" href="assets/elements/poly-login/poly-login.html">
<link rel="import" href="assets/elements/poly-signup/poly-signup.
html">
```

Now let's add our custom elements to the `index.html` file.

Just add these elements inside a div, under the tagline paragraph:

```
<div>
      <poly-login></poly-login>
      <poly-signup></poly-signup>
</div>
```

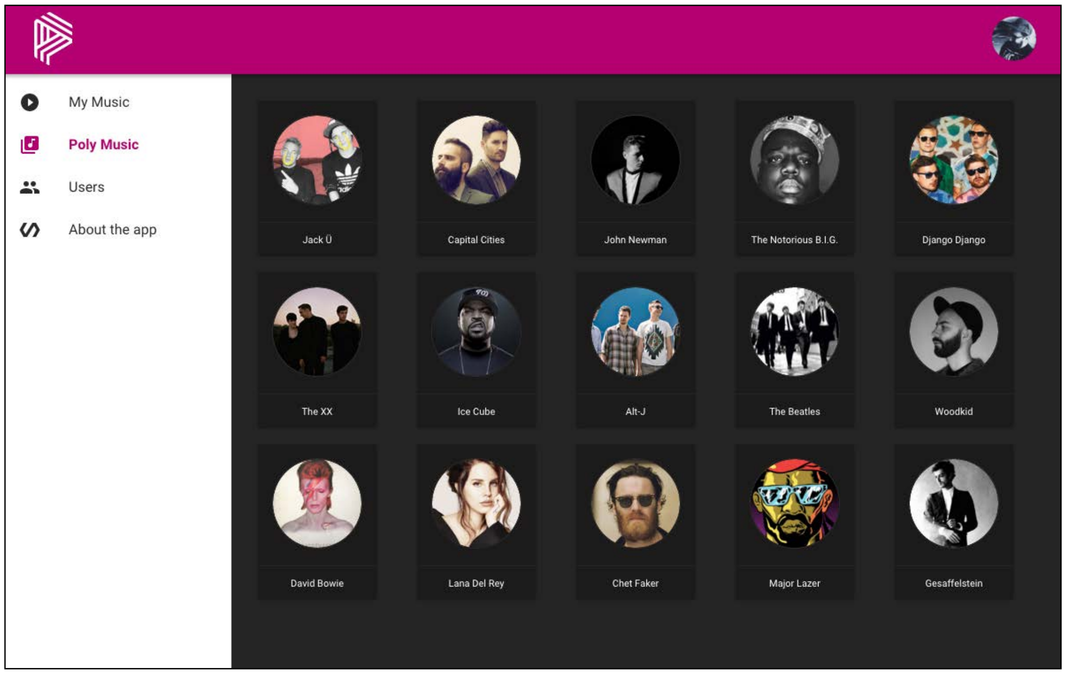
That's it! We have created our first custom elements. Cheers!

Now, let's go to `/makesomenoice`. By the way, you should write Apache rules in `.htaccess` to let `makesomenoice` open `home.html`, like this:

```
RewriteEngine On

RewriteRule ^makesomenoice/?$    home.html    [L]
```

P O L Y – the app page (home.html)

We've finished the login page, the page where the users will enter first. Now the second part of *P O L Y* is the music player, where you can see some artists and their songs and add them to your playlist. You can also see other users in *P O L Y*, visit their profiles, and listen to their music, as shown in the following image:



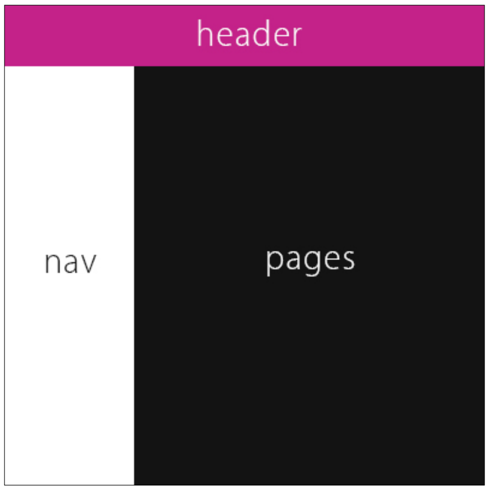
The player should have the following components:

- `<poly-profile>`: This is your profile page where you can see your profile or log out from the app
- `<poly-gravatar>`: This is the gravatar of the user
- `<poly-app>`: This is the part under the header panel, that is, the music part
- `<poly-user-music>`: This is the element for displaying the My Music section, that is, the user's music
- `<poly-daily-music>`: This is the element for showing all the artists and their music
- `<poly-songs>`: This is the flexible element for showing any song list you want to show
- `<poly-player>`: This is the audio player for *P O L Y*

- `<poly-users>`: This is the page for showing all users on *P O L Y*
- `<poly-about>`: This is the page that tells people about the app

We need to develop each of these elements to finish the app. So, let's go!

The structure of `home.html` is like this:



The main app view should have a default Material Design drawer panel, with the header panel in the top. So let's start from the top, with the header panel, that is, `<paper-header-panel>`.

In the language of Polymer, this will look like the following:

```
<div class="poly_app">
   <paper-header-panel class="pink">
   <div class="paper-header">
       <div class="layout horizontal content">
          <div class="flex">
             <iron-image class="app_logo" sizing="cover" preload
src="assets/img/poly_white.svg"></iron-image>
           </div>
           <div class="flex layout horizontal center end-justified">
         <poly-profile></poly-profile>
           </div>
          </div>
        </div>
        <div class="content">
          <div class="tabs">
```

```
                <poly-app></poly-app>
            </div>
          </div>
     </paper-header-panel>
  </div>
```

The header of the app is a simple flexbox, but we should give `<paper-header-panel>` a height:

```
    paper-header-panel {
      width: 100vw;
      height: 100vh;
    }
```

This will make the header panel fill the viewport of the user, and the app will be full screen.

Then we have `<poly-profile>`—let's start with its development.

<poly-profile>

The `<poly-profile>` element shows the user's profile data. We should have the gravatar and all details about the user inside this. Here's the code:

```
  <dom-module is="poly-profile">
    <template>

      <iron-ajax id="ajax" url="" handle-as="json" on-
  response="handleResponse"></iron-ajax>

      <poly-gravatar logout user$="{{user}}"></poly-gravatar>

      <paper-ripple></paper-ripple>

    </template>
    <script>
      Polymer({

        is: 'poly-profile',
        properties: {
          user: {
            type: Object,
            value: {}
          }
        },
        ready: function(){
      /* Requesting the user data */

          this.$.ajax.url = " api/api.php?action=get_user_data&id=" +
  sessionStorage.getItem('id');

          this.$.ajax.params = {
            userId: sessionStorage.getItem('id'),
            token: sessionStorage.getItem('token')
          };
          this.$.ajax.generateRequest();
        },
        handleResponse: function(request) {

          if(!request.detail.response) window.location = "/";
          this.user = request.detail.response;
        }
      });
    </script>
  </dom-module>
```

As you can see, the preceding code is a simple `<dom-module>`, which requests this user data, and then binds it into `<poly-gravatar>` (which we will create later). When a user clicks on it, it opens the profile page of the user.

<poly-gravatar>

The `<poly-gravatar>` element is responsible for showing the user's gravatar.

To show the gravatar, we should make an image and give it a link, but we have one problem. First, we need the user's e-mail ID. Second, we need to encode the user e-mail with the MD5 algorithm, and then bind it to the image link.

To use MD5 encoding functions in JavaScript, we need to connect `md5.min.js`, but there's no need for any additional script tags; we can just download the library into the `assets/js/` folder, and it will automatically be in JavaScript dist files.

Apart from the gravatar, we need a page to show the user data. That's why we should do some magic with CSS and HTML. We can just create a div in the `<poly-gravatar>` element, and toggle some class when the user clicks on the avatar. It will transform the div to the top, and everyone will be happy.

In the future `<poly-users>` element, we may also use `<poly-gravatar>` to show users, so that's why it is important to think about the future and include the following code:

```
<dom-module is="poly-gravatar">
  <template>

    <iron-image preload class="mini" fade on-click="togglePage"
src="https://s.gravatar.com/avatar/{{gid(user.email)}}?s=120"
sizing="cover" ></iron-image>

    <div class$="{{setClass(show)}}" id="profile_page">
      <paper-icon-button on-click="togglePage" icon="icons:arrow-
back">
        <paper-ripple></paper-ripple>
      </paper-icon-button>

      <div class="layout vertical">
        <div class="flex layout horizontal wrap">

          <paper-fab hidden$="{{check(logout)}}" id="logout_btn" on-
click="appLogout" icon="icons:exit-to-app"></paper-fab>

