# Web Component Architecture & Development with AngularJS

David G. Shapiro

# Web Component Architecture & Development with AngularJS

## Building reusable UI components

David Shapiro

This book is for sale at http://leanpub.com/web-component-development-with-angularjs

This version was published on 2015-03-19

# Contents

# Preface

Thank you for reading *Web Component Architecture & Development with AngularJS*. This is a work-in-progress and always will be, because the web doesn't stand still. My hope is that this will become a web community work-in-progress. The content as of 10/14 is essentially 1st draft material. I invite everyone to point out mistakes both spelling and grammar, as well as, technical. I'm also soliciting your feedback on content, and how to make it more useful for UI engineers as a practical guide for visual component architecture.

While this book is currently free to read on the web, if you choose to do so, please still click the button and make a *free* purchase. This book is being updated frequently with critical information, and the only way to be notified of major updates is if you give LeanPub your email for notifications on the purchase input form. Update notifications will be sent at most once per month, and no spam.

This book originally started when an acquisition editor for a tier one tech publisher contacted me about writing a book about AngularJS. They were looking for a "me too" book on building a web app with AngularJS. It was mid 2013. There were already some good AngularJS books on the market with many more far along in the publishing pipeline. I didn't want to write yet another AngularJS book that showed readers how to build a canned and useless "Todo" app. If I was going to commit the hundreds of hours writing a book takes, I wanted it to focus on solving real world problems that web app developers face on the job each day. One problem that I frequently run into at work is source code bloat due to multiple implementations by different developers of the same UI functionality. I had already had some good success creating re-usable widgets with AngularJS directives that could be embedded inside a big Backbone.js application. I thought that this would be a worthwhile topic and proposed it to the publisher.They agreed to my proposal, and I started writing chapters using their MS Word templates.

Early on it became apparent that the subject matter was a bit advanced, and locating technical reviewers would be a challenge for the publisher. The result was little to no feedback as I submitted each chapter. As the months wore on, it became apparent that the review and editing process could not possibly happen before the material would become stale. In the meantime, I read a blog post by Nicholas Zakas describing his experience using LeanPub for his latest JavaScript book. LeanPub is fully electronic format publishing, so the same iterative / agile processes that are used in software could be used in book writing. This works perfectly for cutting edge tech material that is constantly in flux. The downside is much less commission than for traditional distribution channels, but that did not matter because I wasn't interested in making money as a professional author. I contacted the editor, and asked for a contract termination so I could move it over to LeanPub.

Moving the content over to LeanPub was a bit painful. They use markdown for manuscripts, whereas, traditional publishers all use Word. Reformatting the content took a significant amount of time, and a good chunk of the formatting cannot be transfered because markdown is limited. The trade off is that the book can constantly evolve, errors can be addressed quickly, and there can be community participation just like any open source project.

# Release Notes

- One of the techniques used to illustrate major concepts in the original Word version of the manuscript was the use of bold font in code listings to highlight the lines of particular importance or focus as examples evolve. Unfortunately, I still have not figured out a way to make that happen in LeanPub format, or what an effective alternative would be. Also, there seems to be no bold font formatting whatsoever in the PDF version. Hopefully LeanPub will address these issues.
- There is still a one and a half chapters yet to write if anyone is curious as to why chapter 9 ends suddenly.
- There are a handful of architecture diagrams for illustrating concepts in the first third of the book which haven't been created yet.

# Conventions

The term "Web Components" is vague and used to refer to a group of W3C browser specs that are still undergoing change. Because of this it would be misleading or confusing to refer to a visual software component as a "web component" unless it made use of the Custom Element API at a bare minimum. Therefore, in this book we will only refer to a component as a **web component** if it is registered as an element using the W3C Custom Elements API. For the impatient, you can skip to chapter 10 where we discuss and construct actual Web Components with AngularJS.

For all other cases we will use the term **UI component**. This includes the components we construct to mimic web components using AngularJS 1.x.x. Eventually all existing examples will be upgraded to Web Components.

AngularJS, Web Component Specs, browser implementations, and other supporting libraries and polyfills are undergoing rapid evolution. Therefore, interim updates to sections with info that has fallen out of date will appear in these information blocks with updates and corrections pending permanent integration with the existing chapter.

# Git Repo, Bugs, and Suggestions

This source code for the first 2/3rds of this book is available on GitHub. Please create tickets for any bugs and errors that need fixing, as well as, (constructive) suggestions for improvements.

https://github.com/dgs700/angularjs-web-component-development[1]

The source code for the AngularCustomElement module used in chapter 10 is located in its own repo:

https://github.com/dgs700/angular-custom-element[2]

[1] https://github.com/dgs700/angularjs-web-component-development
[2] https://github.com/dgs700/angular-custom-element

# Introduction

## The Problem...

In web development, we are now in the era, of the "single page app". We interact with fortune 500 companies, government institutions, non-profits, online retail, and each other via megabytes of JavaScript logic injected into our web browsers. While the AJAX revolution was intended to boost the performance by which content updates are delivered to the browser by an order of magnitude, a good chunk of that gain has been eroded not just by the massive amount of badly bloated JavaScript squeezed through our Internet connections, but also by the increased amount of developer resources required to maintain such bloat. Most web applications begin life in the womb of a server rapid development framework like Rails, which hide the ugliness of request-response handling, data modeling, and templating behind a lot of convention and a bit of configuration. They start here because start-ups to Fortune 500 companies all want their web application to be launched as of yesterday. Computing resources such as RAM, CPU, and storage are cheap commodities in server land. But in the confines of the browser, they are a precious luxury. On the server, source code efficiency is not a priority, and organization is abstracted through convention. But what happens when developers used to this luxurious server environment are suddenly tasked with replicating the same business logic, routing, data modeling, and templating on the client with JavaScript? The shift in computing from server to browser over the past few years has not been very smooth.

Many organizations are finding themselves in situations where the size, performance, and complexity of their client code has become so bad that applications in the browser grind to a halt or take many seconds just to load, and they are forced to throw out their code base and start again from scratch. The term "jQuery spaghetti" is becoming quite common in the front-end community. So how did we get to this situation?

## A Brief History of Front-End development for the Web

In 2005 "AJAX" was the buzzword of the Internet. While the technology behind updating content on a web page without "reloading" was not new, it was at this time that compatibility between browser vendors had converged enough to allow the web development community to implement asynchronous content updating "en masse". The performance of content delivery via the Internet was about to take a huge leap forward.

2005 was also the dawn of the server "rapid application development framework" (RAD) with RubyOnRails taking the lead followed by CakePHP, Grails, Django and many others. Before the RAD framework, developing a web application might have taken months of coding time. RAD frameworks

reduced that to weeks, days, and even hours since much of the heavy lifting was abstracted out by following a few standard conventions. Anyone with an idea for an Internet start-up could have their website up and be selling to the public in a couple weeks thanks to Rails and company.

In 2006, a JavaScript toolkit called jQuery hit the scene. jQuery smoothed out the sometimes ugly differences in behavior between competing browser brands, and made life for front-end developers somewhat tolerable. Developers could now write one JavaScript program leveraged with jQuery for DOM interaction, and have it run in all browser brands with minor debugging. This made the concept of the "single page app" not just a possibility, but an inevitability. Assembling and rendering a visual web page, previously in the domain of the server, was now moving into the domain of the browser. All that would be left for the server to do is serve up the data necessary to update the web page. The rise of jQuery was accompanied by the rise of a "best practice" called "unobtrusive JavaScript". Using JavaScript "unobtrusively" means keeping the JavaScript (behavior) separate from the content presentation (HTML).

Back in the 90s, JavaScript in the browser was once used to add a few unnecessary bells and whistles like image rollovers and irritating status bar scrolling, as well as, the occasional form validation. It was not a language most people schooled in computer science could take seriously. Now, it is not uncommon to have a megabyte or more of logic implemented in JavaScript executing in the browser. Of course, this code became quite large and messy because established server-side programming patterns like MVC didn't always translate well from a multi-threaded, dedicated server environment to a single threaded browser DOM tree. Much of the bloat is a direct result of the "unobtrusive JavaScript best practice". Because all of the behavior related to DOM manipulation should be done outside of the HTML, the amount of code with hard binding dependencies to and from the DOM began to grow. But these dependencies tend to be situational making code reuse difficult and testability next to impossible.

A number of JavaScript libraries, toolkits, frameworks, and abstractions have risen (and fallen) along side jQuery to help us organize and structure our front-end code. YUI, Prototype, GWT, and Extjs were popular among the first wave. Some of these frameworks are built on top of jQuery, some replace it while adding their own classical OO patterns on top, and some make the attempt at removing JavaScript from our eyeballs altogether. One commonality among the first wave of JavaScript frameworks is that they were all created by, and to make life easier for, the serve-side developer. Extjs and Dojo were built on top of a classical object-oriented inheritance hierarchy for components and widgets that encouraged code-reuse to a certain extent, but often required a monolithic core, and opinionated approach to how web apps "should" be structured. YUI and Prototype were primarily jQuery alternatives for DOM manipulation. Each of these had stock widget libraries built on top.

Around 2010 or so, client-side development had progressed to the point where certain UI patterns were being described for or applied to the client-side such as Model-View-Presenter (MVP), and Model-View-ViewModel (MVVM). Both are somewhat analogous to the Model-View-Controller pattern which, for a couple decades, was the one way to approach separating and decoupling dependencies among domains of application logic related to visual presentation, business logic, and data models. The Struts library for Java, and the Rails framework for Ruby lead MVC from

the desktop to the web server domain. Component architecture was another way of separating application concerns by grouping together and encapsulating all logic related to a specific interaction within an application such as a widget. Java Server Faces (JSF) from Sun Microsystems was a popular implementation.

The newer wave of JavaScript toolkits and frameworks, as well as, the latest iterations of some of the older frameworks has incorporated aspects of MVP and MVVM patterns in their component architectures. Emberjs, Backbone.js, Knockout, and a toolkit mentioned in a few place in this book called AngularJS are very popular as of this writing. Backbone.js anchors one end of the spectrum (least opinionated). It sits on top of a DOM manipulation library, usually jQuery, and adds a few extensible, structural tools for client-side routing, REST, view logic containers, and data modeling. The developer manages bindings between data and view. While Backbone.js is often described as a framework, the reality is that it is more of a foundation upon which a developer's own framework is built. In the hands of an inexperienced front-end developer, a lot of code duplication can grow quickly in a Backbone app.

Ember.js could be described as being at the other end of the spectrum (very opinionated). While Ember.js handles management of data bindings, it requires the developer to adhere to its conventions for any aspect of an application's development. One commonality between Backbone and Ember is they have to run the show. You cannot run a Backbone app inside an Ember app, or visa verse, very easily, since both like to exert their form of control over document level interactions and APIs like deep-linking and history.

Closer to the middle, Knockout and AngularJS are both built around tight data-view binding (managed by the core) and can exist easily within apps controlled by other frameworks. AngularJS has more functionality in a smaller core, lends itself very well to creating well encapsulated UI components, and the AngularJS team, backed by Google, has announced that the development road-map will include polyfills and full compatibility with the upcoming W3C web component standards.

Reusable and well-encapsulated UI components are the key to getting the code bloat in large single page web apps under control without the need for a major re-architecture. Re-factoring a JavaScript code base bloated with jQuery spaghetti can start from the inside out. Architecting and implementing web UI components with AngularJS (and with an eye towards the emerging web components standard) is the focus of this book.

# Why AngularJS?

AngularJS is a new JavaScript toolkit having been production ready since mid-2012. It has already gained a tremendous amount of popularity among web developers and UI engineers from start-up to Fortune 500. While adoption of most JavaScript frameworks for enterprise applications are driven by such factors as commercially available support, ability to leverage existing army of Java engineers, etc. AngularJS' grassroots popularity has been supported primarily by engineers with prior frustration using other frameworks.

If you ask any front-end developer who's drunk the Angular flavored Kool-Aid, including the author, what's so great about it, they might tell you:

"I can manage my presentation behavior declaratively, directly in my HTML instead of imperatively in a second 'tree' full of jQuery bindings."

"I can use AngularJS just for parts of my page inside my app run in another framework."

"Because AngularJS encourages functional programming and dependency injection, test driven development takes almost no extra time, and my tests have a higher degree of validity and reliability."

"I can create custom HTML elements and attributes with behavior that I define."

"Any developer who wants to include my custom AngularJS plugin in their page can do so with one markup tag instead of a big JavaScript configuration object."

"I can drastically reduce the size of my code base."

"It encourages good programming habits among junior developers."

"No more hard global, application, and DOM dependencies inhibiting my testing and breaking my code"

"I can get from REST data to visual presentation much faster than the jQuery way."

## What this Book Is and Isn't

At a high level this is a book about the benefits of web UI component architecture, and AngularJS is used as the implementation framework. This is a book about solving some serious problems in front-end development and the problems are described via "user story" and use case scenarios and solved via examples using AngularJS. This book is an "attempt" to be as current and forward looking that a book on rapidly changing technology can hope to be. The Angular team intends that the toolkit will evolve towards support of the W3C set of web component standards, which is in early draft. This book will explore the proposed standards in later chapters, so code written today might be re-factored into "web components" with minimal effort a few years from now.

While some parts of this book focus exclusively on the AngularJS toolkit, this is not meant to be a comprehensive guide on all of AngularJS' features, a guide on creating a full scale web application, a description of widgets in the AngularUI project, or an AngularJS reference manual. The architectural concepts of UI components are applicable regardless of implementation framework or tool kit, and might help the developer write better code regardless of framework or lack thereof.

# Chapter 1 - Web UI Component Architectures

If this was 2007, and you mentioned the term "web component", something having to do with some kind of Java J2EE standard might come to mind. Apparently you could even get some kind of certification as a "web component" developer from Sun Microsystems. Now the term is in the process of being redefined as Java's heyday as the web development language of choice is long over.