     <div class="flex layout horizontal self-center center-justified">
              <iron-image src="https://s.gravatar.com/avatar/
{{gid(user.email)}}?s=300" sizing="cover" ></iron-image>
          </div>

          <div class="flex layout vertical self-center center-
justified">
            <p class="user_name">{{user.name}} ({{user.username}})</p>
            <p class="user_email">{{user.email}}</p>
          </div>

        </div>
        <div class="flex-2 white">

          <poly-songs id="songs" class="with-artist" songs$="{{user.
songs}}"></poly-songs>

        </div>
      </div>
    </div>

  </template>
```

```
<script>

  Polymer({
    is: 'poly-gravatar',
    properties: {
      user: {
        type: Object,
        value: {}
      },
      show: {
        type: Boolean,
        value: false
      },
      songs: {
        type: Object,
        value: {}
      },
      logout: {
        type: Boolean,
        value: false
      }
    },

    check: function(status) {
      return !status;
    },
    togglePage: function(){
      this.show = !this.show;
    },
    setClass: function (f) {
      return f ? "open" : "";
    },
    appLogout: function(){
      sessionStorage.removeItem('id');
      sessionStorage.removeItem('email');
      sessionStorage.removeItem('token');

      location.href = "/";
    },
    gid: function(email){
      return window.md5(email);
    }
    });

  </script>
</dom-module>
```

Let's see how the preceding code works.

So in `<poly-gravatar>` we have an element called `<iron-image>`, but its `src` attribute binds the MD5 encryption of the user's e-mail address, and that's why we have a `gid` function in the prototype. It returns the encrypted e-mail address.

Then we have a div named `#profile_page`. Polymer binds an "open" class if the `show` parameter is true. When the div has an "open" class, the CSS translates the element on the *Y* axis with a 300 ms transition.

And finally, we have a `<paper-fab>` button for logging out. It is visible when the `logout` parameter is true.

I have created the `<poly-songs>` element here for the future. In the All Users page, we will have the users' profile pages as well as the listing of the current user's music. We will be back here after the `<poly-users>` element.

<poly-app>

The contents of our application, all the menus, pages, and songs, are in the `<poly-app>` element.

As you can see in the design shown at the start of this chapter, `<poly-app>` is a drawer panel, with tabs on the left and pages on the right:

```
<dom-module id="poly-app">
  <template>
    <paper-drawer-panel>
      <div drawer>
   <paper-menu selected="{{selected}}">
     <paper-icon-item>
       <paper-ripple></paper-ripple>
       <iron-icon icon="av:play-circle-filled" item-icon></iron-icon>
My Music
     </paper-icon-item>
     <paper-icon-item>
```

```
          <paper-ripple></paper-ripple>
          <iron-icon icon="av:library-music" item-icon></iron-icon>
   Poly Music
        </paper-icon-item>
        <paper-icon-item>
          <paper-ripple></paper-ripple>
          <iron-icon icon="social:group" item-icon></iron-icon>
   Users
        </paper-icon-item>
        <paper-icon-item>
          <paper-ripple></paper-ripple>
          <iron-icon icon="icons:polymer" item-icon></iron-icon>
   About the app
        </paper-icon-item>
    </paper-menu>
         </div>
         <div main>
           <iron-pages selected="{{selected}}">
        <section>
          <poly-user-music></poly-user-music>
        </section>
        <section>
          <poly-daily-music></poly-daily-music>
        </section>
        <section>
          <poly-users></poly-users>
        </section>
        <section>
          <poly-about></poly-about>
        </section>
       </iron-pages>
         </div>
       </paper-drawer-panel>

       <poly-player id="poly-player"></poly-player>
     </template>

     <script>
       Polymer({
         is: 'poly-app',
         properties: {
           selected: {
             type: Number,
            value: 1
           }
         }
       });
     </script>
   </dom-module>
```

As you can make out from the preceding code, `<poly-app>` is a simple drawer panel, with four menu items: **My Music**, **Poly Music**, **Users**, and **About the app**; each one of them is an element.

The `selected` property is created to change the drawer panel pages; it binds into `<iron-pages>` and into `<paper-menu>`.

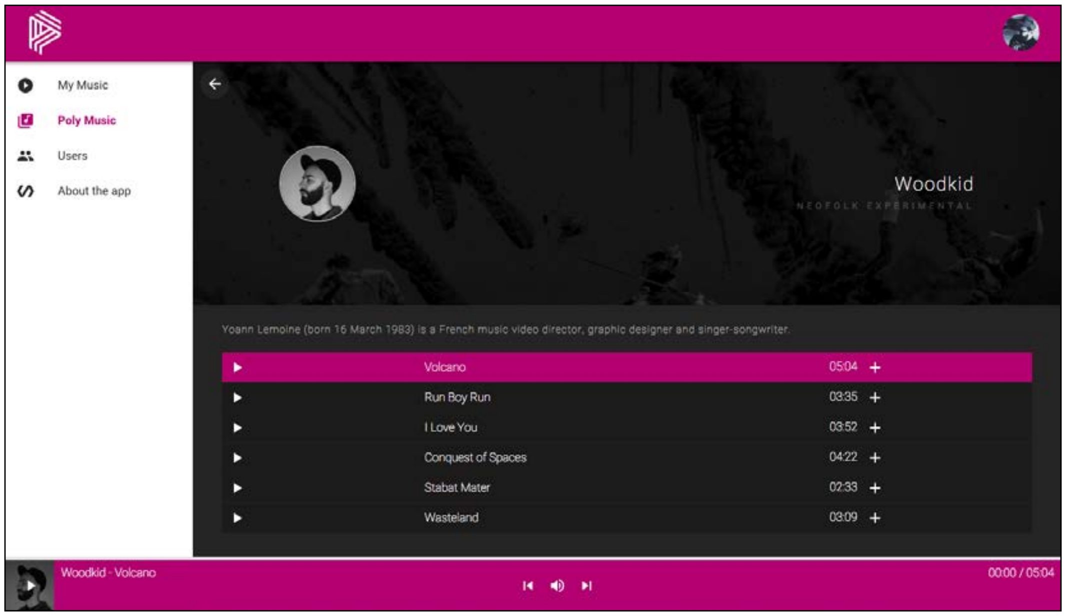
I think it will be better if we develop `<poly-daily-music>` first, because we should have some music to add into the playlist.

<poly-daily-music>

The `<poly-daily-music>` part of the app is responsible for displaying all the artists and their music. We should send a request to the backend, and get all the artists with their songs in the following format:

```
…
{
    cpic: "major-lazer-cover.jpg",
    genre: ["Electronic", "Reggae"],
    id: "2",
    info: "Major Lazer is an electronic music …",
    name: "Major Lazer",
    ppic: "major-lazer.jpg",
    songs: [
      {
        artist: "Major Lazer"
        artistId: "2"
        dir: "majorlazer-leanon.mp3"
        duration: 179
        id: "24"
        name: "Lean On"
        pic: "major-lazer.jpg"
      },
…
    ]
},
…
```

The structure of the element is very similar to `<poly-gravatar>`. There is an image, you click on it, and it animates a div from the bottom with the artist's details and songs (the songs element is `<poly-songs>`). This is depicted in the following image:



Yoann Lemoine (born 16 March 1983) is a French music video director, graphic designer and singer-songwriter.

Volcano	05:04	+
Run Boy Run	03:35	+
I Love You	03:52	+
Conquest of Spaces	04:22	+
Stabat Mater	02:33	+
Wasteland	03:09	+

Let me show you the code of the element, and then I will explain its structure and functionality:

```
<dom-module id="poly-daily-music">
  <template>

    <iron-ajax id="ajax" url="" handle-as="json" on-
response="handleResponse"></iron-ajax>

    <paper-spinner active$="{{spinnerShow}}"></paper-spinner>

    <div class="layout horizontal wrap">
      <template is="dom-repeat" items="{{artists}}">
        <div class="flex">
          <paper-card on-click="openArtistPage" class="daily-cards">
            <paper-ripple center initialOpacity="0.2"></paper-ripple>
            <div class="card-content" id="{{index}}">
              <iron-image preload fade sizing="cover" src="{{item.
ppic}}"></iron-image>
            </div>
```

```
              <div class="card-actions">
                <span>{{item.name}}</span>
              </div>
            </paper-card>
          </div>
        </template>
      </div>

      <div class$="{{setClass(show)}}" id="artist-page">

        <paper-icon-button class="artists-back" on-
click="hideArtistPage" icon="icons:arrow-back">
          <paper-ripple></paper-ripple>
        </paper-icon-button>

        <iron-image preload fade src="{{artist_page.cpic}}"
sizing="cover" class="cover-pic"></iron-image>
        <iron-image preload fade src="{{artist_page.ppic}}"
sizing="cover" class="profile-pic"></iron-image>

        <p class="artist-name">{{artist_page.name}}</p>
        <p class="artist-genre">
          <template is="dom-repeat" items="{{artist_page.genre}}">
            <span>{{item}}</span>
          </template>
        </p>
        <p class="artist-info">{{artist_page.info}}</p>

        <div class="songs-container">
          <poly-songs id="songs" songs$="{{artist_page.songs}}"></poly-
songs>
        </div>

      </div>
    </template>
    <script>
      Polymer({
        is: 'poly-daily-music',
        properties: {
          show: {
            type: Boolean,
            value: false
          },
          showArtists: {
            type: Boolean,
            value: false
          },
          spinnerShow: {
            type: Boolean,
            value: true
          },
          artists: {
            type: Array,
            value: []
          }
        },
        ready: function(){
          this.$.ajax.url = "api/api.php?action=get_artists";
          this.$.ajax.params = {
            userId: sessionStorage.getItem('id'),
            token: sessionStorage.getItem('token')
          };
          this.$.ajax.generateRequest();
        },
        handleResponse: function(res) {
          this.artists = res.detail.response;

          this.spinnerShow = false;
        },
        openArtistPage: function(e) {
          var index = e.target.getAttribute('id');
          if(!index) return;

          this.artist_page = this.artists[index];
          this.show = true;
        },
        hideArtistPage: function(e) {
          this.show = false;
        },
        setClass: function (f) {
          return f ? "open" : "";
        }
      });
    </script>
  </dom-module>
```

In the preceding code, `<iron-ajax>` requests the backend for the list of all artists in the format that I showed you earlier. In response to the AJAX call, we set the `artists` property equal to the response object— it will bind the property, and will render all the artists in the list.

We also have a spinner here, which is deactivated when the `spinnerShow` property is false (it is false when AJAX has a response value). Every artist is in the `<paper-card>` element, and every item has an index, which we need for the click event handler.

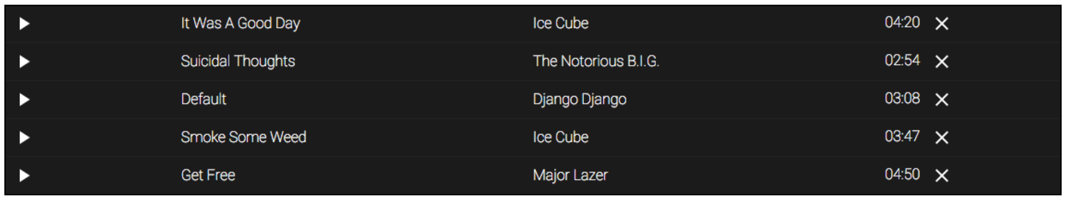
When the user clicks on an artist, we animate the div at the bottom, and bind the artist's details (profile pic, cover pic, name, genres, and so on).

Now let's move on to `<poly-songs>`. The `songs` property binds a list of the current artist's songs, and the `<poly-songs>` element shows them all. Let's take a look at `<poly-songs>`, and figure out how it shows all the songs.

<poly-songs>

`<poly-songs>` is a flexible element with a list of songs. Bind any array you want, and it will show the list you want. I love this element.

The `songs` property of this element has an observer, and when something changes in the object, it notifies us by calling the function we want. So the list gets updated when the audio player is playing or paused; it changes the statuses in real time:



▶	It Was A Good Day	Ice Cube	04:20	✕
▶	Suicidal Thoughts	The Notorious B.I.G.	02:54	✕
▶	Default	Django Django	03:08	✕
▶	Smoke Some Weed	Ice Cube	03:47	✕
▶	Get Free	Major Lazer	04:50	✕

Here is the code:

```
      <template is="dom-repeat" items="{{songs}}">
        <div class="poly-music layout horizontal" id="{{item.id}}">
          <div class="flex-2">
            <paper-icon-button on-click="playTheMusic" id="{{index}}"
   icon="av:play-arrow">
              <paper-ripple></paper-ripple>
            </paper-icon-button>
          </div>
          <div class="flex-4 layout center self-center">
            <span>{{item.name}}</span>
          </div>
          <div class="flex-4 layout center self-center artist">
            <span>{{item.artist}}</span>
          </div>
          <div class="flex-2 layout end-justified self-center">
            <span>{{readableDuration(item.duration)}}</span>
            <span>
              <paper-icon-button on-click="addToMusic" id="{{item.id}}" 
icon="add">
              </paper-icon-button>
              <paper-tooltip position="top">Add to My Music</paper-
tooltip>
            </span>
          </div>
        </div>
      </template>
```

The `<template>` shows all the songs collection that we should have in the `songs` array. When you bind a `songs` array into the attribute, Polymer gets the elements from it and binds into the template, so the template repeats the code for the length of the songs. Then we should add an `<iron-ajax>` to do some requests inside the element, and add some JavaScript here:

```
      <iron-ajax id="ajax" url="" handle-as="json" on-
response="hresponse" debounce-duration="300">

...

Polymer({
        is: 'poly-songs',
        properties: {
          songs: {
            type: Array,
            value: [],
            observer: 'songsUpdate'
          }
        },
```

We have to write a `playTheMusic` function here, which will play the music, and change the properties of the player. You can come back here after the development of the music player to check the properties that I have changed:

```
      attached: function(){
        this.songsUpdate();
      },
      playTheMusic: function(e) {
          var index = parseInt( e.target.getAttribute('id') );

          window.poly_player.song = this.songs[index];
          window.poly_player.show = true;
          window.poly_player.playStatus = false;
          window.poly_player.playPause();
          window.poly_player.playlist = this.songs;
          window.poly_player.index = index;
        },
        highlightCurrent: function(){
          var alls = document.querySelectorAll('.poly-music');
          var index = window.app.player.song.id;

          for( i = 0; i < alls.length; i++ ) {
              alls[i].querySelectorAll('paper-icon-button')[1].
setAttribute('icon', 'add');

              if(window.app.user_music)
                for( j = 0; j < window.app.user_music.length; j++ ) {
                  if(alls[i].getAttribute('id') == window.app.user_
music[j].id) {
                    alls[i].querySelectorAll('paper-icon-button')[1].
setAttribute('icon', 'clear');
                  }
                }

            if(alls[i].getAttribute('id') == index) {
              alls[i].setAttribute('current', true);

              if(window.app.player.playStatus)
                alls[i].querySelector('paper-icon-button').
setAttribute('icon', 'av:pause')
              else
                alls[i].querySelector('paper-icon-button').
setAttribute('icon', 'av:play-arrow')

            } else {
              alls[i].removeAttribute('current');
              alls[i].querySelector('paper-icon-button').
setAttribute('icon', 'av:play-arrow')
            }
          }
        },
```

The function `addToMusic` is for adding music to your playlist. When you click on the Plus button of any song, the code takes the `#id` of the song, and sends a request to the backend, which adds the `songId` and the `userId` to the database:

```
addToMusic: function(e) {
  var songId = parseInt( e.target.getAttribute('id') );

  if(e.target.getAttribute('icon') == "clear") {
    e.target.setAttribute('icon', 'clear');

    this.$.ajax.url = "api/api.php?action=remove_from_music";
  } else {

    e.target.setAttribute('icon', 'add');
    this.$.ajax.url = "api/api.php?action=add_to_music";
  }
  this.$.ajax.params = {
    "userId" : sessionStorage.getItem('id'),
    "songId" : songId,
    "token" : sessionStorage.getItem('token')
  };

  this.$.ajax.generateRequest();

  this.$.ajax.addEventListener('response', function(res){
    window.app.umusic.updateList();
  });
},
hresponse: function(){
  this.songsUpdate();
},
songsUpdate: function(){
  if(!window.poly_player) return;

  var f = this;
  setTimeout(f.highlightCurrent, 1);

},
```

The songsUpdate function is for updating the list when the user interacts with the songs and the player. As you can see, I have added the setTimeout function in the preceding code, because I first need to bind the songs inside the element, and then update the status of each song:

```
          readableDuration: function(seconds) {
              sec = Math.floor( seconds );
              min = Math.floor( sec / 60 );
              min = min >= 10 ? min : '0' + min;
              sec = Math.floor( sec % 60 );
              sec = sec >= 10 ? sec : '0' + sec;
              return min + ':' + sec;
          }
        });
    ...
```

The readableDuration function is for converting seconds into the *mm:ss* format, so if you see the preceding code snippet, each song has a property duration, which is the duration of each song in seconds.

So let's collect all that we've got here.

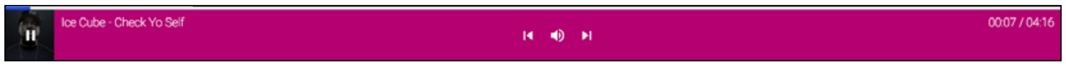
The songs property is the list that we bind into the <poly-songs> element. There are two buttons in the list: Play and Add. When the user clicks on the Play button, the function playTheMusic gets the data for the current song and "sends it" to window.poly_player, which is a <poly-player> (our custom audio player) element.

When the user clicks on the Add icon, it toggles to the "clear" icon and sends a request to the server to remove or insert the song from the user's playlist.

The highlightCurrent function highlights the current song in the audio player, and updates the status of the song in your playlist.

<poly-player>

The <poly-player> is the audio player of *P O L Y*, which becomes visible when the show property is true, as seen in the following screenshot:



The code will be as follows:

```
...
    Polymer({
      is: 'poly-player',
      properties: {
        song: {
          type: Object,
          value: {}
        },
        artist: {
          type: Object,
          value: {}
        },
        playlist: {
          type: Object,
          value: {}
        },
        currentTime: {
          type: String,
          value: "00:00"
        },
        show: {
          type: Boolean,
          value: false
        },
        playStatus: {
          type: Boolean,
          value: false,
          observer: 'playStatusChanged'
        },
        index: {
          type: Number,
          value: 0
        },
        progress: {
          type: Number,
          value: 0
        }
      }
...
playStatusChanged: function(newValue, oldValue) {
```

```
            //if(!newValue) return;

            var alls = document.querySelectorAll('poly-songs');

            for( i = 0; i < alls.length; i++ ) {
              setTimeout(alls[i].highlightCurrent, 1);
            }
        }
    ...
```

The `playStatus` property is a Boolean, which is responsive to the player status. I have added an observer on it, and in every update I call the `<poly-songs>` element's `hightlightCurrent` function to update all lists in the app. The other functions and structure of this element are simple HTML5 Audio API functions, so check out the GitHub repository of the *P O L Y* app to learn more.

That's it! The music now plays! :P But there is a problem here—you can add music to your playlist, but you can't see your playlist. Let's solve this!

<poly-user-music>

The `<poly-user-music>` page is the user's playlist. The user can add or remove any song he or she wants.

It requests the user music, and then binds it into `<poly-songs>` to show the list of the user's music:

```
    <dom-module id="poly-user-music">
      <template>

        <paper-spinner active$="{{spinnerShow}}"></paper-spinner>

        <h2>My Music</h2>
        <iron-ajax id="ajax" url="" handle-as="json" on-
    response="handleResponse"></iron-ajax>
        <poly-songs class="with-artist" songs$="{{songs}}"></poly-songs>

      </template>
      <script>
        Polymer({
          is: 'poly-user-music',
          properties: {
            spinnerShow: {
              type: Boolean,
              value: true
            }
          },
          ready: function(){

            this.songs = [];
            window.app.umusic = this;

            this.updateList();
          },
          handleResponse: function(res){
            this.songs = res.detail.response;
            window.app.user_music = this.songs;
            this.spinnerShow = false;
          },
          updateList: function(){
            this.$.ajax.url = "/api.php?action=get_user_music";
            this.$.ajax.params = {
              userId: sessionStorage.getItem('id'),
              token: sessionStorage.getItem('token')
            };
            this.$.ajax.generateRequest();
          }
        });

      </script>
    </dom-module>
```

Now you can see your music here! Add it or remove it, it's your business now.

The next step is to display all the active users of *P O L Y*. It will be interesting to know what music you are listening to.

<poly-users>

The structure of the `<poly-users>` element is similar to `<poly-daily-music>`, because it requests the list of users, and then renders it inside `<paper-cards>` with `<poly-gravatar>`:

```
  <dom-module id="poly-users">
    <template>

      <iron-ajax id="ajax" url="" handle-as="json" on-
  response="handleResponse"></iron-ajax>
      <paper-spinner active$="{{spinnerShow}}"></paper-spinner>
      <div class="layout horizontal wrap">
```

```
        <template is="dom-repeat" items="{{users}}">
          <div class="flex layout horizontal center-justified">
            <paper-card class="user-cards">
              <paper-ripple center initialOpacity="0.2"></paper-ripple>
              <div class="card-content" id="{{index}}">
                <poly-gravatar user$="{{item}}"></poly-gravatar>
              </div>
              <div class="card-actions">
                <span>{{item.name}}</span>
              </div>
            </paper-card>
          </div>
        </template>
      </div>
    </template>
    <script>
      Polymer({
        is: 'poly-users',
        properties: {
          spinnerShow: {
            type: Boolean,
            value: true
          }
        },
        ready: function(){
          this.users = [];

          this.$.ajax.url = "api/api.php?action=get_all_users";
          this.$.ajax.params = {
            userId: sessionStorage.getItem('id'),
            token: sessionStorage.getItem('token')
          };
          this.$.ajax.generateRequest();
        },
        handleResponse: function(res) {
          this.users = res.detail.response;

          this.spinnerShow = false;
        }
      });

    </script>
  </dom-module>
```

There is a list of gravatars. So when you click on the item, it opens the profile of the current user.

DONE!

You can check the online version of the app at `http://spacee.xyz/poly`.

Summary

We've come a long way in this chapter—we created a simple music application with Polymer. We also created a lot of custom elements, used the Material Design concepts to make the app look beautiful, and used our creativity to make something cool called *P O L Y*.

By the way, if you have created any reusable custom element, then you can add your element on GitHub, or even better, on the Custom Elements website (`http://customelements.io`). Feel free to create cool stuff for humanity!

Check if we got the same result or not? Tweet me your results.

> You can also check the *P O L Y* repository on GitHub at `https://github.com/AKHXtern/poly`.
> Drop me a line, or something!

In the next chapter, we will take a deep look at how to make a simple app without coding in Polymer Designer Tool and how to get started with Polymer Starter Kit.

6 Polymer Designer Tool and Polymer Starter Kit

In the previous chapters, you've learned a lot of concepts of Material Design, about web components, and about Polymer elements, and have built an app using all these skills.

The Polymer team is working hard to make people's lives easier and build complex applications with Polymer. Polymer Designer Tool and Polymer Starter Kit are now created by the team, which we will cover in this chapter, as follows:

- Polymer Designer Tool
    - Right-hand side panel
    - Top panel
    - Using Polymer elements
    - Toolbar properties and styles
    - Changing the code
- Polymer Starter Kit
    - Installation
    - Usage

Polymer Designer Tool

Polymer Designer Tool is a kind of Polymer app builder, a UI designer for HTML and custom elements.

A big plus of the Designer Tool is that designers or other non-developer people who don't want/know how to code can make some cool Material Design apps with this tool. If your designer is getting angry with you, just show them this tool and tell them to make the app themselves. (I'm joking...we need developers).

To enter Polymer Designer Tool, go to `polymer-designer.appspot.com`.

As you can see on the webpage, there's a big blank area, a toolbar at the top, and one on the right-hand side. We will talk about each area here:

- **The blank area**: This is the view where you build your Polymer app, and you can drag elements and drop them here.
- **The top panel**: This panel is responsible for the application; here, you can change the code, publish the app, and more.
- **The right panel**: This panel is the builder part. You can find a lot of elements, properties, and other stuff connected with the design here.

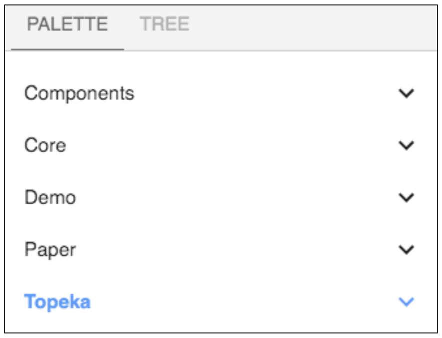
In the right-hand side panel, you can see two divided parts. The top one is to style the selected element and give it some properties you want (class name, ID, attributes, and so on). The bottom one is responsible for the elements themselves. So, as you can note, there are the **Palette** and **Tree** tabs.

The right-hand side panel

There are two tabs in the right-hand side panel of Polymer Designer Tool: **Palette** and **Tree**. Each one of them has a specific functionality, so let's go ahead and take a look at each tab's content.

Palette

The **Palette** tab contains all the elements you can use in Polymer. So, for example, **Components** contains elements about Google Maps. You can add the map, then add some direction on it, and finally add the Google Maps search bar. A simple app just in seconds! That's pretty cool.



In the **Core** item, you can see all the core elements we've used before from the Polymer core elements collection: menus, header and drawer panels, icons, and more.

The **Demo** tab is for the already-built-in demos from the Polymer team. You can add them into your app and make them play by your rules.

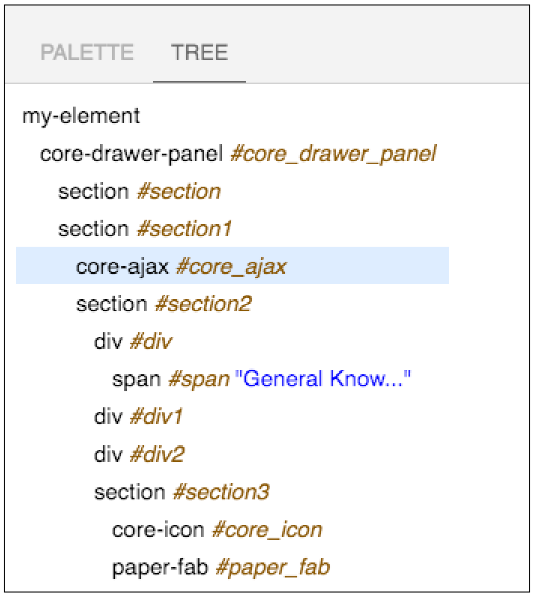
The **Paper** tab contains all the elements from the paper elements of Polymer. By adding the buttons, ripples, and other paper stuff, you can easily make your imagination come true (of course, for big and complex apps, you need to code a bit). However, as I mentioned, this tool is for creating simpler apps.

In the **Topeka** tab, you can get some elements that were used in the Topeka game that was created with Polymer: SVG icons, illustrations, and some demos.

Tree

The **Tree** tab is the tree (or sitemap) of your app. You can take a look at which components you have added in the app, and you can easily remove unwanted components from the app.

It's a very useful thing when you have a bunch of components in the app.