The "component" part of "web components" is well understood in the software development community. A definition pulled from Wikipedia describes a software component as "a software package, a web service, a web resource, or a module that encapsulates a set of related functions (or data)". Another description from Wikipedia on component based software engineering:

> Component-based software engineering (CBSE) (also known as component-based development (CBD)) is a branch of software engineering that emphasizes the separation of concerns in respect of the wide-ranging functionality available throughout a given software system. It is a reuse-based approach to defining, implementing and composing loosely coupled independent components into systems. This practice aims to bring about an equally wide-ranging degree of benefits in both the short-term and the long-term for the software itself and for organizations that sponsor such software.

http://en.wikipedia.org/wiki/Component-based_software_engineering[3]

Some of goals (benefits) of a software component as described above are that they be:

- Reusable
- Portable
- Consumable
- Consistent
- Maintainable
- Encapsulated
- Loosely coupled
- Quick to implement
- Self describing (semantic)

[3] http://en.wikipedia.org/wiki/Component-based_software_engineering

Web Components are an emerging set of standards from the W3C to describe a way to create encapsulated, reusable blocks of UI presentation and behavior entirely with client side languages-HTML, JavaScript and CSS. Since 2006 or so various JavaScript widget toolkits (Dojo, ExtJS, YUI, AWT) have been doing this in one form or another. ExtJS inserts chunks of JavaScript created DOM based on widget configurations, the goal being to minimize the amount JavaScript a developer needs to know and write. AWT does the same, but starting from the server side. The Dojo toolkit adds the advantage of the developer being able to include the widgets declaratively as HTML markup. Other widget frameworks like YUI and jQuery UI required their widgets to be defined configured and included in the DOM imperatively. Most of these toolkits use a faux classical object oriented approach in which widgets can be "extensions" of other widget. All of these toolkits have drawbacks in one form or another.

Some drawbacks common to most first and second-generation JavaScript frameworks

- Excessive code devoted to DOM querying, setting up event bindings, and repaints of the DOM following certain events.
- Hard dependencies between the DOM html and JavaScript application code that requires maintenance of application logic in a dual tree fashion. This can be quite burdensome for larger apps and make testing difficult.

One common denominator of these drawbacks is poor separation of concerns and failure to follow established design patterns at the component level.

Front-end design patterns have become well established at the application level. The better application frameworks and UI developers will apply these patterns including, but not limited to, Model-View-Controller (and its variants MVVM, MVP) separation, mediator, observer or publish and subscribe, inversion of control or dependency injection. This has done a tremendous job establishing JavaScript as a programming language to be taken seriously.

What is often overlooked, however, is that these same patterns are applicable at any level of the DOM node tree, not just at the page (`<html>`, `<body>`) level. The code that comprises the widget libraries that accompany many frameworks often has poor separation of concerns, and low functional transparency *.

The same can be said of much of the front-end code produced by large enterprise organizations. The demand for experienced front-end developers has far outstripped supply, and the non-technical management that often drives web-based projects has a tendency to trivialize the amount of planning and effort involved while rushing the projects toward production. The engineers and architects tasked with these projects are typically drafted against their will from server application environments, or cheap offshore contractors are sought. The path of least resistance is taken, and the resulting web applications suffer from code bloat, minimum quality necessary to render in a browsers, bad performance, and non-reusable, unmaintainable code.

In 2009 an engineer at Google, Miško Hevery, was on a UI project that had grown out of control with regards to time and code base. As a proposed solution he created in a few weeks what would

eventually become AngularJS. His library took care of all the common boilerplate involved in DOM querying/manipulation and event binding. This allowed the code base to be drastically reduced and the engineers to focus specifically on the application logic and presentation.

This new JavaScript framework took a very different approach than the others at the time. In addition to hiding the DOM binding boilerplate, it also encouraged a high degree of functional transparency*, TDD or test-driven development, DRY meaning don't repeat yourself and allowed dynamic DOM behavior to be included in HTML markup declaratively.

*For a discussion on functional or referential transparency see http://en.wikipedia.org/wiki/Referential_-transparency_(computer_science)

One of the most useful and powerful features of AngularJS are directives. AngularJS directives are a framework convention for creating custom extensions to HTML that can encapsulate both the view and behavior of a UI component such as a site search bar in a way that can limit external dependencies, as well as, providing for very flexible configuration parameters that any application or container level code may wish to provide.

The AngularJS team state that using AngularJS is a way to overcome the primary shortcoming of HTML, mainly that it was designed for static presentation not dynamic, data driven interaction and presentation- a key feature of today's web. Because HTML has not evolved to provide dynamic presentation, JavaScript frameworks have risen to fill the gap, the most popular being jQuery.

jQuery gives us common boilerplate code for querying the DOM, binding and listening for user or browser events, and then imperatively updating or "re-painting" the DOM in response. This was great when web applications were simple. But web applications have grown very complex, and the boilerplate code can add up to thousands of lines of source mixed in with the actual application logic.

AngularJS removes the boilerplate, thus allowing JavaScript developers to opportunity to concentrate solely on the business logic of the application, and in a way that hard dependencies to the DOM are removed.

**1.0 Basic comparison of jQuery and AngularJS**

```
<!-- as a user types their name, show them their input in real time -->
<!-- the jQuery IMPERATIVE way -->
<form name="imperative">
  <label for="firstName">First name: </label>
  <input type="text" name="firstName" id="firstName">
  <span id="name-output"></span>
</form>

<script>
// we must tell jQuery to
// a) register a keypress listener on the input element
// b) get the value of the input element in each keypress
```

```
// c) append that value to the span element
// and we must maintain HTML string values in our JavaScript
$(function(){
    // boilerplate: bind a listener
    $('#firstName').keypress(function(){
        // boilerplate: retrieve a model value
        var nameValue = $('#firstName').value();
        // boilerplate: paint the model value into the view
        $('#name-output').text(nameValue);
    });
});
</script>

<!-- the AngularJS DECLARATIVE way -->
<form name="declarative">
  <label for="firstName">First name: </label>
  <!-- the next two lines create the 2-way binding between the model and view -->
  <input type="text" name="firstName" id="firstName" ng-model="firstName">
  <span>{{firstName}}</span>
</form>

<script>
// As long as AngularJS is included in the page and we provide some
// declarative annotations, no script needed!
// The input is data-bound to the output and AngularJS takes care of all
// The imperative boilerplate automatically freeing the developer to
// focus on the application.
</script>
```

AngularJS directives can be used to create re-usable, exportable UI components that are quite similar to the maturing W3C specification for Web Components. In fact, the AngularJS team intends for it to evolve towards supporting the new specification as browsers begin to support it.

A full section of this text title is devoted to discussion of Web Components, but for context here is the introduction from the W3C draft:

The component model for the Web ("Web Components") consists of five pieces:

1. Templates, which define chunks of markup that are inert but can be activated for use later.
2. Decorators, which apply templates based on CSS selectors to affect rich visual and behavioral changes to documents.
3. Custom Elements, which let authors define their own elements, with new tag names and new script interfaces.

4. Shadow DOM, which encapsulates a DOM subtree for more reliable composition of user interface elements.
5. Imports, which defines how templates, decorators and custom elements are packaged and loaded as a resource.

Each of these pieces is useful individually. When used in combination, Web Components enable Web application authors to define widgets with a level of visual richness and interactivity not possible with CSS alone, and ease of composition and reuse not possible with script libraries today.

See http://www.w3.org/TR/2013/WD-components-intro-20130606/ for the full introduction. Another key standard on the horizon for ECMA 7 is Object.observe(). Any JavaScript object will be able to listen and react to mutations of another object allowing for direct two-way binding between a data object and a view object. This will allow for model-driven-views (MDV). Data binding is a key feature of AngularJS, but it is currently accomplished via dirty checking at a much lower performance level. As with the Web Components specification, AngularJS will utilize Object.observe() as it is supported in browsers.

# Key Patterns in UI Component Development

For a UI component such as a search widget, social icon box, or drop down menu to be re-usable it must have no application or page level dependencies. The same example from above can illustrate this point:

**1.1 Basic comparison of jQuery and AngularJS**

```
<!-- as a user types their name, show them their input in real time -->
<!-- the jQuery IMPERATIVE way -->
<form name="imperative">
  <label for="firstName">First name: </label>
  <input type="text" name="firstName" id="firstName">
  <span id="name-output"></span>
</form>

<script>
$(function(){
    // Notice how we must make sure the ID strings in both HTML
    // and JavaScript MUST match
    $('#firstName').keypress(function(){
        // boilerplate: retrieve a model value
        var nameValue = $('#firstName').value();
        // boilerplate: paint the model value into the view
        $('#name-output').text(nameValue);
    });
```

```
});
</script>

<!-- the AngularJS DECLARATIVE way -->
<form name="declarative">
  <label for="firstName">First name: </label>
  <!-- the next two lines create the 2-way binding between the model and view -->
  <input type="text" name="firstName" id="firstName" ng-model="firstName">
  <span>{{firstName}}</span>
</form>

<script>
// Because AngularJS hides the boilerplate, no DOM string references to maintain
// between HTML and JavaScript
</script>
```

In this example the need to keep the ID reference strings synced between HTML and code results in the business logic of the function being tightly coupled to the containing page. If the ID used in the HTML template happens to change perhaps when attempting to "re-use" the code in another area of the application, the function breaks.

Most experienced UI developers would consider this to be a bad programming practice, yet large enterprise web applications or web front-ends tend to be riddled with it- some to the degree that maintenance and upgrades are more costly in time and money then just starting over.

At the most basic level, this is an example of tight coupling between model and view, or between UI component and the containing page. For "high quality" UI components, meaning those that accomplish the goals listed earlier, the UI component should not know about anything outside of itself such as the page HTML. The component should provide an API for injecting the data and configuration in to it as its only source of dependencies, and a component should never have to communicate directly with another component when it performs some action. The component should perform a discreet business function for the application, and it should keep its own concerns separated for easy maintenance.

The following software design patterns applied to component development can help insure higher quality components.

# MVC, MVP, MVVM, MVwhatever

For simplicity I'll refer to MVC, model-view-controller, to illustrate this concept. The term MVC has become somewhat polluted due to excessive miss-use by marketing professionals, and there is much debate over correct definitions that can be applied to front-end development.

However, at a high level, the pattern refers to keeping a clean separation between the code in an application or component for the view or user interface presentation, the controller or the business logic, and the model or the data representation. Clean refers to limiting interdependencies between the three layers. Code written in this fashion is easier to maintain and upgrade, as any layer can be changed without affecting the rest of the code.

An in depth discussion of MV* and other front-end design patterns are beyond the scope of this book, but are very important to understand thoroughly. For in-depth treatment of the key patterns, I recommend Addy Osmani's book Learning JavaScript Design Patterns - O'Reilly Media, Inc.

We will talk about MV* as implemented in AngularJS directives in the next chapter.

# Dependency Injection (IOC)

Dependency Injection or Inversion of Control (IOC) became popular with the Spring framework for Java. In Java applications prior to the advent of Spring, the norm was to run the application in a very "heavy weight" container such as JBOSS. By "heavy weight" I am referring to an application that included the code and libraries for every common enterprise dependency the application might need. Application level dependencies were made available to any function by direct reference because they might be needed at some point.

The reality was that many of these enterprise application dependencies often went unused in the application. Regardless, the memory footprint of the JVM required to run these applications inside these enterprise containers were in the hundreds of megabytes to gigabyte range.

One of the primary goals of the Spring Framework was to "invert control" of the application from the container to the application itself by allowing the application to pick and choose loading only the dependencies it knows its going to need. This is typically accomplished, in part, by passing in the dependency references directly as function parameters. The benefits of this approach were multi-fold but two that stand out were massive reduction in code and memory required to run the application, and code that had a much higher degree of referential transparency which translates directly to higher maintainability and testability because outside references do not need to be changed accordingly or mocked for testing.

**1.2 Dependency Injection in JavaScript**

```
<script>
/* Assume this code is run in a web browser. This function has a hard dependency\
 on its containing environment which is a webbrowser. If the global reference fo\
r console is not found or if it does not have amethod  called "log" then we get \
a fatal error. Code of this nature is difficultto maintain or port. */
function containerDependent(){
    // console is a hard, global dependency - not good
    console.log( 'a message' );
}
```

```
/*This function has its dependency explicitly injected by the calling function. \
$log can refer to the console.log, stdout, /dev/null, or a testing mock. This fu\
nction is much easier to maintain and move around. It has a much higher degree o\
f functional or referential transparency. */
function dependencyInjected( $log ){
    $log('a message');
}
</script>
```

The same benefits of dependency injection apply to JavaScript and are a characteristic of functional programming style. AngularJS and its directives make heavy use of dependency injection as the preferred way of declaring what the outside dependencies for any directive might be.

# Observer and Mediator Patterns

The observer pattern or a minor variant called publish and subscribe or Pub/Sub. Is a method of indirect communication between UI components. Since “high-quality” UI components should not know about what exists and what’s happening outside of themselves, how can they communicate significant events in their lifecycle? Similarly, how can a component receive vital information necessary to it’s functioning if it cannot reference another component directly?

The solution is for a component to shout about what is does blindly, and to listen in the same way. This is one of the most fundamental patterns in front-end development. Setting up a listener for an event, and invoking a function in response to the event with some information about the event is how the JavaScript we write can respond to mouse clicks, key presses, scrolling, or any custom event that might be defined. It is also fundamental for reacting to future events such as when an AJAX call returns some data (Promises and Deferreds).

By itself, the observer pattern is quite useful. But components that are designed to perform some type of business logic, can get polluted with event handling code.

This is where the Mediator pattern becomes useful. A mediator is a function or component whose job it is to know about the other components and facilitate communication between them. Mediators often act as event busses – listening for events and invoking other functions in response to such events.

Figure 1.0 Diagram of the Mediator Pattern

# Module Pattern

The Module pattern in JavaScript in its most basic form is a self-executing function that:

- Provides variable scope encapsulation

- Imports any dependencies via function parameters
- Exports it own functionality as a return statement

**1.3 Basic Module Pattern Implementation**

```
<script>
var MyModule = ( function( dependency ) {
    var myScopedVar = dependency || {};
    return {
        //module logic
    };
})(dependency);
</script>
```

Variable scope encapsulation (functional scope) is the only current way to prevent variable declarations from polluting the global namespace causing naming collisions- a epidemic among inexperienced front-end developers.

Dependency injection via function parameter is how we avoid hard coded outside dependencies within the function or module which can reduce its maintainability and portability.

Returning a function or object that encapsulates the business logic of the module is how we include the modules functionality and make it available in our library.

This describes the basic module pattern in JavaScript, but in practice libraries have taken this pattern and extended it to provide common APIs, global registration, and asynchronous loading. See CommonJS and AMD.

http://www.commonjs.org/[4]

http://requirejs.org/docs/whyamd.html[5]

# Loose Coupling of Dependencies

These patterns are all key players in creating loosely coupled, modular front-end code that may function as a reusable UI component. AngularJS and AngularJS directives arguably employ these patterns in its UI component building blocks better than any other framework to date, and is one of the reasons for its immense popularity. The AngularJS framework API has provisions for defining discreet views, controllers, and data objects all encapsulated within a directive, which can be configured and invoked with simple declarative markup.

[4] http://www.commonjs.org/

[5] http://requirejs.org/docs/whyamd.html

**1.4 AngularJS Directive Skeleton**

```
<!---- define the directive -->
<script>
    myLibrary.directive('searchBox', ['serviceDeps', function factory($_ref){
        return {
       //view
            template: '<div class="x-search-box">' +
                '<form ng-submit="submit()">' +
                '<input ng-model="searchString" type="text"' +
                'id="x-search-input" ng-focus="focus()" ng-blur="blur()">' +
                '<div class="x-sprite-search" x-click="submit()"></div>' +
                '</form>' +
                '</div>',
            //controller
            controller: function(deps){
               //business logic here
            },
            //data
            scope: {},
            link: function postLink(scope, elm, attrs) {
                    var config = attrs.config;
            }
        };
    }]);
</script>

<!---invoke the directive -->
<search-box config=""></search-box>
```

# Summary

In this chapter we discussed UI components, some key attributes of high quality UI components, the key software design patterns that make these key attributes, and brief introductions to the new W3C Web Component specification and how we can model Web Components using AngularJS directives today.

In the next couple chapters we will dive deeper into AngularJS' implementation of these key patterns in its directive API with a practical example.

# Chapter 5 - Standalone UI Components by Example

In part one we presented a good deal of discussion concerning UI component architecture and the tools that AngularJS provides for building and encapsulating reusable UI components. However, discussion only gets us so far, and in UI development, is not very useful without corresponding implementation examples. In this chapter we will examine a hypothetical use-case that can be addressed with a discreet UI component that is importable, configurable, and reusable.

*"As a web developer for a very large enterprise organization I need to be able to quickly include action buttons on my microsite that adhere to corporate style guide standards for color, shape, font, etc, and can be configured with the necessary behavior for my application."*

Most enterprise organizations have strict branding guidelines that dictate specifications for the look and feel of any division microsites that fall under the corporate website domain. The guidelines will normally cover such item as typography, colors, logo size and placement, navigation elements and other common UI element styles in order to provide a consistent user experience across the entire organization's site. Organizations that don't enforce their global style guides end up with websites from one division to the next that can be very different both visually and behaviorally. This gives the impression to the customer that one hand of the organization doesn't know what the other hand is doing.

For a web developer in any corporate division, the task is usually left to them to create an HTML element and CSS structure that adheres to the corporate style guide which can be time consuming and detract from the task at hand. Some of the UI architects in the more web savvy organizations have made the propagation of the corporate look and feel easier for individual development teams by providing UI component pallets or menus that contain prefabricated common UI elements and components complete with HTML, CSS, images, and JavaScript behavior. A good analogy would be the CSS and JavaScript components included as part of Twitter's Bootstrap.

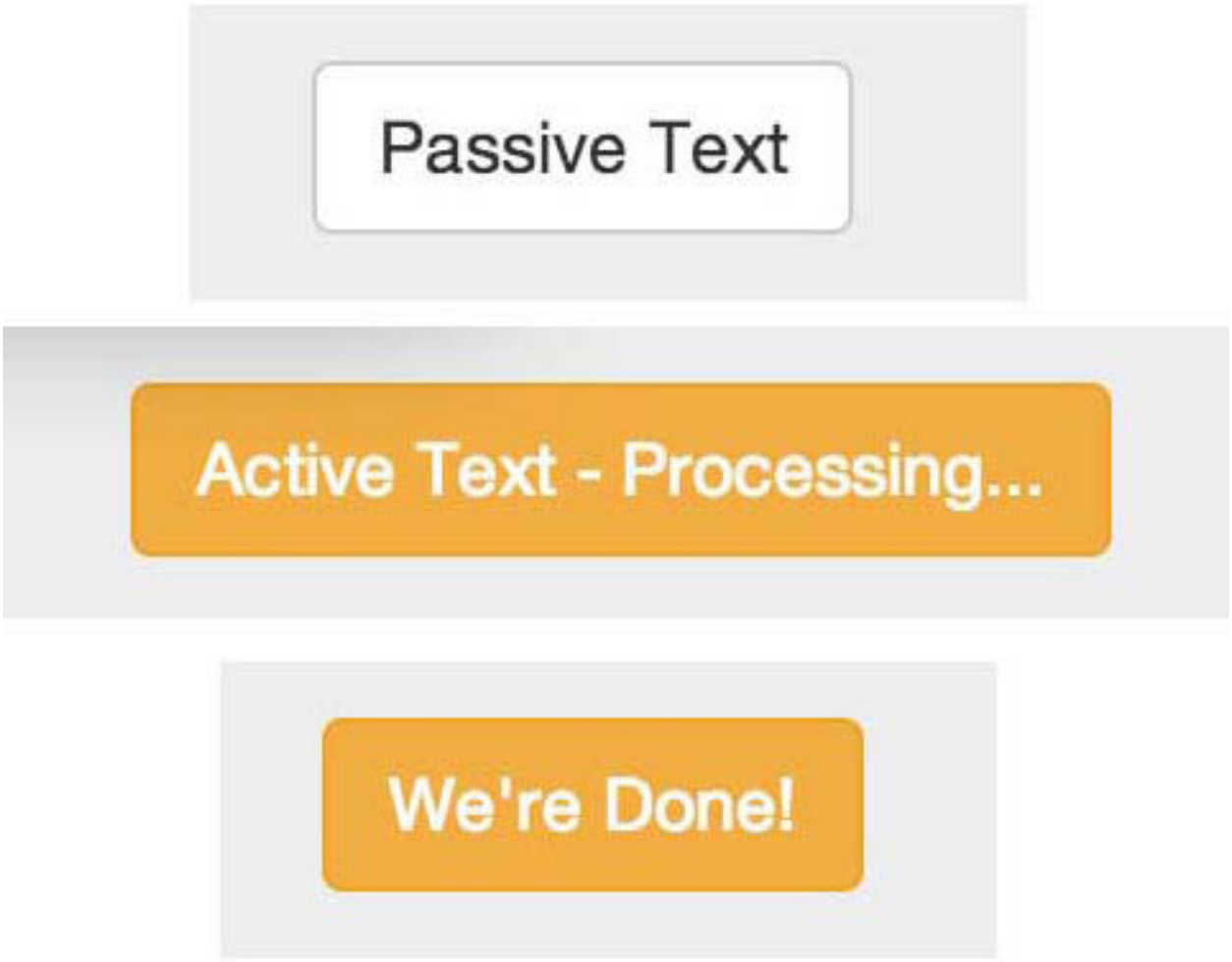
A proprietary framework often used for the above is ExtJS. Underneath ExtJS is a lot of monolithic JavaScript and DOM rewriting. It works great if used as is out of the box, but breaks down severely when styling or behavior modifications are needed. It also has serious performance problems and does not play well with other toolkits and frameworks. Such has been the price for providing customized HTML through JavaScript hacking, when the ability to create custom HTML should be available in native HTML and browser DOM methods.

AngularJS gives us the tools to create pre-built, custom HTML components in a much more declarative and natural way as if we were extending existing elements, and in a way more in line with the W3C proposed set of Web Components standards. An added advantage is much less intrusiveness towards other toolkits and frameworks.

Our example component will be, in essence, an extension to the HTML `<button>` element. We will extend it to include our corporate styling (Bootstrap.css for sake of familiarity) plus additional attributes that will serve as the element's API in the same way as existing HTML elements. We will name it "smart-button" and developers will be able to include it in their markup as `<smart-button>` along with any desired attributes. The smart-button's available attributes will allow any necessary information to be passed in, and give the button the ability to perform various, custom actions upon click. There is a lot of things a smart button could do: fire events, display pop-up messages, conditional navigation, and handle AJAX requests to name a few. This will be accomplished with as much encapsulation that AngularJS directives will allow, and with minimal outside dependencies.

While creating a "button" component might not be very complex, clever, or sexy, the point is actually to keep the functionality details reasonably simple and focus more on the methods of encapsulation, limiting hard dependencies via dependency injection and pub-sub, while providing re-usability, portability, and ease of implementation.

# Building a *Smart Button* component




**Various states of our smart button**

We'll start with a minimal component directive, and use Twitter's Bootstrap.css as the base set of available styles. The initial library file dependencies will be the most recent, stable versions of angular.min.js and bootstrap.min.css. Create a root directory for the project that can be served via any web server, and then we will wrap our examples inside Bootstrap's narrow jumbotron. Please use the accompanying GitHub repo for working code that can be downloaded. As we progress, we will build upon these base files.

**5.0 Smart Button starter directories**

```
/project_root
       smart_button.html
       /js
              UIComponents.js
              SmartButton.js
       /lib
              angular.min.js
              bootstrap.min.css
```

**5.1 Smart Button starter HTML**

```
<!DOCTYPE html>
<html ng-app="UIComponents">
<head>
    <title>A Reusable Smart Button Component</title>
    <link href="./lib/bootstrap.min.css" rel="stylesheet">
    <script src="./lib/angular.min.js"></script>
</head>
<body>
    <div class="jumbotron">
        <h1>Smart Buttons!</h1>
        <p><!-- static markup version -->
            <a class="btn btn-default">Dumb Button</a>
        </p>
        <p><!-- dynamic component version-->
            <smart-button></smart-button>
        </p>
    </div>
    <script src="./js/UIComponents.js"></script>
    <script src="./js/SmartButton.js"></script>
</body>
</html>
```

**5.2 Smart Button starter JavaScript**

```
// UIComponents.js
(function(){
    'use strict';
    // this just creates an empty Angular module

    angular.module('UIComponents',[]);
})();

// SmartButton.js directive definition
(function(){
    'use strict';
    var buttons = angular.module('UIComponents');
    buttons.directive('smartButton', function(){
        var tpl = '<a ng-class="bttnClass">{{defaultText}}</a>';
        return {
            // use an inline template for increaced
            template: tpl,
            // restrict directive matching to elements
            restrict: 'E',
            // replace entire element
            replace: true,
            // create an isolate scope
            scope: {},
            controller: function($scope, $element, $attrs){
                // declare some default values
                $scope.bttnClass = 'btn btn-default';
                $scope.defaultText = 'Smart Button';
            },
            link: function(scope, iElement, iAttrs, controller){}
        };
    });
})();
```

If we load the above HTML page in a browser, all we see are two very basic button elements that don't do anything when clicked. The only difference is that the first button is plain old static html, and the second button is matched as a directive and replaced with the compiled and linked template and scope defaults.

Naming our component "smart-button" is for illustrative purposes only. It is considered a best-practice to prepend a distinctive namespace of a few letters to any custom component names. If we create a library of components, there is less chance of a name clash with components from another

library, and if multiple component libraries are included in a page, it helps to identify which library it is from.

# Directive definition choices

## In-lining templates

There are some things to notice about the choices made in the directive definition object. First, in an effort to keep our component as self-contained as possible, we are in-lining the template rather than loading from a separate file. This works in the source code if the template string is at most a few lines. Anything longer and we would obviously want to maintain a separate source file for the template html, and then in-line it during a production build process.

In-lining templates in source JavaScript does conflict with any strategy that includes maintaining source code in a way that is forward compatible with web components structure. When the HTML5 standard `<template>` tag is finally implemented by Internet Explorer, the last holdout, then all HTML template source should be maintained in `<template>` tags.

## Restrict to element

Next, we are restricting the directive matching to elements only since what we are doing here is creating custom HTML. Doing this makes our components simple for designers or junior developers to include in a page especially since the element attributes will also serve as the component API.

## Create an Isolate Scope

Finally, we set the scope property to an empty object,{}. This creates an isolate scope that does not inherit from, is not affected by, or directly affects ancestor AngularJS scopes. We do this since one of the primary rules of component encapsulation is that *the component should be able to exist and function with no direct knowledge of, or dependency on the outside world.*

## Inline Controller

In this case we are also in-lining the code for the component's primary controller. Just like the inline template, we are doing this for increased encapsulation. The in-lined controller function should contain code, functions and default scope values that would be *common to all instances of that particular type of directive.* Also, just like with the templates, controllers functions of more than a few lines would likely best be maintained in a separate source code file and in-lined into the directive as part of a build process.

As a best-practice (in this case, DRY or don't repeat yourself), business logic that is common to different types of components on your pallet should not be in-lined, but "required" via the require

attribute in the definition object. As this does cause somewhat of an external dependency, use of *required* controllers should be restricted to situations where groups of components from the same vendor or author that use the logic are included together in dependency files. An analogy would be the required core functionality that must be included for any jQuery UI widget.

One thing of note, as of this writing, the AngularJS docs for $compile mention the use of the controllerAs attribute in the definition object as useful in cases where directives are used as components. controllerAs allows a directive template to access the controller instance (this) itself via an alias, rather than just the $scope object as is typical. There are different opinions, and a very long thread discussing this in the AngularJS Google group. By using this option, what is essentially happening is that the view is gaining direct access to the business logic for the directive. I lean toward the opinion that this is not really as useful for directives used as components as it is for junior developers new to AngularJS and not really familiar with the benefits of rigid adherence to MVVM or MV* patterns. One of the primary purposes of $scope is to expose the bare minimum of logic to view templates as the views themselves should be kept as dumb as possible.

Another thing that should be thoroughly understood is that variables and logic that are specific to any *instance* of a directive, especially those that are set at runtime should be located in the link function since that is executed after the directive is matched and compiled. For those with an object oriented background, an analogy would be the controller function contents are similar to class variables and functions, whereas, the link function contents are similar to the instance variables and functions.

# Attributes as Component APIs

It's often said that an API (application programmer interface) is the *contract* between the application provider and the application consumer. The contract specifies how the consumer can interact with the application via input parameters and returned values. Note that a component is really a mini application, so we'll use the term component from now on. What is significant about the contract is that there is a guarantee from the provider that the inputs and outputs will remain the same even if the inner workings of the component change. So while the provider can upgrade and improve parts of the component, the consumer can be confident that these changes will not break the consuming application. In essence, the component is a black box to the consumer.

Web developers interact with APIs all the time. One of the largest and most used APIs is that provided by the jQuery toolkit. But the use of jQuery's API pales in comparison to the most used API in web development, which is the API provided by the HTML specification itself. All major browsers make this API available by default. To specifically use the HTML5 API we can start an HTML document with `<!DOCTYPE html>`.