```
my-element
  core-drawer-panel #core_drawer_panel
    section #section
    section #section1
      core-ajax #core_ajax
      section #section2
        div #div
          span #span "General Know..."
        div #div1
        div #div2
        section #section3
          core-icon #core_icon
          paper-fab #paper_fab
```

Properties

The **Properties** section is responsible for the properties of the selected component. For example, if you are adding Google Maps, you can give the center coordinate, specify the zoom, and do some other stuff as well.

Each element has his own list of properties, so it would be good to consider them all.

Styles

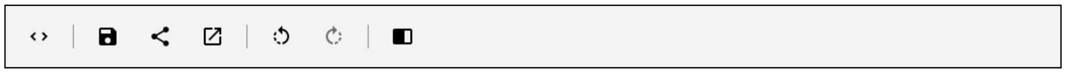
Irrespective of whether you set the styling of components from the **Styles** tab or you add manual CSS, it will add the local CSS inside your `my-element` element.

From the **Styles** tab, you can play with the colors of the element (in some of the events as well, such as hover, focus, and so on) and give them some flexbox properties, paddings, margins, and a lot of stuff related to CSS.

The **Styles** tab is one of the most useful parts of the tool. *No styles, no life*.

The top panel

The top panel is also important for us as we can change the core code of the element, save and publish the app, and preview, undo, and redo our changes (the last icon is to hide the right-hand side panel), as in the following screenshot:

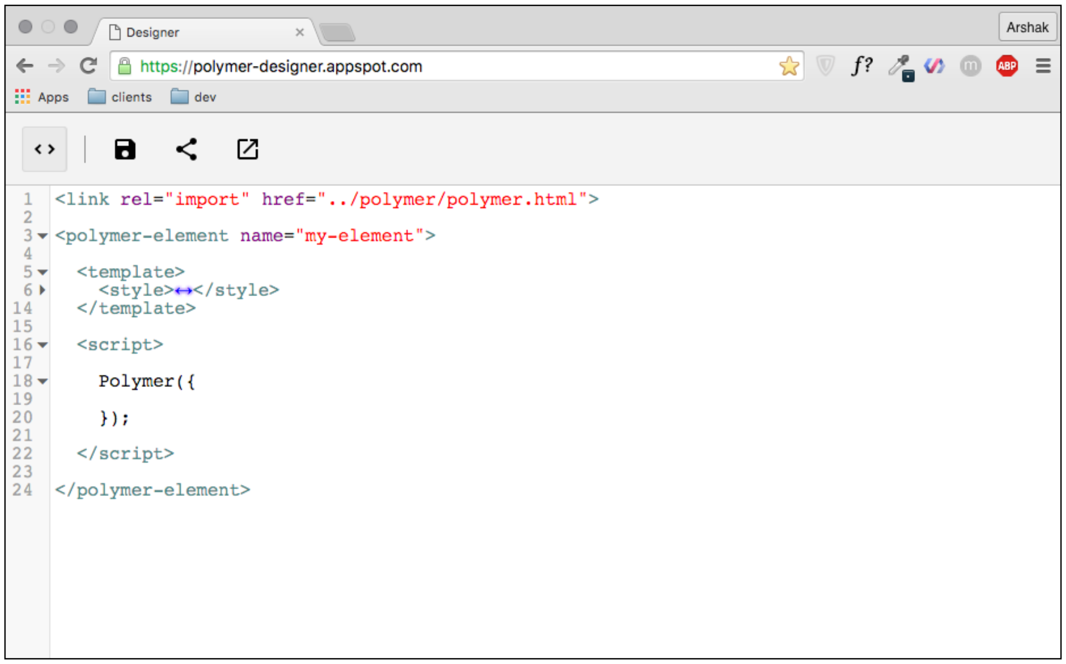


Code

As you can note in the upcoming screenshot, the code section contains the code for your custom Polymer element. Do what you want with it. It gives you a lot of opportunities to be clearer in some parts of the development process.

In my opinion, it is better and comfortable to add some AJAX calls from the code than from the UI. I don't know why! It's probably because I don't want to see any UI; however, really, isn't it cooler to write the AJAX call yourself?

Every time you change something in the viewer, the code updates itself, and you can always come back to make code changes.



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

<polymer-element name="my-element">

  <template>
    <style>↔</style>
  </template>

  <script>

    Polymer({

    });

  </script>

</polymer-element>
```

Save, share, and preview

The save, share, and preview buttons work only when you have the GitHub application's OAuth token.

If you don't have the GitHub token, you should go to your GitHub **Settings** page and generate a new token from the **OAuth applications** tab. There's no need to talk about the Redo and Undo buttons because you probably are using them every day of your life now (no matter whether you're a developer or designer).

Polymer Starter Kit

Don't know where to start the development of the application from? It's solved. Polymer Starter Kit is here. It has all the points you need to create the app. It is the starting point of your app.

It includes the following:

- The Core, Paper, Iron, and Neon elements
- The Material Design layout
- Unit testing
- End-to-end build tooling
- Routing with Page.js

Installation

To install the PSK, you should download the latest version from the GitHub repository. The version I am working with is 1.3.0.

There are two packages of PSK: the light and the full. The difference between these two packages is that the light version has a simple structure of the starter app, while the complete one has a build process and developer tooling.

The link to download PSK is `https://github.com/PolymerElements/polymer-starter-kit/releases/latest`. Then, follow these steps:

1. Download the ZIP file, unzip it to the directory you want, and open up the terminal (or command line) for this directory.
2. Run `npm install` and `bower install` to install all the dependencies and builds.