## Attributes Aren't Just Function Parameters and Return Values

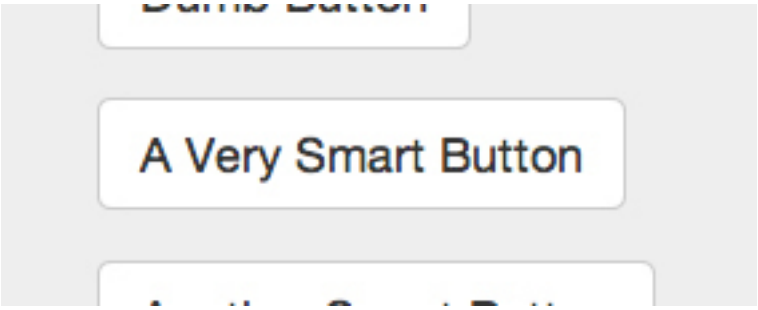
Many web developers are not aware of it, but DOM elements from the browser's point of view are actually components. Underneath the covers many DOM elements encapsulate HTML fragments

that are not directly accessible to the developer. These fragments are known as shadow DOM, and we will be hearing quite a lot about this in the next few years. We will also take an in-depth look at shadow DOM later in this book. `<input type="range">` elements are a great example. Create or locate an HTML document with one of these elements, and in Chrome developer tools check the Shadow Dom box in settings, and click on the element. It should expand to show you a greyed out DOM fragment. We cannot manipulate this DOM fragment directly, but we can affect it via the element attributes min, max, step and value. These attributes are the element's API for the web developer. Element attributes are also the most common and basic way to define an API for a custom AngularJS component.

Our smart button component really isn't all that smart just yet, let's start building our API to add some usefulness for the consumers of our component.

**5.3 Adding a button text API**

```
<!-- a custom attribute allowing text manipulation -->
<smart-button default-text="A Very Smart Button"></smart-button>
link: function(scope, iElement, iAttrs, controller){
    // <string> button text
    if(iAttrs.defaultText){
        scope.defaultText = iAttrs.defaultText;
    }
}
```




**Screen grab of the "default-text" attribute API**

The code additions in bold illustrate how to implement a basic custom attribute in an AngularJS component directive. Two things to note are that AngularJS auto camel-casing applies to our custom attributes, not just those included with AngularJS core. The other is the best-practice of always providing a default value unless a certain piece of information passed in is essential for the existence and basic functioning of our component. In the case of the later, we still need to account situations where the developer forgets to include that information and fail gracefully. Otherwise, AngularJS will fail quite un-gracefully for the rest of the page or application.

Now we've given the consumer of our component the ability to pass in a default string of text to display on our still not-to-smart button. This alone is not an improvement over what can be done with basic HTML. So, let's add another API option that will be quite a bit more useful.

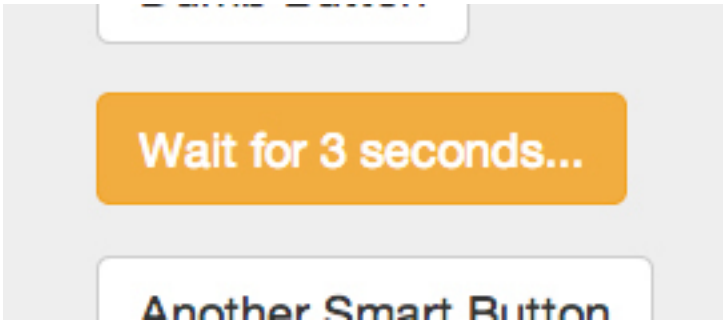
### 5.4 Adding an activetext API option

```
<!-- notice how simple and declarative our new element is -->
<smart-button
    default-text="A Very Smart Button"
    active-text="Wait for 3 seconds..."
 ></smart-button>

(function(){
    'use strict';
    var buttons = angular.module('UIComponents');
    buttons.directive('smartButton', function(){
        var tpl = '<a ng-class="bttnClass" '
            + 'ng-click="doSomething()">{{bttnText}}</a>';

        return {
            // use an inline template for increased encapsulation
            template: tpl,
            // restrict directive matching to elements
            restrict: 'E',
            replace: true,
            // create an isolate scope
            scope: {},
            controller: function($scope, $element, $attrs, $injector){

                // declare some default values
                $scope.bttnClass = 'btn btn-default';
                $scope.bttnText = $scope.defaultText = 'Smart Button';
                $scope.activeText = 'Processing...';
                // a nice way to pull a core service into a directive
                var $timeout = $injector.get('$timeout');
                // on click change text to activeText
                // after 3 seconds change text to something else
                $scope.doSomething = function(){
                    $scope.bttnText = $scope.activeText;
                    $timeout(function(){
                        $scope.bttnText = "We're Done!";
                    }, 3000);
                };
            },

            link: function(scope, iElement, iAttrs, controller){
                // <string> button text
```

```
                if(iAttrs.defaultText){
                    scope.bttnText = scope.defaultText = iAttrs.defaultText;
                }
                // <string> button text to diplay when active
                if(iAttrs.activeText){
                    scope.activeText = iAttrs.activeText;
                }
            }
        };
    });
})();
```

We have now added an API option for text to display during a temporary active state, a click event handler to switch to the active state for three seconds and display the active text, and injected the AngularJS $timeout service. After three seconds, the button text displays a "finished" message.

You can give it a run in a browser to see the result, and you definitely cannot do this with standard HTML attributes. While we are essentially just setting a delay with setTimout(), this would be the same pattern if we were to fire an AJAX request with the $http service. Both service functions return a promise object whose success() and error() function callbacks can execute addition logic such as displaying the AJAX response value data upon success, or a failure message upon an error. In the same way we've created an API for setting text display states, we could also create API attribute options for error text, HTTP configuration objects, URLs, alternative styling, and much more.




**Screen grab of the "active-text" attribute API**

Take a look at the HTML markup required to include a smart button on a page. It's completely declarative, and simple. No advanced JavaScript knowledge is required, just basic HTML making it simple for junior engineers and designers to work with.

## Events and Event Listeners as APIs

Another basic way to keep components independent of outside dependency is to interact using the observer, or publish and subscribe pattern. Our component can broadcast event and target

information, as well as, listen for the same and execute logic in response. Whereas element attributes offer the easiest method for component configuration by page developers, events and listeners are the preferred way of inter-component communication. This is especially true of widget components inside of a container component as we will explore in the next chapter.

Recall that AngularJS offers three event methods: $emit(name, args), $broadcast(name, args), and $on(name, listener). Discreet components like the smart button will most often use $emit and $on. Container components would also make use of $broadcast as we will discuss in the next chapter. Also, recall from earlier chapters that $emit propagates events up the scope hierarchy eventually to the $rootScope, and $broadcast does the opposite. This holds true even for directives that have *isolate* scopes.

A typical event to include in our API documentation under the "Events" section would be something like "smart-button-click" upon a user click. Other events could include "on-success" or "on-failure" for any component generated AJAX calls, or basically anything related to some action or process that the button handles. Keep in mind that a $destroy event is always broadcasted upon scope destruction which can and should be used for any binding cleanup.

**5.5 Events and Listeners as Component APIs**

```
// UIComponents.js
(function(){
    'use strict';

    angular.module('UIComponents',[])
    .run(['$rootScope', function($rootScope){
        // let's change the style class of a clicked smart button
        $rootScope.$on('smart-button-click', function(evt){
            // AngularJS creates unique IDs for every instantiated scope
            var targetComponentId = evt.targetScope.$id;
            var command = {setClass: 'btn-warning'};
            $rootScope
                .$broadcast('smart-button-command', targetComponentId, command);
        });
    }]);
})();

(function(){
    'use strict';

    var buttons = angular.module('UIComponents');
    buttons.directive('smartButton', function(){
        var tpl = '<a ng-class="bttnClass" '
            + 'ng-click="doSomething(this)">{{bttnText}}</a>';
```

```
        return {
            // use an inline template for increased encapsulation
            template: tpl,
            // restrict directive matching to elements
            restrict: 'E',
            replace: true,
            // create an isolate scope
            scope: {},
            controller: function($scope, $element, $attrs, $injector){
                // declare some default values
                $scope.bttnClass = 'btn btn-default';
                $scope.bttnText = $scope.defaultText = 'Smart Button';
                $scope.activeText = 'Processing...';
                // a nice way to pull a core service into a directive
                var $timeout = $injector.get('$timeout');
                // on click change text to activeText
                // after 3 seconds change text to something else
                $scope.doSomething = function(elem){
                    $scope.bttnText = $scope.activeText;
                    $timeout(function(){
                        $scope.bttnText = "We're Done!";
                    }, 3000);
                    // emit a click event
                    $scope.$emit('smart-button-click', elem);
                };
                // listen for an event from a parent container
                $scope.$on('smart-button-command', function(evt,
                            targetComponentId, command){
                    // check that our instance is the target
                    if(targetComponentId === $scope.$id){
                        // we can add any number of actions here
                        if(command.setClass){
                            // change the button style from default
                            $scope.bttnClass = 'btn ' + command.setClass;
                        }
                    }
                });
            },

            link: function(scope, iElement, iAttrs, controller){
                // <string> button text
                if(iAttrs.defaultText){
```

```
                scope.bttnText = scope.defaultText = iAttrs.defaultText;
            }
            // <string> button text to diplay when active
            if(iAttrs.activeText){
                scope.activeText = iAttrs.activeText;
            }
        }
    };
});
})();
```

In the bold sections of the preceding code, we added an event and an event listener to our button API. The event is a basic onclick event that we have named “smart-button-click”. Now any other AngularJS component or scope that is located on an ancestor element can listen for this event, and react accordingly. In actual practice, we would likely want to emit an event that is more specific to the purpose of the button since we could have several smart buttons in the same container. We could use the knowledge that we gained in the previous section to pass in a unique event string as an attribute on the fly.

The event listener we added also includes an event name string plus a unique ID and command to execute. If a smart button instance receives an event notification matching the string it checks the included $scope.$id to see if there is a match, and then checks for a match with the command to execute. Specifically in this example, if a “setClass” command is received with a value, then the smart button’s class attribute is updated with it. In this case, the command changes the Bootstrap button style from a neutral “btn-default” to an orange “btn-warning”. If the consumers of our component library are not technical, then we likely want to keep use of events to some specific choices such as setting color or size. But if the consumers are technical, then we can create the events API for our components to allow configurable events and listeners.

One item to note for this example is that our component is communicating with the AnguarJS instance of $rootScope. We are using $rootScope as a substitute for a container component since container components will be the subject of the next chapter. The $rootScope in an AngularJS application instance is similar conceptually to the global scope in JavaScript, and this has pros and cons. One of the pros is that the $rootScope is always guaranteed to exist, so if all else fails, we can use it to store data, and functions that need to be accessed by all child scopes. In other words, any component can count on that dependency always being there. But the con is that it is a brittle dependency from the standpoint that other components from other libraries have access and can alter values on it that might conflict with our dependencies.

# Advanced API Approaches

Until now we have restricted our discussion of component APIs to attribute strings and events, which are most common and familiar in the web development world. These approaches have been

available for 20 years since the development of the first GUI web browsers and JavaScript, and cover the vast majority of potential use-cases. However, some of the tools provided by AngularJS allow us to get quite a bit more creative in how we might define component APIs.

## Logic API options

One of the more powerful features of JavaScript is its ability to allow us to program in a functional style. Functions are "first-class" objects. We can inject functions via parameters to other functions, and the return values of functions can also be functions. AngularJS allows us to extend this concept to our component directive definitions. We can create advanced APIs that require logic as the parameter rather than just simple scalar values or associative arrays.

One option for using logic as an API parameter is via the scope attribute of a directive definition object with &attr:

```
scope: {
    functionCall: & or &attrName
}
```

This approach allows an AngularJS expression to be passed into a component and executed in the context of a parent scope.

**5.6 AngularJS Expressions as APIs**

```
<!-- index.html -->
<smart-button
    default-text="A Very Smart Button"
    active-text="Wait for 3 seconds..."
    debug="showAlert('a value on the $rootScope')"
></smart-button>

// UIComponents.js
(function(){
    'use strict';

    angular.module('UIComponents',[])
    .run(['$rootScope', '$window', function($rootScope, $window){
        // let's change the style class of a clicked smart button
        $rootScope.$on('smart-button-click', function(evt){
            // AngularJS creates unique IDs for every instantiated scope
            var targetComponentId = evt.targetScope.$id;
            var command = {setClass: 'btn-warning'};
            $rootScope.$broadcast('smart-button-command', targetComponentId,
```

```
                command);
        });
        $rootScope.showAlert = function (message) {
            $window.alert(message);
        };
    }]);
})();

// SmartButton.js
(function(){
    'use strict';

    var buttons = angular.module('UIComponents');
    buttons.directive('smartButton', function(){
        var tpl = '<a ng-class="bttnClass" '
            + 'ng-click="doSomething(this);debug()">{{bttnText}}</a>';

        return {
            template: tpl, // use an inline template for increaced
            restrict: 'E', // restrict directive matching to elements
            replace: true,
            // create an isolate scope
            scope: {
                debug: '&'
            },
            controller: function($scope, $element, $attrs, $injector){
                // declare some default values
                $scope.bttnClass = 'btn btn-default';
                $scope.bttnText = $scope.defaultText = 'Smart Button';
                $scope.activeText = 'Processing...';
                // a nice way to pull a core service into a directive
                var $timeout = $injector.get('$timeout');
                // on click change text to activeText
                // after 3 seconds change text to something else
                $scope.doSomething = function(elem){
                    $scope.bttnText = $scope.activeText;
                    $timeout(function(){
                        $scope.bttnText = "We're Done!";
                    }, 3000);
                    // emit a click event
                    $scope.$emit('smart-button-click', elem);
                };
```

```
                // listen for an event from a parent container
                $scope.$on('smart-button-command', function(evt,
                            targetComponentId, command){
                    // check that our instance is the target
                    if(targetComponentId === $scope.$id){
                        // we can add any number of actions here
                        if(command.setClass){
                            // change the button style from default
                            $scope.bttnClass = 'btn ' + command.setClass;
                        }
                    }
                });
            },

            link: function(scope, iElement, iAttrs, controller){
                // <string> button text
                if(iAttrs.defaultText){
                    scope.bttnText = scope.defaultText = iAttrs.defaultText;
                }
                // <string> button text to diplay when active
                if(iAttrs.activeText){
                    scope.activeText = iAttrs.activeText;
                }
            }
        };
    });
})();
```

The bold code in our contrived example shows how we can create an API option for a particular debugging option. In this case, the "debug" attribute on our component element allows our component to accept a function to execute that helps us to debug via an alert() statement. We could just as easily use the "debug" attribute on another smart button to map to a value that includes logic for debugging via a "console.log()" statement instead.

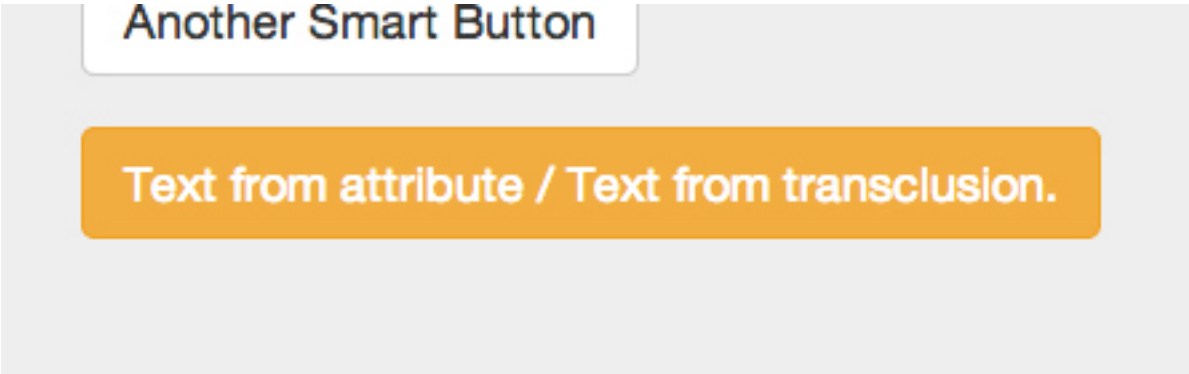
Another strategy for injecting logic into a component directive is via the require: value in a directive definition object. The value is the controller from another directive that contains logic meant to be exposed. A common use case for require is when a more direct form of communication between components than that offered by event listeners is necessary such as that between a container component and its content components. A great example of such a relationship is a tab container component and a tab-pane component. We will explore this usage further in the next chapter on container components.

## View API Options

We have already seen examples where we can pass in string values via attributes on our component element, and we can do the same with AngularJS expressions, or by including the "require" option in a directive definition object. But AngularJS also gives us a way to create a UI component whose API can allow a user to configure that component with an HTML fragment. Better yet, that HTML fragment can, itself, contain AngularJS directives that evaluate in the parent context similar to how a function closure in JavaScript behaves. The creators of AngularJS have made up their own word, *Transclusion*, to describe this concept.

**5.7 HTML Fragments as APIs - Transclusion**

```
<!-- index.html -->
<smart-button
    default-text="Text from attribute"
    active-text="Wait for 5 seconds..."
 >
    Text from transclusion.
</smart-button>

// SmartButton.js
(function(){
    'use strict';

    var buttons = angular.module('UIComponents');
    buttons.directive('smartButton', function(){
        var tpl = '<a ng-class="bttnClass"'
            + 'ng-click="doSomething(this);debug()">{{bttnText}} '
            +'<span ng-transclude></span></a>';

        return {
            template: tpl, // use an inline template for increaced
            restrict: 'E', // restrict directive matching to elements
            replace: true,
            transclude: true,
            // create an isolate scope
            scope: {
                debug: '&'
            },
            controller: function($scope, $element, $attrs, $injector){
                ...
```




**Screen shot of Smart Button component with a transcluded text node**

```
27          <p><!-- transclusion version-->
28              <smart-button
29                      default-text="Text from attribute"
30                      active-text="Wait for 5 seconds..."
31                      >
32                  / Text from transclusion.
33              </smart-button>
34          </p>
```

**The pre-compiled HTML markup utilizing element contents as an API**

While this example is about as minimalist as you can get with transclusion, it does show how the inner HTML (in this case, a simple text node) of the original component element can be transposed to become the contents of our compiled directive. Note that the ngTransclude directive must be used together with transclude:true in the component directive definition in order for transclusion to work.

When working with a button component there really isn't a whole lot that you would want to transclude, so the example here is just to explain the concept. Transclusion becomes much more valuable for the component developer who is developing UI container components such as tabs, accordions, or navigation bars that need to be filled with content.

## The Smart Button Component API

Below is a brief example of an API we might publish for our Smart Button component. In actual practice, you'd want to be quite a bit more descriptive and detailed in usage with matching examples, especially if the intended consumers of your library are anything other than senior web developers. Our purpose here is to provide the "contract" that we will verify with unit test coverage at the end of this chapter.

COMPONENT DIRECTIVE: `smartButton`

USAGE AS ELEMENT:

```
<smart-button
default-text="initial button text to display"
active-text="text to display during click handler processing"
debug="AngularJS expression"
>
"Transclusion HTML fragment or text"
</smart-button>
```

ARGUMENTS:

| Param | Type | Details |
|---|---|---|
| defaultText | string | initial text content |
| activeText | string | text content during click action |
| debug | AngularJS | expression |
| transclusion | content | HTML fragment |

EVENTS:

| Name | Type | Trigger | Args |
|---|---|---|---|
| 'smart-button-click' | $emit() | ngClick | element |
| 'smart-button-command' | $on() | | |

---

# What About Style Encapsulation?

We mentioned earlier in this book that what makes discreet UI components valuable for large organizations with very large web sites or many microsites under the same domain is the ability to help enforce corporate branding or look-and-feel throughout the entire primary domain. Instead of relying on the web developers of corporate microsites to adhere to corporate style guides and standards on their own, they can be aided with UI component pallets or menus whose components are prebuilt with the corporate look-and-feel.

While all the major browsers, as of this writing, allow web developers to create UI components with encapsulated logic and structure whether using AngularJS as the framework or not, none of the major browsers allow for the same sort of encapsulation when it comes to styling, at least not yet. The current limitation with developer supplied CSS is that there is no way to guarantee 100% that CSS rules intended to be applied only to a component will not find their way into other parts

of the web document, or that global CSS rules and styles will not overwrite those included with the component. The reality is that any CSS rule that appears in any part of the web document has the possibility to be matched to an element anywhere in the document, and there is no way to know if this will happen at the time of component development.

When the major browsers in use adopt the new standards for Web Components, specifically *Shadow DOM*, style encapsulation will be less of a concern. Shadow DOM will at least prevent component CSS from being applied beyond the component's boundaries, but it is still unknown if the reverse will be true as well. Best guess would be probably not.

08/18/2014 update - special CSS selectors are planned (unfortunately) that will allow shadow DOM CSS to be applied outside the encapsulation boundary. Conversely, global CSS will NOT traverse into shadow DOM (thankfully) unless special pseudo classes and combinators are used.

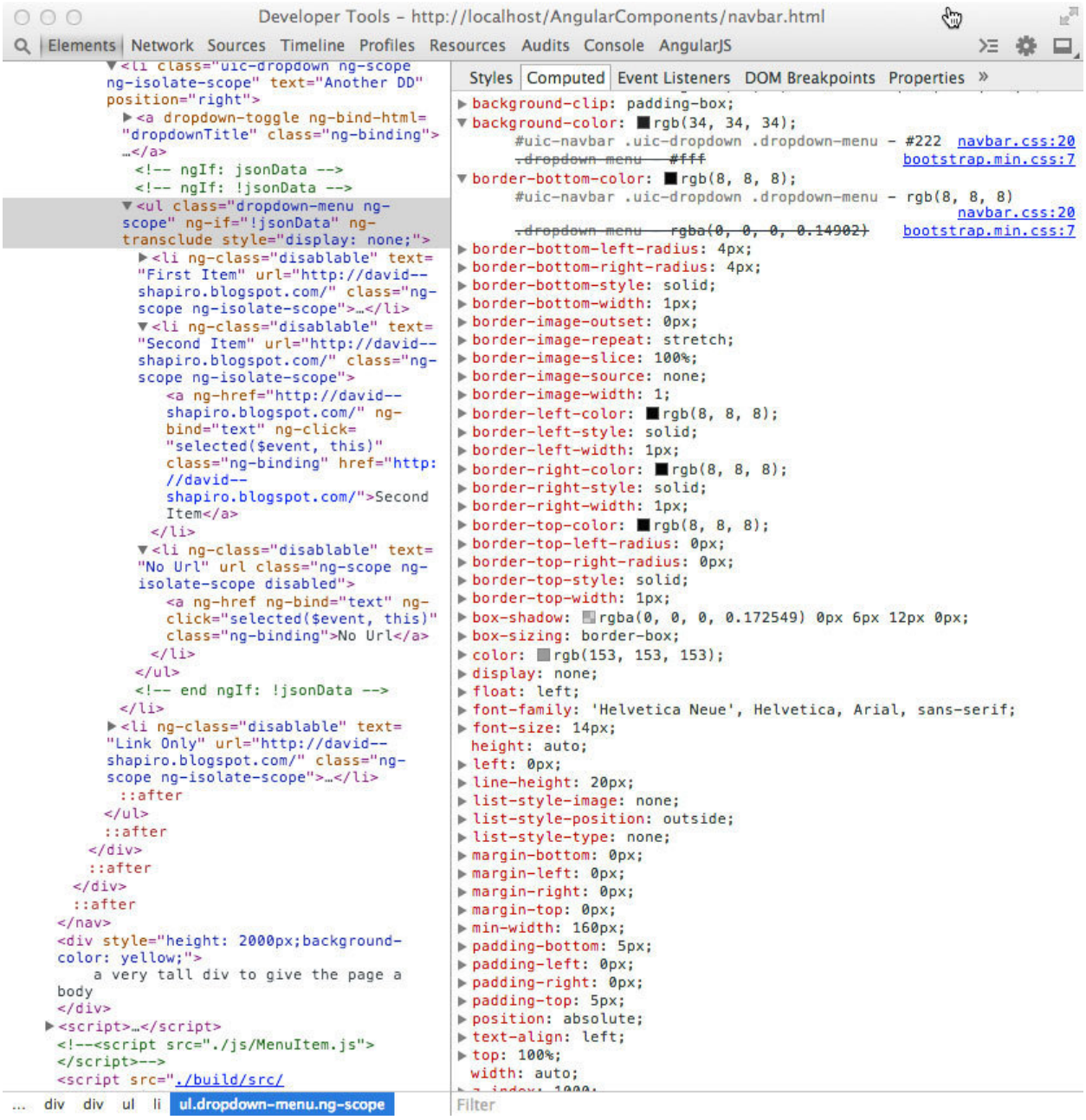
## Style Encapsulation Strategies

The best we can do is to apply some strategies with the tools that CSS provides to increase the probability that our components will display with the intended styling. In other words, we want to create CSS selectors that will most likely have the highest priority of matching just the elements in our components.

If style encapsulation is a concern, then a thorough understanding of CSS selector priority is necessary, but beyond the scope of this book. However, what is relevant to mention here is that styles that are in-lined with an element or selectors post-fixed with "!important" generally win. The highest probability that a selector will be applied is one that is in-lined and has "!important" post-fixed. For up to a few style rules this can be used. But it becomes impractical for many rules.

For maintaining a component style sheet, the suggestion would be to use a unique class or ID namespace for the top level element of the component and post fix the rules with "!important". ID namespaces provide a higher priority over class, but can only be used in situations where there will never be more than one component in a document such as a global page header or footer. The namespace will prevent component CSS from bleeding out of the component's boundaries especially when "!important" is also used.

The final challenge in attempts to provide thorough style encapsulation is that of accounting for all the possible rules that can affect the presentation of any element in a component. If thoroughness is necessary, a good strategy is to inspect all the component elements with a browser's web developer tools. Chrome, Safari, and Firefox allow you to see what all the computed styles are for an element whether the applied style rules are from a style sheet or are the browser default. Often the list can be quite long, but not all the rules necessarily affect what needs to be presented to the visitor.




**Inspection of all the applied style rules with overrides expanded**

# Unit Testing Component Directives

Every front-end developer has opinions about unit testing, and the author of this book is no exception. One of the primary goals of the AngularJS team was to create a framework that allowed development style to be very unit test friendly. However, there is often a lax attitude toward unit testing in the world of client side development.

## Validity and Reliability

At the risk of being flamed by those with heavy computer science or server application backgrounds, there are environments and situations where unit testing either doesn't make sense or is a futile endeavor. These situations include prototyping and non-application code that either has a very short shelf life or has no possibility of being reused. Unfortunately this is an often occurrence in client side development. While unit testing for the sake of unit testing is a good discipline to have, unit tests themselves need to be both valid and reliable in order to be pragmatic from a business perspective. It's easy for us propeller heads to lose site of that.

However, this is NOT the case with UI components that are created to be reused, shared and exhibit reliable behavior between versions. Unit tests that are aligned with a component's API are a necessity. Fortunately, AngularJS was designed from the ground up to be very unit test friendly.

Traditionally, unit and integration test coverage for client-side code has been difficult to achieve. This is due, in part, to a lack of awareness of the differences between software design patterns for front and back end development. Server-side web development frameworks are quite a bit more mature, and MVC is the dominant pattern for separation of concerns. It is through the separation of concerns that we are able to isolate blocks of code for maintainability and unit test coverage. But traditional MVC does not translate well to container environments like browsers which have a DOM API that is hierarchical rather than one-to-one in nature.

In the DOM different things can be taking place at different levels, and the path of least resistance for client-side developers has been to write JavaScript code that is infested with hard references to DOM nodes and global variables. The simple process of creating a jQuery selector that binds an event to a DOM node is a basic example:

```
$('a#dom_node_id').onclick(function(eventObj){
    aGlobalFunctionRef($('#a_dom_node_id'));
});
```

In this code block that is all too familiar, the JavaScript will fail if the global function or a node with a certain ID is nowhere to be found. In fact, this sort of code block fails so often, that the default jQuery response is to just fail silently if there is no match.