The directory structure

The PSK directory has a style of structure, a simple standard of structure that will help you to organize the work you should do, which is as follows:

```
| --- app/
|        | --- elements/
|        | --- images/
|        | --- scripts/
|        | --- styles/
|        | --- test/
| --- docs/
| --- dist/
```

Let's take a look at each directory now:

- `app/`: This directory keeps all your development code
- `elements/`: This directory keeps your custom elements
- `images/`: This is the directory for static images
- `scripts/`: This directory is for JS files
- `styles/`: This directory is for shared styles and CSS rules
- `test/`: Write your Unit Tests in this directory
- `docs/`:This directory has some docs to add features to your app
- `dist/`: This is the directory that should be deployed to the server

Build and run

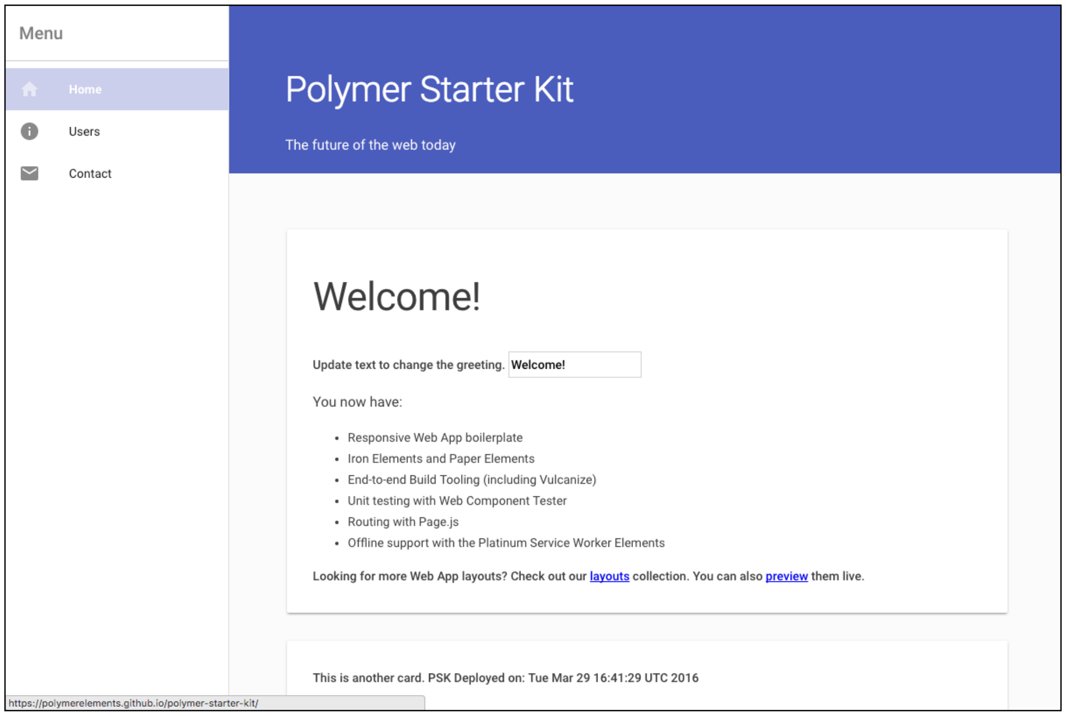
Now, when you have all your stuff with Polymer Starter Kit, you can build it and then run it locally. To build the app, you should go to the app directory using your terminal (or command line) and run the following:

```
gulp
```

Then, you should serve the app locally, as follows:

```
gulp serve
```

The app will run on the local server as `gulp serve` is running. To take a look at the result, you should open up your browser and go to `http://localhost:5000`.



Welcome!

Update text to change the greeting. Welcome!

You now have:

- Responsive Web App boilerplate
- Iron Elements and Paper Elements
- End-to-end Build Tooling (including Vulcanize)
- Unit testing with Web Component Tester
- Routing with Page.js
- Offline support with the Platinum Service Worker Elements

Looking for more Web App layouts? Check out our layouts collection. You can also preview them live.

This is another card. PSK Deployed on: Tue Mar 29 16:41:29 UTC 2016

https://polymerelements.github.io/polymer-starter-kit/

Let's make some changes in PSK; for example, let's add a menu item in the left-hand side panel and route a page to it via the following steps:

1. In `app/index.html`, find the `paper-menu` element.
2. Add a new menu item at the bottom of the menu element by executing the following code:

   ```
   <a data-route="movies" href="{{baseUrl}}movies">
       <iron-icon icon="movie_creation"></iron-icon>
       <span>Movie</span>
   </a>
   ```

   Now, when the menu item is added, we need to add the page for it. Let's go find this place.

3. In `app/index.html`, find the `iron-pages` element.

4. Add a new section inside the pages, as follows:

   ```
   <section data-route="movies">
     <paper-material elevation="1">
       <p>Heeey! It's our first page in Polymer Starter
          Kit!</p>
     </paper-material>
   </section>
   ```

5. The third step is to add the route for the `movies` page. In `app/elements/routing.html`, you can find the routing system of the app by working with Page.js, so we can add our page route here, as follows:

   ```
   page('/movies', function () {
     app.route = 'movies';
   });
   ```

And that's it! If you look at the app page of PSK now, you will see that there is a new menu item called **Movies**, and when you click on it, it opens a page connected to `movies`.

So, the principle of Polymer Starter Kit is to provide you with a starter point for the app. It uses the template of the Polymer website (a drawer panel with menu items and pages), so you can also create a similar website in seconds. However, you know how to rock it with Polymer from the previous chapter.

Summary

In this chapter, you learned about Polymer Designer Tool and Polymer Starter Kit. Each one helps us create faster and more beautiful apps, but we know that in complex apps too, they can help as a small part of them.

Let's go ahead to the next chapter, in which we will cover how to work with Polymer and Dart.

7
Working with Polymer.dart

Have you heard about Dart? Do you know what is it? Nope? That's OK! Dart is a new programming language created by Google and you will learn about how to use it with Polymer.

In this chapter, we will cover the following topics:

- What is Dart?
- Structuring an application
- Installing Dart for Polymer
- Custom elements—creation and usage
- Building an app
- Tools

What is Dart?

Dart is a new programming language created by Google in 2011 and positioned as an alternative language for JavaScript. It's a class-based, single inheritance, and object-oriented language with C syntax, which compiles into JavaScript or native code.

One of the developers of this language, Mark S. Miller, once said, "JavaScript has fundamental flaws" which are impossible to fix and that's why Dart was created.

The tasks assigned to the developers of Dart were the following:

- Create a structured and flexible language for the Web
- Make the language similar to existing ones to facilitate learning

We now have two methods to execute a Dart app—using a virtual machine or an intermediate translation in JavaScript.

So what can we do with Polymer.dart you may ask? Here's what we can do:

- Use Polymer custom elements
- Design your own custom components with styles and scripts
- Create live, two-way bindings between Dart objects and DOM nodes
- Use all standard web components—HTML imports, custom elements, Shadow DOM, and templates

Installing Dart

To install Dart on your computer, follow these steps:

- On Windows:
    1. Install Chocolatey on your Windows.
    2. Run the `choco install dart-sdk -version <version>` command.
- On Mac:
    1. Install Homebrew.
    2. Run the `brew install dart` command.
- On Linux (Debian):
    1. Run the `sudo apt-get install dart` command.

Structuring an application

The standard Polymer.dart project structure might contain a `web` folder and a `pubspec.yaml` file. It follows the **Pub Package Layout Conventions**:

- `web`: This folder is for all the app's HTML, Dart scripts, styles, and all the stuff we need for the app itself.
- `pubspec.yaml`: This is like the NPM's `package.json` file. It contains all the details and dependencies of the app.

Download Polymer.dart from `https://pub.dartlang.org/packages/polymer` and then edit your `pubspec.yaml` file to depend on the Polymer package:

```
dependencies:
  polymer: ^1.1.0
  web_components: ^0.12.0
transformers:
- polymer:
    entry_points:
web/index.html
```

The `web/index.html` file is the main file of the app.

Using custom elements in Dart

Nothing changes here. To use Polymer components, you should download the component you want and then import it with Dart.

Unfortunately, we can't use Bower here, but we can add the dependencies to the `pubspec.yaml` file instead. Just add the dependencies like this:

```
name: my_app
description: An application that uses polymer elements
dependencies:
  polymer: ">=0.15.1 <0.17.0"
  paper-button: ">=0.6.0 <0.7.0"
  paper-input: ">=0.6.0 <0.7.0"
transformers:
  polymer
```

Next, we need to run `pub get` to get all the dependencies we've mentioned in the `pubspec.yaml` file.

How to use custom elements in code

Using custom elements in code works the same way as it does in JavaScript:

1. Import the HTML file that defines the custom element.
2. Instantiate the element.
3. Initialize Polymer.

Let's look at an example of how to use Polymer components with Dart:

```
<!-- In an HTML file -->
<head>
  <link rel="import" href="packages/paper_button/
    paper_button.html">
  ...
</head>
<body unresolved>
  <paper-button raised>Hello</paper-input>
  ...
  <script type="application/dart">
    export 'package:polymer/init.dart';
  </script>
</body>
```

So what do we need to do? We need to import the component, use it in the document, and export the `init.dart` file inside the Polymer package. It's the main initialization file for the app.

You can always replace the `init.dart` file with your own Dart file. Let's create our own Dart file and initialize Polymer there.