## Referential Transparency

AngularJS handles DOM event binding and dependencies quite differently. Event binding has been moved to the DOM/HTML itself, and any dependencies are *expected* to be injected as function parameters. As mentioned earlier, this contributes to a much greater degree of referential or functional transparency which is a fancy way of saying that our code logic can be isolated for testing without having to recreate an entire DOM to get the test to pass.

```
<a ng-click="aScopedFunctionRef()"><a>
```

```
// inside an Angular controller function
ngAppModule.controller(function( $injectedDependencyFn ){
      $scope.aScopedFunctionRef= $injectedDependencyFn;
});
```

# Setting Up Test Coverage

In this section we will focus on unit test coverage for our component directive. As with much of this book, this section is not meant to be a general guide on setting up unit test environments or the various testing strategies. There are many good resources readily available starting with the AngularJS tutorial at http://docs.angularjs.org/tutorial. Also see:

http://jasmine.github.io/[6]

http://karma-runner.github.io/[7]

https://github.com/angular/protractor[8]

https://code.google.com/p/selenium/wiki/WebDriverJs[9]

http://phantomjs.org/[10]

For our purposes, we will take the path of least resistance in getting our unit test coverage started. Here are the prerequisite steps to follow:

1. If not already available, install Node.js with node package manager (npm).
2. With root privileges, install Karma: >npm install -g karma
3. Make sure the Karma executable is available in the command path.
4. Install the PhantomJS browser for headless testing (optional).
5. Install these Karma add-ons:

   ```
   npm install -g karma-jasmine --save-dev npm install -g karma-phantomjs-launcher
   --save-dev npm install -g karma-chrome-launcher --save-dev npm install -g
   karma-script-launcher --save-dev npm install -g karma-firefox-launcher --save-dev
   npm install -g karma-junit-reporter --save-dev
   ```
6. Initialize Karma
   - Create a directory under the project root called /test
   - Run >karma init componentUnit.conf.js
   - Follow the instructions. It creates the Karma config file.
   - Add the paths to all the necessary *.js files to the config file

[6] http://jasmine.github.io/
[7] http://karma-runner.github.io/
[8] https://github.com/angular/protractor
[9] https://code.google.com/p/selenium/wiki/WebDriverJs
[10] http://phantomjs.org/

7. Create a directory called /unit under the /test directory
    - In project/test/unit/ create a file called SmartButtonSpec.js
    - Add the above path and files to componentUnit.conf.js
    - Add the angular-mocks.js file to project/lib/
8. Under /test create a launching script, test.sh, that does something like the following:

```
karma start karma.conf.js $*
```

9. Run the script. You will likely need to debug for:
    - directory path references
    - JavaScript file loading order in componentUnit.conf.js

When the karma errors are gone, and "Executed 0 of 0" appears, we are ready to begin creating our unit tests in the SmartButtonSpec.js file we just created. If the autoWatch option in our Karma config file is set to true, we can start the Karma runner once at the beginning of any development session and have the runner execute on any file change automatically.

The above is our minimalist set up list for unit testing. Reading the docs, and googling for terms like "AngularJS directive unit testing" will lead to a wealth of information on various options, approaches, and alternatives for unit test environments. Jasmine is described as behavioral testing since it's command names are chosen to create "plain English" sounding test blocks which the AngularJS core team prefers since AngularJS, itself, is meant to be very declarative and expressive. But Mocha and QUnit are other popular unit test alternatives. Similarly, our environment is running the tests in a headless Webkit browser, but others may prefer running the tests against the real browsers in use: Chrome, Firefox, Safari, Internet Explorer, etc.

# Component Directive Unit Test File

For every unit test file there are some commonalities to include that load and instantiate all of our test and component dependencies. Here is a nice unit test skeleton file with a single test.

**5.8 A Skeleton Component Unit Test File**

```
// SmartButtonSpec.js
describe('My SmartButton component directive', function () {
    var $compile, $rootScope, $scope, $element, element;
    // manually initialize our component library module
    beforeEach(module('UIComponents'));
    // make the necessary angular utility methods available
    // to our tests
    beforeEach(inject(function (_$compile_, _$rootScope_) {
        $compile = _$compile_;
        $rootScope = _$rootScope_;
        $scope = $rootScope.$new();
```

```
    }));

    // create some HTML to simulate how a developer might include
    // our smart button component in their page
    var tpl = '<smart-button default-text="A Very Smart Button" '
        + 'active-text="Wait for 3 seconds..." '
        + 'debug="showAlert(\'a value on the $rootScope\')"'
        + '></smart-button>';

    // manually compile and link our component directive
    function compileDirective(directiveTpl) {
        // use our default template if none provided
        if (!directiveTpl) directiveTpl = tpl;
        inject(function($compile) {
            // manually compile the template and inject the scope in
            $element = $compile(directiveTpl)($scope);
            // manually update all of the bindings
            $scope.$digest();
            // make the html output available to our tests
            element = $element.html();
        });
        // finalize the directive generation
    }

    // test our component initialization
    describe('the compile process', function(){
        beforeEach(function(){
            compileDirective();
        });
        // this is an actual test
        it('should create a smart button component', function(){
            expect(element).toContain('A Very Smart Button');
        });
    });
    // COMPONENT FUNCTIONALITY TESTS GO HERE
});
```

## A Look Under the Hood of AngularJS

If you don't have a good understanding of what happens under the hood of AngularJS and the directive lifecycle process, you will after creating the test boilerplate. Also, if you were wondering when many of the AngularJS service API methods are used, or even why they exist, now you know.

In order to set up the testing environment for component directives, the directive lifecycle process and dependency injection that is automatic in real AngularJS apps, must be handled manually. The entire process of setting up unit testing is a bit painful, but once it is done, we can easily adhere to the best-practice of test driven development (TDD) for our component libraries.

There are alternatives to the manual setup process described above. You can check out some of the links under the AngularJS section on the Karma website for AngularJS project generators:

https://github.com/yeoman/generator-karma[11]

https://github.com/yeoman/generator-angular[12]

There is also the AngularJS Seed GIT repository that you can clone:

https://github.com/angular/angular-seed[13]

The links above will set up a preconfigured AngularJS application directory structure and scaffolding including starter configurations for unit and e2e testing. These can be helpful in understanding all the pieces that are needed for a proper test environment, but they are also focused on what someone else's idea of an AngularJS application directory naming and structure should be. If building a component library, these directory structures may not be appropriate.

# Unit Tests for our Component APIs

Unit testing AngularJS component directives can be a little tricky if our controller logic is inlined with the directive definition object and we have created an isolate scope for the purposes of encapsulation. We need to set up our pre-test code a bit differently than if our controller was registered directly with an application level module. The most important difference to account for is that the scope object created by $rootScope.$new() is not the same scope object with the values and logic from our directive definition. It is the parent scope at the level of the pre-compiled, pre-linked directive element. Attempting to test any exposed functions on this object will result in a lot of "undefined" errors.

**5.9 Full Unit Test Coverage for our Smart Button API**

```
// SmartButtonSpec.js
describe('My SmartButton component directive', function () {
    var $compile, $rootScope, $scope, $element, element;
    // manually initialize our component library module
    beforeEach(module('UIComponents'));
    // make the necessary angular utility methods available
    // to our tests
    beforeEach(inject(function (_$compile_, _$rootScope_) {
```

[11] https://github.com/yeoman/generator-karma
[12] https://github.com/yeoman/generator-angular
[13] https://github.com/angular/angular-seed

```
        $compile = _$compile_;
        $rootScope = _$rootScope_;
        // note that this is actually the PARENT scope of the directive
        $scope = $rootScope.$new();
    }));
    // create some HTML to simulate how a developer might include
    // our smart button component in their page that covers all of
    // the API options
    var tpl = '<smart-button default-text="A Very Smart Button" '
        + 'active-text="Wait for 5 seconds..." '
        + 'debug="showAlert(\'a value on the $rootScope\')"'
        + '>{{bttnText}} Text from transclusion.</smart-button>';

    // manually compile and link our component directive
    function compileDirective(directiveTpl) {
        // use our default template if none provided
        if (!directiveTpl) directiveTpl = tpl;
        inject(function($compile) {
            // manually compile the template and inject the scope in
            $element = $compile(directiveTpl)($scope);
            // manually update all of the bindings
            $scope.$digest();
            // make the html output available to our tests
            element = $element.html();
        });
    }

    // test our component APIs
    describe('A smart button API', function(){
        var scope;
        beforeEach(function(){
            compileDirective();
            // get access to the actual controller instance
            scope = $element.data('$scope').$$childHead;
            spyOn($rootScope, '$broadcast').andCallThrough();
        });

        // API: default Text
        it('should use the value of "default-text" as the displayed bttn text',
        function(){
            expect(element).toContain('A Very Smart Button');
        });
```

```
        // API: activeText
        it('should display the value of "active-text" when clicked',
        function(){
            expect(scope.bttnText).toBe('A Very Smart Button');
            scope.doSomething();
            expect(scope.bttnText).toBe('Wait for 5 seconds...');
        });

        // API: transclusion content
        it('should transclude the content of the element', function(){
            expect(element).toContain('Text from transclusion.');
        });

        // API: debug
        it('should have the injected logic available for execution', function(){
            expect(scope.debug()).toBe('a value on the $rootScope');
        });

        // API: smart-button-click
        it('should emit any events as APIs', function(){
            spyOn(scope, '$emit');
            scope.$emit('smart-button-click');
            expect(scope.$emit).toHaveBeenCalledWith('smart-button-click');
        });

        // API: smart-button-command
        it('should listen and handle any events as APIs', function(){
            $rootScope.$broadcast('smart-button-command',
                scope.$id, {setClass: 'btn-warning'});
            expect(scope.bttnClass).toContain('btn-warning');
        });
    });
});
```

Code blocks in bold above are additions or changes to our unit test code skeleton. Note that the template used, covers all the API options. If this is not possible in a single template, than include as many as are need to approximate consumer usage of your component. Also note the extra step involved in accessing the controller and associated scope of the compiled and linked directive. If a template is used that produces more than one instance of the component, then referencing the scope via "$$childHead" will not be reliable.

Recall earlier in this chapter that software component APIs, on a conceptual level, are a contract

between the developer and consumer. The developer guarantees to the consumer that the component can be configured to behave in a certain way regardless of the internal details. Comprehensive unit test coverage of each API is essential to back up the developer's end of the agreement when the internals change via bug fixes and improvements. This is especially critical if the component library is commercial and expensive for the consumer, and where breakage due to API changes can result in legal action.

The above unit tests will server as the first line of defense in insuring that consistent API behavior is maintained during code changes. Only after a standard, published period of "deprecation" for any API should unit test coverage for it be removed.

Another beneficial side-effect of unit testing can be gained when considering the amount of time it takes to produce a unit test for a unit of functionality. If, on average, creating the test takes significantly longer than normal, then that can be a clue to questions the quality of the approach taken to provide the functionality. Are hard-coded or global dependencies being referenced? Are individual functions executing too much logic indicating they should be split up?

# Summary

In this chapter, we've hopefully covered all the necessary aspects of developing a re-usable and portable UI component directive that is as *high quality* as possible given the current limitations of today's major browsers. The "quality" referred to includes maximum encapsulation, documentation, test coverage, and API that facilitates easy consumption. Later in this book we will compare what we have built here to a version of the same component built using the looming W3C Web Components standards that will eventually obsolete the need for JavaScript frameworks in favor of native DOM and JavaScript methods for component Development.

In the next chapter, we will move up a level to what can be conceptually thought of as "UI container components". These are DOM container component elements (tabs containers, accordions, menu bars, etc.) whose primary purpose is to be filled with and manage a set of UI components.

# Chapter 8 - W3C Web Components Tour

Beginning in 2012 there has been serious chatter on all the usual developer channels about something called Web Components. The term might already sound familiar to you from both Microsoft and Sun Microsystems. Microsoft used the term to describe add-ons to Office. Sun (now Oracle) used the term to describe Java, J2EE servlets. Now the term is being used to refer to a set of W3C draft proposals for browser standards that will allow developers to accomplish natively in the browser what we do today with JavaScript UI frameworks.

## The Roadmap to Web Components

As the AJAX revolution caught on, and data was being sent to browsers asynchronously, technics for displaying the data using the tools browsers provided (JavaScript, CSS, and HTML) needed development. Thus, client-side UI widget frameworks arose including DOJO, jQuery UI, ExtJS, and AngularJS. Some of the early innovations by the DOJO team really stood out. They developed modules to abstract AJAX and other asynchronous operations (promises and deferreds. On top of that, they developed an OO based UI widget framework that allowed widget instantiation via declarative markup. Using DOJO you could build full client-side applications out of modules they provided in a fraction of the time it took to write all of the low level code by hand. Furthermore, using DOJO provided an abstraction layer for different browser incompatibilities.

While DOJO provided a "declarative" option for their framework, other framework developers took different approaches. jQuery UI, built on top of jQuery, was imperative. GWT (Google Windowing Toolkit) generated the JavaScript from server-side code, and ExtJS was configuration based. AngularJS largely built upon the declarative approach that DOJO pioneered.

A lot of brilliant thinking and hard work has gone into client-side frameworks over the last decade. As we mentioned earlier, there are literally thousands of known client-side UI frameworks out there, and huge developer communities have sprung up around those in the top ten.

Sadly though, at the end of the day (or decade) all of this brilliance and work adds up to little more than hacks around the limitations of the browser platforms themselves. If the browser vendors had built in ways to encapsulate, package, and re-use code in HTML and DOM APIs at the same time as AJAX capability, modern web development would be a very different beast today. Be as it may, browser standards progress in a "reactive" manner. Semantic HTML5 tags like `<header>` and `<nav>` only became standards after years of `<div class="header">` and `<div class="nav">`.

Now that we've had workarounds in the form of UI frameworks for the better part of the last decade that handle tasks like templating, encapsulation, and data-binding, the standards committees are

now discussing adding these features to ECMA Script and the DOM APIs. While still a couple years away from landing in Internet Explorer, we are starting to get prototype versions of these features in Chrome and Firefox that we can play with now.

The set of specifications falling under the "Web Components" umbrella include:

- The ability to create **custom elements** with the option of extending existing elements
- A new `<template>` tag to contain inert, reusable chunks of HTML, JavaScript, and CSS for lazy DOM insertion
- A way to embed a DOM tree within a DOM tree called **shadow DOM** that provides encapsulation like an iframe without all the overhead
- The ability (called **HTML imports**) to load HTML documents that contain the items above in an HTML document

The above proposed standards are under the domain of the W3C. A closely related standard under the domain of the ECMA are additions to the Object API that allow property mutations to be tracked natively that folks are referring to as "**Object.observe()**". `Object.observe()` is proposed for ES7 which is a ways off. ES6, which is already starting to be implemented in Firefox and Node.js, will add long-awaited conveniences like modules and classes for easier code reuse in Web Components.

Certain forward looking framework authors, including the AngularJS team, are well aware of the proposed standards and are now incorporating them into their product roadmaps.

## Disclaimers and Other CYA Stuff

While we will be covering some detailed examples of Web Component standards implementation, the intent is to present what's to come at a high level or abstracted somewhat from the final APIs. The actual standards proposals are still in a very high state of flux. The original proposals for `<element>` and `<decorator>` tags have been shelved due to unforeseen implementation issues. Other proposals are still experiencing disagreement over API details. Therefore, any and all examples in this section should be considered as pseudo-code for the purposes of illustrating the main concepts, and there is no guarantee of accuracy.

The W3C maintains a status page for the Web Components drafts at:

http://www.w3.org/standards/techs/components#w3c_all[14]

This page has links to the current status of all the proposals.

This presentation is also intended to be unbiased from an organization standpoint. The Web Components proposals have been largely created, supported and hyped by members of Google's Chrome development team with positive support from Mozilla. Apple and Microsoft, on the other hand, have been largely silent about any roadmap of support for these standards in their browsers.

[14] http://www.w3.org/standards/techs/components#w3c_all

The top priority for the latter two companies is engineering their browsers to leverage their operating systems and products, so they will be slower to implement. Browser technology innovations are more central to Google's and Mozilla's business strategy.

At the time of this writing, your author has no formal affiliation with any of these organizations. The viewpoint presented is that of a UI architect and engineer who would be eventually implementing and working with these standards in a production setting. Part of this viewpoint is highly favorable towards adopting these standards as soon as possible, but that is balanced by the reality that it could still be years before full industry support is achieved. Any hype from Google that you can start basing application architecture on these standards and associated polyfills in the next six months or year should be taken with a grain of salt.

Just about everyone is familiar with the SEC disclaimer about "forward looking statements" that all public companies must include with any financial press releases. That statement can be safely applied to the rest of the chapter in this book.

## The Stuff They Call "Web Components"

In the following sections we will take a look at the specifics of W3C proposal technologies. This will include overviews of the API methods current at the time of writing plus links to the W3C pages where the drafts proposals in their most current state can be found. If you choose to consult these documents, you should be warned that much of their content is intended for the engineers and architects who build browsers. For everyone else, it's bedtime reading, and you may fall asleep before you can extract the information relevant to web developers.

The example code to follow should, for the most part, be executable in Chrome Canary with possible minor modification. It can be downloaded at the following URL.

http://www.google.com/intl/en/chrome/browser/canary.html[15]

Canary refers to what happens to them in coal mines. It is a version of Chrome that runs a few versions ahead of the official released version and contains all kinds of features in development or experimentation including web component features. Canary can be run along side official Chrome without affecting it. It's great for getting previews of new feature, but should never, ever be used for regular web browsing as it likely has a lot of security features turned off. Then again, if you have a co-worker you dislike greatly, browsing the net with it from their computer is probably ok.

Update 9/2014, all of the Web Components example code now works in production Chrome, so you no longer need to use Canary. You may still with to use Canary for some of the new JavaScript ES6/7 features.

[15] http://www.google.com/intl/en/chrome/browser/canary.html

# Custom Elements

Web developers have been inserting tags with non-standard names into HTML documents for years. If the name of the tag is not included in the document's DOCTYPE specification, the standard browser behavior is to ignore it when it comes to rendering the page. When the markup is parsed into DOM, the tag will be typed as HTMLUnknownElement. This is what happens under the covers with AngularJS element directives such as `<uic-menu-item>` that we used as example markup earlier.

The current proposal for custom element capability includes the ability to define these elements as first class citizens of the DOM.

http://www.w3.org/TR/custom-elements/[16]

Such elements would inherit properties from HTMLElement. Furthermore, the specification includes the ability to create elements that inherit from and extend existing elements. A web developer who creates a custom element definition can also define the DOM interface and special properties for the element as part of the custom element's prototype object. The ability to create custom elements combined with the additions of classes and modules in ES6 will become a very powerful tool for component developers and framework authors.

## Registering a Custom Element

As of the time of this writing, registering custom elements is done imperatively via JavaScript. A proposal to allow declarative registration via an `<element>` tag has been shelved due to timing issues with document parsing into DOM. This is not necessarily a bad thing since an `<element>` tag would encourage "hobbyists" to create pages full of junk. By requiring some knowledge of JavaScript and the DOM in order to define a new element, debugging someone else's custom element won't be as bad.

There are three simple steps to registering a custom element.

1. Create what will be its prototype from an existing HTMLElement object
2. Add any custom markup, style, and behavior via it's "created" lifecycle callback
3. Register the element name, tag name, and prototype with the document

A tag that matches the custom element may be added to the document either before or after registration. Tags that are added before registration are considered "unresolved" by the document, and then automatically upgraded after registration. Unresolved elements can be matched with the :unresolved CSS pseudo class as a way to hide them from the dreaded FOUC (flash of unstyled content) until the custom element declaration and instantiation.

[16] http://www.w3.org/TR/custom-elements/

**8.0 Creating a Custom Element in its Simplest Form**

```
<script>
    // Instantiate what will be the custom element prototype
    // object from an element prototype object
    var MenuItemProto = Object.create( HTMLElement.prototype );

    // browsers that do not support .registerElement will
    // throw an error
    try{
        // call registerElement with a name and options object
        // the name MUST have a dash in it
        var MenuItem = document.registerElement( 'menu-item', {
            prototype: MenuItemProto
        });
    }catch(err){}
</script>

<!-- add matching markup -->
<menu-item>1</menu-item>
<menu-item>2</menu-item>
<menu-item>3</menu-item>
```

**Listing 8.0** illustrates the minimal requirements for creating a custom element. Such an element will not be useful until we add more to the definition, but we can now run the following in the JavaScript console which will return true:

```
document.createElement('menu-item').__proto__ === HTMLElement
```

versus this for no definition or .registerElement support:

```
document.createElement('menuitem').__proto__ === HTMLUnknownElement
```

**8.1 Extending an Existing Element in its Simplest Form**

```
<script>

    // Instantiate what will be the custom element prototype
    // object from an LI element prototype object
    var MenuItemProto = Object.create( HTMLLIElement.prototype );

    // browsers that do not support .registerElement will
    // throw an error
    try{
```

```
        // call registerElement with a name and options object
        // the name MUST have a dash in it
        var MenuItem = document.registerElement( 'menu-item', {
            prototype: MenuItemProto,
            extends: 'li'
        });
    }catch(err){}
</script>

<!-- add matching markup -->
<li is="menu-item">1</li>
<li is="menu-item">2</li>
<li is="menu-item">3</li>
```

**Listing 8.1** illustrates the minimal steps necessary to extend an existing element with differences in bold. The difference here is that these elements will render to include any default styling for an `<li>` element, typically a bullet. In this form parent element name "li" must be used along with the "is="element-type" attribute.

It is still possible to extend with HTMLLIElement to get access to its methods and properties without the extends: 'li' in order to use the `<menu-item>` tag, but it will not render as an `<li>` tag by default. Elements defined by the method in the latter example are called "**type extention elements**". Elements created similar to the former example are called "**custom tags**". In both cases it must be noted that custom tag names *must* have a dash in them to be recognized as valid names by the browser parser. The reasons for this are to distinguish custom from standard element names, and to encourage the use of a "namespace" name such as "uic" as the first part of the name to distinguish between source libraries.

If this terminology remains intact until widespread browser implementation, it will likely be considered a best-practice or rule-of-thumb to use type extensions for use-cases where only minor alterations to an existing element are needed. Use-cases calling for heavy modification of an existing element would be better served descriptively and declaratively as a custom tag. Most modern CSS frameworks, such as Bootstrap, select elements by both tag name and class name making it simple to apply the necessary styling to custom tag names.

## Element Lifecycle Callbacks and Custom Properties

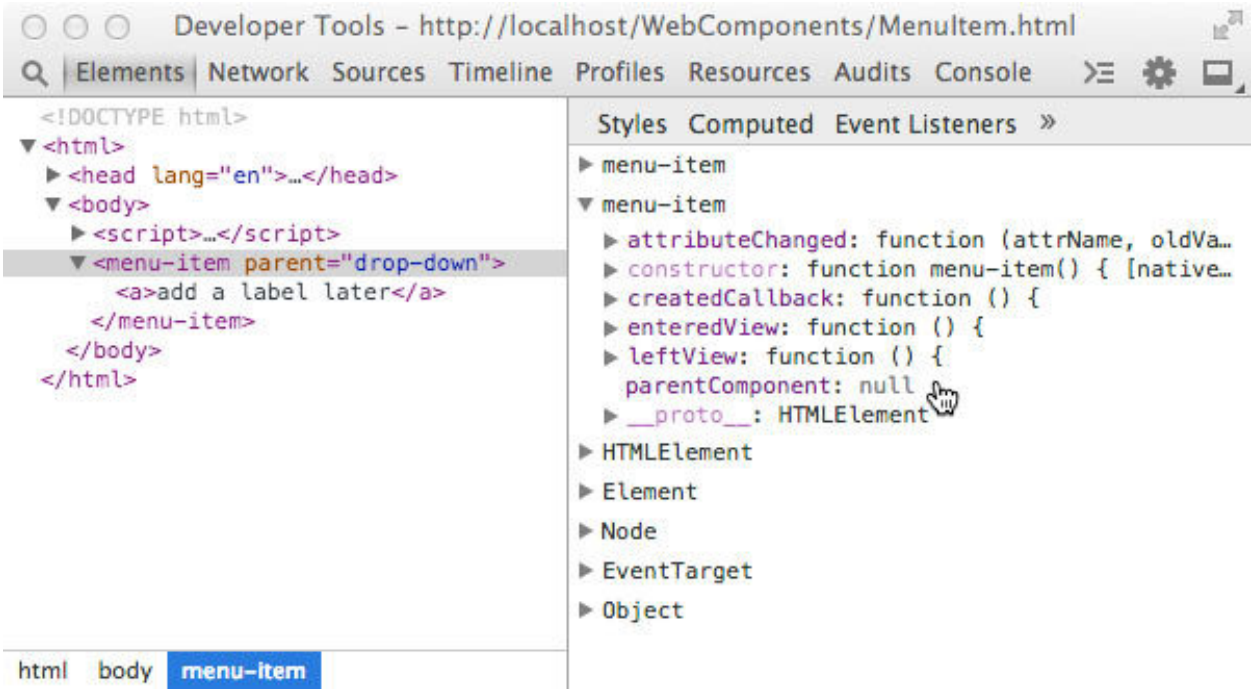
Until now the custom element extension and definition examples are not providing any value-add. The whole point of defining custom elements, and ultimately custom components, is to add the properties and methods that satisfy the use-case. This is done by attaching methods and properties to the prototype object we create, and by adding logic to a set of predefined lifecycle methods.

**8.2 Custom Element Lifecycle Callbacks**

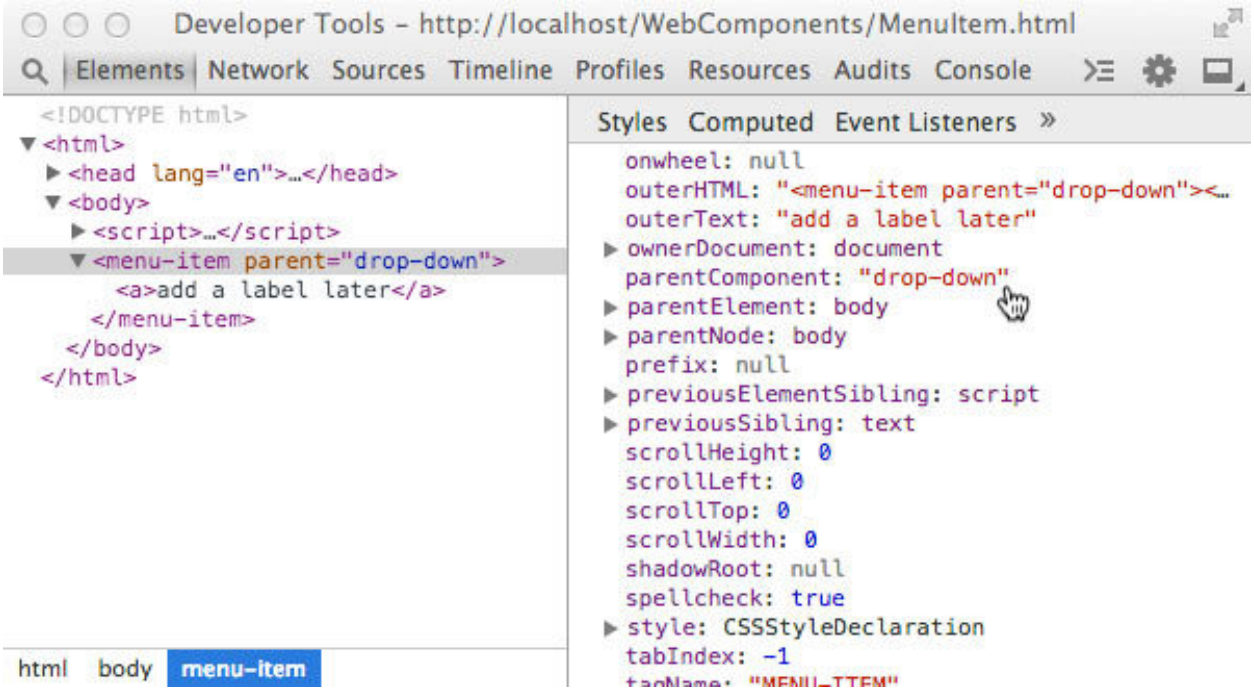
```
<script>
    // Instantiate what will be the custom element prototype
    // object from an element prototype object
    var MenuItemProto = Object.create(HTMLElement.prototype);
    // add a parent component property
    MenuItemProto.parentComponent = null;
    // lifecycle callback API methods
    MenuItemProto.created = function() {
        // perform logic upon registration
        this.parentComponent = this.getAttribute('parent');
        // add some content
        this.innerHTML = "<a>add a label later</a>";
    };
    MenuItemProto.enteredView = function() {
        // do something when element is inserted
        // in the DOM
    };
    MenuItemProto.leftView = function() {
        // do something when element is detached
        // from the DOM
    };
    MenuItemProto.attributeChanged = function(attrName, oldVal, newVal){
        // do something if an attribute is
        // modified on the element
    };
    // browsers that do not support .registerElement will
    // throw an error
    try{
        // call registerElement with a name and options object
        var MenuItem = document.registerElement('menu-item', {
            prototype: MenuItemProto
        });
    }catch(err){}
</script>

<menu-item parent="drop-down"></menu-item>
```

The proposed custom element specifications include a set of lifecycle callback method names. The actual names are in a state of flux. The ones in bold in the example are current as of mid 2014. Whatever the final names are, there will most likely be a callback upon element registration where

most logic should execute. Upon execution of this callback, :unresolved would no longer apply as a selector. The other callbacks will likely run upon DOM insertion, detachment from the DOM, and upon any attribute mutation. The parameters for attribute mutation events will likely be name plus old, and new values for use in any necessary callback logic.