Replace the `<script>` tag with `<script type="application/dart" src="app.dart"></script>`, so the `app.dart` that we will create now will initialize the `main()` function to run Polymer:

```
import 'package:polymer/polymer.dart';

main() {
  initPolymer().run(() {
    // all the main code related the app should be here
  });
}
```

The `initPolymer()` function has everything from Polymer and, when we call its `run()` method, it initializes all of that, so all the code we need to write is inside the callback function.

Creating custom elements in Dart

The Polymer library provides a set of features for creating custom elements. These features are designed to make it easier and faster to make custom elements that work like standard DOM elements. With a standard DOM element, you can expect the following:

- You can create it using a constructor or `document.createElement`
- You can configure it using attributes or properties
- It has some default style and can be styled from the outside
- It may provide methods to allow you to manipulate its internal state

The Dart style of custom components creation is similar to the way we know. All we need is the template and the script to run it; we'll look at an example of it.

The template part is the same, except the imports. We should import other components inside Dart.

The template `custom-component.html` will be as follows:

```
<dom-module id="custom-component">
  <style>
    /* CSS rules for your element */
  </style>
  <template>
    <!-- local DOM for your element -->
    <p>My name is {{name}}</p> <!-- data bindings in local DOM -->
  </template>
</dom-module>
```

Now comes Dart. We will create the Dart file, for example, `custom-element.dart`:

```
@HtmlImport('custom-component.html')
library my_package.custom_component;

import 'package:polymer/polymer.dart';
import 'package:web_components/web_components.dart' show HtmlImport;

@PolymerRegister(custom-component')
class ElementName extends PolymerElement {
  ElementName.created() : super.created();

  @property
  String name = 'Arshak!';
}
```

And the component is ready! As you can see in the code, we are importing the HTML template of our component, then importing the stuff we need for our element, and then registering the element. The only thing you should learn a bit about is the syntax of Dart; it's a bit strange after JavaScript, but it's just a matter of time.

You can also use the following:

- **Registration and life cycle**: These are callbacks to manage the life cycle. Use behaviors to share code
- **Declared properties**: Properties can optionally support change observers, two-way data bindings, and reflection to attributes
- **Local DOM**: This is the DOM created and managed by the element
- **Events**: These are event listeners to the local DOM children
- **Data binding**: These are property bindings and bindings to attributes
- **Behaviors**: These are reusable modules of code that can be mixed into Polymer elements

Building an app

So, now that we are done with all the development on our app, we should build it. To build the app, run `pub build` to compile your Polymer.dart app into JavaScript so that it can run across the Web.

By the way, you can specify the page to which the user should navigate by adding the path in `entry_points` in your `pubspec.yaml` file, as follows:

```
transformers:
- polymer:
    entry_points: web/index.html
```

The `pub build` command generates the `build` folder. Inside it, you can find all the HTML, CSS, and JavaScript files you need for your app.

And that's it. Your app is ready and working great!

Polymer.dart is open source. You can view and contribute to the source of Polymer.dart and its many component packages on GitHub. Make it look more awesome!

Tools

There are special tools for Dart, which will help you to develop with the Dart language.

Tools for the Sublime Text editor

For the Sublime Text editor, there's a package for Dart. You can install it from GitHub (`https://github.com/guillermooo/dart-sublime-bundle`) or from the Sublime Text package installer (`https://packagecontrol.io/installation`).

It has the following features:

- It has syntax highlighting
- It has integrated package management via `pub`
- It contains editing features—snippets, comment/uncomment, and so on
- It generates new project from the templates using stagehand
- It has an integrated source code formatter
- It runs server apps in the console
- It runs apps in any browser

Tools for Atom

For the Atom editor, there's a package called `dart-tools`. You can download it from Atom's website (`https://atom.io/packages/dart-tools`) or from the editor itself.

It has the following features:

- It has an updated grammar file
- It can compile Dart files to JS files upon saving
- It has an autocomplete feature, using the `autocomplete-plus` package
- It generates new projects using stagehand
- It shows errors quickly

Some important commands are as follows:

- `pub get`
- `pub update`
- `SDK info`
- `format code`

Summary

Let's recap what we covered in this chapter.

You learned how to download Dart, how to install it in your operating system, how to configure Polymer.dart, how to structure the project folder, and how to use Dart to create a Polymer app.

In the next chapter, we will explore the best practices of how to write clean and awesome code with high performance and with minimal bugs.

8
Best Practices

Like every programming language in the world, Polymer has its best practices for how to write clean and awesome code with high performance and with minimal bugs.

This chapter is all about best practices, so let's get going. We will take a look at a lot of cool stuff related to Polymer on the Internet and will cover some gotchas. The following are the topics we will cover:

- The mystery with `<paper-dialog>`
- How to import HTML files using RequireJS
- Floating action button with items
- The `paper-video` element
- Elements in a collection
- Yeoman Polymer Generator

The mystery with <paper-dialog>

In the course of my current work with Cambridge Semantics, Boston, we are developing a dashboard for data visualization with Polymer.

We wanted to make a dialog with dynamic content. So, we created another component as a template and created `<paper-dialog>` from paper components inside it, as shown in the following code:

```
<dom-module id="dialog-template">
    <template>
        <paper-dialog id="dialog" entry-animation="scale-up-
          animation" exit-animation="fade-out-animation">
            <h2 class="title">{{title}}</h2>
            <paper-dialog-scrollable>
```

```
                <content></content>
            </paper-dialog-scrollable>
            <div class="buttons" id="buttons">
                <paper-button dialog-dismiss>Cancel</paper-button>
                <paper-button dialog-confirm autofocus>Accept
                  </paper-button>
            </div>
        </paper-dialog>
    </template>
    ...
</dom-module>
```

Additionally, with jQuery, we did the following:

```
var $content = $('dialog-template');
$content.html(SOME_OTHER_HTML);

$(body).append($content);
```

However, something wasn't working right. When we called the JS part, it worked well the first time; it added any content that we wanted to show inside the dialog. But when we opened the dialog again (for the second time), it replaced `<paper-dialog>` with the HTML that we passed.

The problem was that, when we passed the HTML content, it was appended inside the `<content>` tag, but when we did it for the second time, there was no `<content>` tag and it replaced all the template content. So, how do we fix this problem? Hell yeah! Polymer has its own API for appending, removing, and other stuff.

We replaced the jQuery part with the following:

```
var content = document.createElement('dialog-template');

Polymer.dom(content).appendChild(SOME_OTHER_HTML);
Polymer.dom(document.body).appendChild(content);
```

And that worked! Polymer added the content that we passed inside the `<content>` tag every time it was called! That was awesome!

Actually there is a lot of stuff in the Polymer API, so you can use all of that as well.

For adding and removing children, you can use the following:

```
Polymer.dom(parent).appendChild(node)
Polymer.dom(parent).insertBefore(node, beforeNode)
Polymer.dom(parent).removeChild(node)
```

For parent and child APIs, you can use the following:

```
Polymer.dom(parent).childNodes
Polymer.dom(parent).children
Polymer.dom(node).parentNode
Polymer.dom(node).firstChild
Polymer.dom(node).lastChild
Polymer.dom(node).firstElementChild
Polymer.dom(node).lastElementChild
Polymer.dom(node).previousSibling
Polymer.dom(node).nextSibling
Polymer.dom(node).textContent
Polymer.dom(node).innerHTML
```

Importing HTML using RequireJS

If you are working with RequireJS, it is not always comfortable to use `<link rel="import" href="…">`, there are some cases you should use HTML imports using RequireJS.

There's a plugin on GitHub for HTML imports at `https://github.com/gartz/requirejs-link`.

This is all you need to do:

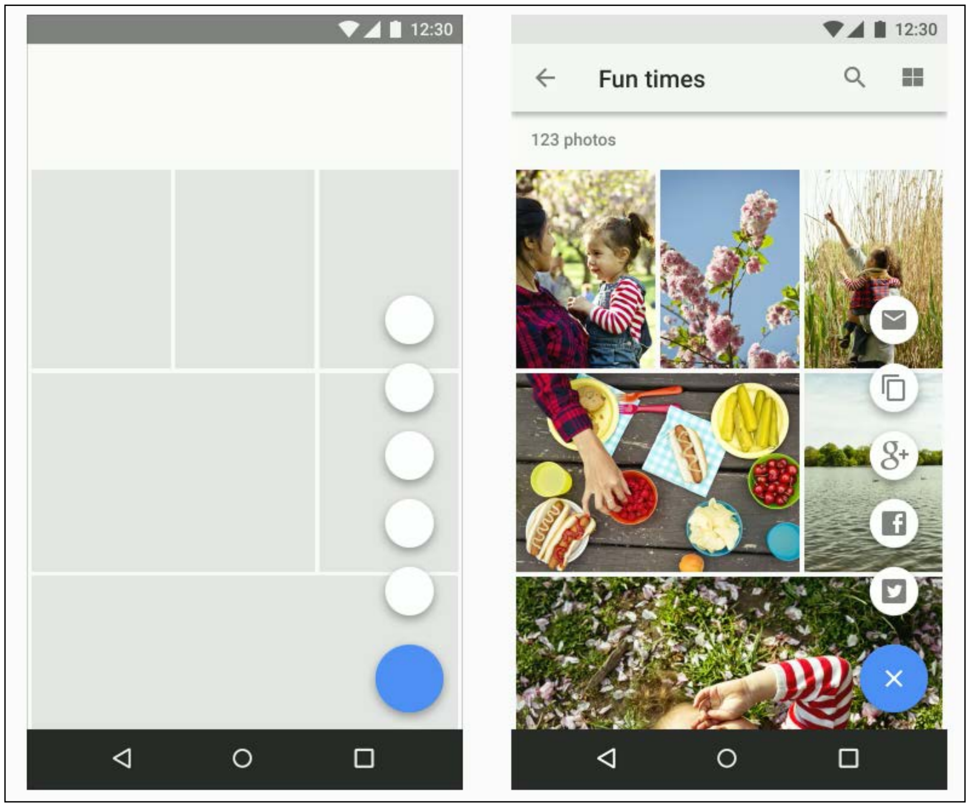
```
define(['link!my-web-component.html'], function (component) {
  console.log(component); // do some stuff with it
});
```

It's very easy to use, so I recommend you use it!

Floating action button (FAB) with menu items

I have searched the Internet, but I couldn't find any `paper-fab` components with menu items. Material Design has a concept of floating action buttons with menu items, but there's nothing in Polymer.

Check out the Material Design page at `https://www.google.com/design/spec/components/buttons-floating-action-button.html`.



So I've decided to create such an FAB and share it with the Polymer community. It uses `<paper-fab>` for the main button and `<paper-fab mini>` for the items. You can check it out on the customelements.io website at `https://customelements.io/AKHXtern/paper-fab-menu/`.

Alternatively, you can install it via Bower:

```
bower install --save paperfabmenu
```

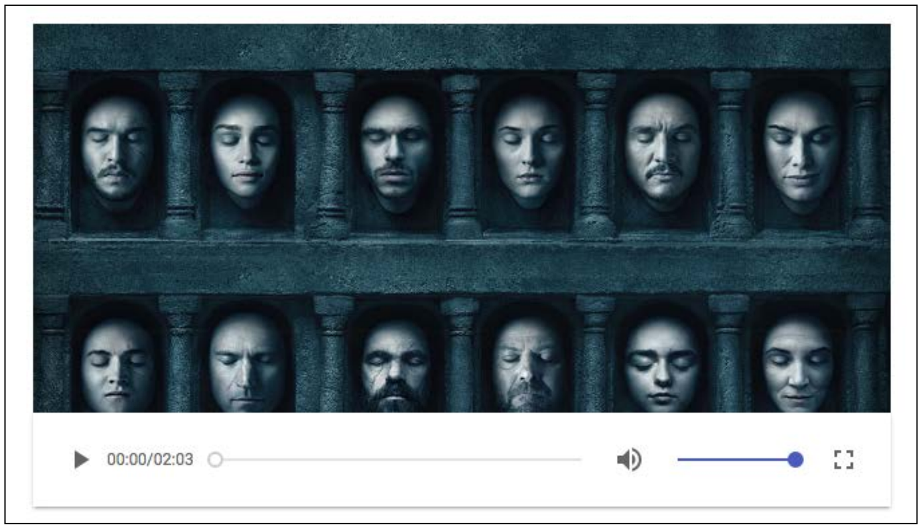
All you need to do is create the main (big) button and the items inside it, as follows:

```
<import rel="import" src="bower_components/paper-fab-menu/paper-
  fab-menu.html" />

<paper-fab-menu icon="add" position="vertical">
    <paper-fab-menu-item label="Polymer" icon="polymer" ></paper-
      fab-menu-item>
    <paper-fab-menu-item label="Favorites" icon="star" ></paper-
      fab-menu-item>
    <paper-fab-menu-item label="Refresh" icon="refresh" ></paper-
      fab-menu-item>
</paper-fab-menu>
```

You can also use the links inside `<paper-fab-menu>`.

The paper-video element

I have seen mobile versions of video players that were in Material Design styles, but I couldn't find any Polymer video players for the web. That's why I've created one and put it on GitHub:



You can find it on the customcomponents.io website at `https://customelements.io/AKHXtern/paper-video/`.

Alternatively, you can install it with Bower:

```
bower install --save paper-video
```

All you need to do is create a simple `<paper-video>` tag that has all the properties and attributes of a `<video>` tag. I've added a control autohide property for fading the controls after a period of time:

```
<import rel="import" src="bower_components/paper-video/paper-
  video.html" />

<paper-video controls autoplay autohide-controls="500"
  src="video/video.mp4">
</paper-video>
```

Elements in a collection

If you have noticed, in *Chapter 5, First Application with Polymer*, when we started creating our *P O L Y* application, we created `elements.html` for all common imports in the application.

For those who didn't notice it, you just need to create `elements.html` in the `app` folder and import it in `index.html`:

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

So, what advantages does the "one-file-of-imports" have?

- There isn't a bunch of imports inside the index file
- There's no need to import each component for every custom component
- It is just easy to read

For example, if you are using all the paper elements, you can simply write it like this:

```
elements.html

<!-- Polymer Paper elements -->
<link rel="import" href="paper-input.html" />
<link rel="import" href="paper-button.html" />
…

<!-- Polymer Iron elements -->
<link rel="import" href="iron-icons.html" />
<link rel="import" href="iron-dropdown.html" />
…
```

Use Yeoman Polymer Generator! It's awesome!

Yeoman Polymer Generator (**YPG**) provides Polymer scaffolding using Yeoman (a scaffolding tool for the Web), which lets you easily create and customize Polymer (custom) elements via the command line and import them using HTML imports. This saves the time you'd take to write boilerplate code, so you can start writing the logic for your components straight away.

I have built a lot of apps with Yeoman and Sails.js. I must say they work awesomely together. I have created full (frontend and backend) apps in a week.

With YPG, you can do the following:

- Create Polymer elements for your app
- Quickly deploy to GitHub pages
- Get `web-component-tester` support

Installation

Follow these steps to install Polymer Generator and scaffold using Yeoman:

1. Install the generator:

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

2. Make a new directory and `cd` into it:

   ```
   mkdir -p custom-project && cd $_
   ```

3. Scaffold a new Polymer project called `yo polymer`.

The polymer:element generator

To generate a new element with YPG, just run `yo polymer:element my-custom-element`. It will automatically generate the `app/elements` folder and optionally append an import to `app/elements/elements.html`, which is imported from `index.html` from your `app` folder.

The polymer:seed generator

This generates a reusable polymer element based on the seed-element workflow. This should only be used if you're creating a standalone element repo that you intend to share with others via Bower.

To preview your new element, you'll want to use the `polyserve` tool:

```
mkdir -p custom-foo && cd $_
yo polymer:seed custom-foo
polyserve
```

The polymer:gh generator

This generator generates a GitHub page branch for your seed-element.

This requires that you have SSH keys set up on GitHub:

```
cd custom-foo
yo polymer:gh
```

Testing with web-component-tester

YPG uses `web-component-tester`. To run local tests, just run the following in the terminal:

```
gulp test:local
```

Summary

And here we are at the end of the last chapter of the book. This chapter was dedicated to how you can ease your work with Polymer. We introduced you to several components created by me and introduced several instruments such as Yeoman Polymer Generator, which generates a Polymer app in a few seconds. You learned how to import HTML files using RequireJS and that it is not a good idea to use jQuery with Polymer, as you can just use the Polymer API.

You can also check out the website created for this book and add reviews at `http://spacee.xyz/polymer/`.

With all this knowledge, I can assure you that you will not have any problem with Polymer in the future. It's great for use in production of applications, it's awesome to use in small applications. But, what if there are some problems with it? Well, life is interesting with its problems. I'm always online, so you can contact me on social networks for any type of questions and I will answer them.

This is the only the chapter where I cannot say "see you in the next chapter". I hope you liked the book because I invested a lot of time to see it printed and I hope to write more such useful books in the future.

So, see you in the next book.

Index