**Inspection of the MenuItem prototyope object "parentComponent" property**




**Inspection of the `<menu-item>` "parentComponent" instance property**

In the above example we are adding a parentComponent property to the element. The purpose would be analogous to the parentElement property except that a UI component hierarchy would likely have DOM hierarchies within each UI component. So "walking" the component hierarchy would not be the same as walking the DOM hierarchy. Along these lines, the example has us getting the parentComponent reference name from a "parent" attribute. It also has a simple example of directly adding some innerHTML. In actual practice, we would likely add any element content as **shadow DOM** via `<template>` tags as we will illustrate in the next sub-sections.

To recap on custom element definitions, they consist of the **custom element type**, the **local name** (tag name), the **custom prototype**, and **lifecycle callbacks**. They also may have an optional XML namespace prefix. Once defined, custom elements must be registered before their properties can be applied. Custom elements, while useful in their own right, will be greatly enhanced for use as UI components when they are used in combination with shadow DOM and template tags. Also, as mentioned earlier, plans for declarative definitions via an `<element>` tag have been shelved, but the Polymer framework has a custom version of the tag called `<polymer-element>` that can be used in a declarative fashion. We will explore Polymer and other polyfill frameworks in the next chapter.

# Shadow DOM

I like to describe Shadow DOM as kind of like an iframe, but with out the "frame".

Up to now the only way to embed third party widgets in your page with true encapsulation is inside a very "expensive" iframe. What I mean by expensive, is that an iframe consumes the browser

resources necessary to instantiate an entirely separate window and document. Even including an entirely empty `<iframe>` element in a page is orders of magnitude more resource intensive than any other HTML element. Examples of iframe widgets can be social networking buttons like a Google+ share, a Twitter tweet, a Facebook like, or a Disqus comments block. They can also include non-UI marketing and analytics related apps.

Often times an iframe makes sense from a security standpoint if the content is unknown or untrusted. In those situations, you want to be able to apply CSP (content security policy) restrictions to the embedded frame. But iframes are also the only choice if you need to encapsulate your widget's look and feel from the styling in the parent page even if there are no trust issues. CSS bleed is the largest obstacle when it comes to developing portable UI components meant to propagate a particular set of branding styles. Even the HTML in such components can be inadvertently altered or removed by code in the containing page. Only JavaScript, which can be isolated in an anonymous self-executing function is relatively immune from outside forces.

http://www.w3.org/TR/shadow-dom/[17]

http://w3c.github.io/webcomponents/spec/shadow[18]

Shadow DOM is a W3C specification proposal that aims to fill the encapsulation gap in modern browsers. In fact, all major browsers already use it underneath some of the more complex elements like `<select>` or `<input type="date">`. These tags have their own child DOM trees of which they are rendered from. The only thing missing is an API giving the common web developer access to the same encapsulation tools.

## Like an extremely light-weight iframe

Shadow DOM provides encapsulation for the embedded HTML and CSS of a UI component. Logically the DOM tree in a Shadow DOM is seperate from the DOM tree of the page, so it is not possible to traverse into, select, and alter nodes from the parent DOM just like with the DOM of an embedded iframe. If there is an element with id=”comp_id” in the shadow DOM, you cannot use jQuery, `$('#comp_id')`, to get a reference. Likewise, CSS styles with that selector will not cross the shadow DOM boundary to match the element. The same holds true for certain events. In this regard, Shadow DOM is analogous to a really “light-weight” iframe.

That said, it is possible to traverse into the shadow DOM indirectly with a special reference to what is called the **shadow root**, and it is possible to select elements for CSS styling with a special pseudo class provided by the spec. But both of these tasks take conscious effort on the part of the page author, so inadvertent clobbering is highly unlikely.

JavaScript is the one-third of the holy browser trinity that behaves no different. Any `<script>` tags embedded in a shadow tree will execute in the same window context as the rest of the page. This is why shadow DOM would not be appropriate for untrusted content. However, given that JavaScript can already be encapsulated via function blocks in ES5 and via modules and classes in ES6,

[17] http://www.w3.org/TR/shadow-dom/

[18] http://w3c.github.io/webcomponents/spec/shadow

Shadow DOM provides the missing tools for creating high quality UI components for the browser environment.

## Shadow DOM Concepts and Definitions

We have touched on the high-level Shadow DOM concepts above, but to really get our heads wrapped around the entire specification to the point where we can start experimenting, we need to introduce some more concepts and definitions. Each of these has a coresponding API hook that allows us to create and use shadow DOM to encapsulate our UI components. Please take a deep breath before continuing, and if you feel an anurysim coming on, then skip ahead to the APIs and examples!

### Tree of Trees

The Document Object Model as we know it, is a tree of nodes (leaves). Adding **shadow DOM** means that we can now create a **tree of trees**. One tree's nodes do not belong to another tree, and there is no limit to the depth. Trees can contain any number of trees. Where things get murky is when the browser renders this tree of trees. If you have been a good student of graph theory, then you'll have a leg up groking these concepts.

## Page Document Tree and Composed DOM

The final visual output to the user is that of a single tree which is called the **composed DOM**. However, what you see in the browser window will not seem to jibe with what is displayed in your developer tools, which is called the **page document tree**, or the tree of nodes above. This is because prior to rendering, the browser takes the chunk of **shadow DOM** and attaches it to a specified **shadow root**.

## Shadow Root and Shadow Host

A **shadow root** is a special node inside any element which "hosts" a **shadow DOM**. Such an element is referred to as a **shadow host**. The **shadow root** is the root node of the **shadow DOM**. Any content within a **shadow host** element that falls outside of the hosted **shadow DOM** will not be rendered directly as part of the **composed DOM**, but may be rendered if transposed to a **content insertion point** described as follows.

## Content and Shadow Insertion Points

A **shadow host** element may have regular DOM content, as well as, one or more **shadow root(s)**. The **shadow DOM** that is attached to the **shadow root** may contain the special tags `<content>` and `<shadow>`. Any regular element content will be transposed into the `<content>` element body

wherever it may be located in the **shadow DOM**. This is refered to as a **content insertion point**. This is quite analogous to using AngularJS ngTransclude except that rather than retaining the original $scope context, transposed **shadow host** content retains its original CSS styling. A shadow DOM is not limited to one **content insertion point**. A shadow DOM may have several `<content>` elements with a special `select="css-selector"` attribute that can match descendent nodes of the **shadow host** element, causing each node to be transposed to the corresponding **content insertion point**. Such nodes can be refered to as **distributed nodes**.

Not only may a **shadow host** element have multiple **content insertion point**s, it may also have multiple **shadow insertion point**s. Multiple **shadow insertion point**s end up being ordered from oldest to youngest. The youngest one wins as the "official" **shadow root** of the **shadow host** element. The "older" **shadow root**(s) will only be included for rendering in the **composed DOM** if **shadow insertion point**s are specifically declared with the `<shadow>` tag inside of the youngest or official **shadow root**.

## Shadow DOM APIs

The following is current as of mid 2014. For the latest API information related to Shadow DOM please check the W3C link at the top of this section. The information below also assumes familiarity with the current DOM API and objects including: Document, DocumentFragment, Node, NodeList, DOMString, and Element.

From a practical standpoint we will start with the extensions to the current Element interface.

`Element.createShadowRoot()` - This method takes no parameters and returns a ShadowRoot object instance. This is the method you will likely use the most when working with ShadowDom given that there currently is no declarative way to instantiate this type.

`Element.getDestinationInsertionPoints()` - This method also takes no parameters and returns a *static* NodeList. The NodeList consists of insertion points in the destination insertion points of the context object. Given that the list is static, I suspect this method would primarily be used for debugging and inspection purposes.

`Element.shadowRoot` - This attribute either referes to the *youngest* ShadowRoot object (read only) if the node happens to be a shadow host. If not, it is *null.*

`ShadowRoot` - This is the object type of any shadow root which is returned by 'Element.createShadowRoot(). A ShadowRoot object inherits from DocumentFragment.

The ShadowRoot type has a list of methods and attributes, several of which are not new, and are the usual DOM traversal and selection methods (getElementsBy*). We will cover the new ones and any that are of particular value when working shadow dom.

`ShadowRoot.host` - This attribute is a reference to its shadow host node.

`ShadowRoot.olderShadowRoot` - This attribute is a reference to any previously defined ShadowRoot on the shadow host node or *null* if none exists.

`HTMLContentElement <content>` - This node type and tag inherits from HTMLElement and represents a content insertion point. If the tag doesn't satisfy the conditions of a content insertion point it falls back to a HTMLUnknownElement for rendering purposes.

`HTMLContentElement.getDistributedNodes()` - This method takes no arguments and either returns a static node list including anything transposed from the shadow host element, or an empty node list. This would likely be used primarily for inspection and debugging.

`HTMLShadowElement <shadow>` - Is exactly the same interface as HTMLContentElement except that it describes shadow insertion point instead.

## Events and Shadow DOM

When you consider that shadow DOMs are distinct node trees, the standard browser event model will require some modification to handle user and browser event targets that happen to originate inside of a shadow DOM. In some cases depending on event type, an event will not cross the shadow boundary. In other cases, the event will be "retargeted" to appear to come from the shadow host element. The event "retargeting" algorithm described in specification proposal is quite detailed and complex considering that an event could originate inside several levels of shadow DOM. Given the complexity of the algorithm and the fluidity of the draft proposal, it doesn't make sense to try to distill it here. For now, we shall just list the events (mostly "window" events) that are halted at the shadow root.

- abort
- error
- select
- change
- load
- reset
- resize
- scroll
- selectstart

## CSS and Shadow DOM Encapsulation

As mentioned earlier, provisions are being made to include CSS selectors that will work with shadow DOM encapsulation. By default, CSS styling does not cross a shadow DOM boundary in either direction. There are W3C proposals on the table for selectors to cross the shadow boundary in controlled fashion designed to prevent inadvertent CSS bleeding. A couple terms, **light DOM** and **dark DOM** are used to help differentiate the actions these selectors allow. Light DOM refers to any DOM areas not under shadow roots, and dark DOM are any areas under shadow roots. These

specs are actually in a section (3) of a different W3C draft proposal concerning CSS scoping within documents.

http://www.w3.org/TR/css-scoping-1/#shadow-dom[19]

The specification is even less mature and developed than the others related to Web Components, so everything discussed is highly subject to change. At a high level CSS scoping refers to the ability apply styling to a sub-tree within a document using some sort of scoping selector or rule that has a very high level of specificity similar to the "@" rules for responsive things like device width.

What is currently being discussed are pseudo elements matching `<content>` and `<shadow>` tags, and a pseudo class that will match the shadow host element from within the shadow DOM tree. Also being discussed is a "combinator" that will allow document or shadow host selectors to cross all levels of shadow DOM to match elements. The current names are:

**`:host or :host( [selector] )`** - a pseudo class that would match the shadow host element always or conditionally from a CSS rule located within the shadow DOM. This pseudo class allows rules to cross from the dark to the light DOM. The use case is when a UI component needs to provide its host node with styling such as for a theme or in response to user interaction like mouse-overs.

**`:host-context( [selector] )`** - a pseudo class that may match a node anywhere in the shadow host's context. For example, if the CSS is located in the shadow DOM of a UI component one level down in the document, this CSS could reach any node in the document.

Hopefully this one *never* sees the light of day in browser CSS standards! It is akin to JavaScript that pollutes the global context, and it violates the spirit of component encapsulation. The use case being posited that page authors may choose to "opt-in" to styles provided by the component is extremely weak. What is highly likely to happen is the same "path of least resistance" rushed developement where off-shore developers mis-use the pseudo class to quickly produce the required visual effect. Debugging and removing this crap later would be nothing short of nightmarish for the developer who ultimately must maintain the code.

**`::shadow`** - a pseudo element that allows CSS rules to cross from the light to the dark DOM. The ::shadow pseudo element could be used from the shadow host context to match the actual shadow root. This would allow a page author to style or override any styling in the shadow DOM of a component. A typical use-case would be to style a particular component by adding a shadow root selector:

```
.menu-item::shadow > a { color: green }
```

This presumably would override the link color of any MenuItem component.

**`::content`** - a pseudo element that presumably matches the parent element of any distributed nodes in a shadow tree. Recall that earlier we mentioned that any non-shadow content of a shadow host element that is transposed into the shadow tree with `<content>` retains its host-context styling. This would allow any CSS declared within the shadow DOM to match and style the transposed content.

[19] http://www.w3.org/TR/css-scoping-1/#shadow-dom

An analogous AngularJS situation would be something like the uicInclude directive we created in chapter 6 to replace parent scope with component scope for transcluded content.

The preceding pseudo classes and elements can only be used to cross a single shadow boundary such as from page context to component context or from container component context to "contained" component context. An example use case could be to theme any dropdown menu components inside of a menu-bar container component with CSS that belongs to the menu-bar. However, what if the menu-bar component also needed to provide theming for the menu-item components that are children of the dropdown components. None of the above selectors would work for that. So there is another selector (or more precisely a combinator) that takes care of situation where shadow DOMs are nested multiple levels.

**`/deep/`** - a combinator (similar to "+" or ">") that allows a rule to cross down all shadow boundaries no matter how deeply nested they are. We said above that ::shadow can only cross one level of shadow boundary, but you can still chain multiple `::shadow` pseudo element selectors to cross multiple nested shadow boundaries.

```
menu-bar::shadow dropdown::shadow > a { ...
```

This is what you would do if you needed to select an element that is exactly two levels down such as from menu bar to menu item. If on the other hand, you needed to make sure all anchor links are green in every component on a page, then you can shorten your selector to:

```
/deep/ a { ... }
```

Once again, the proposal for these specs is in high flux, and you can expect names like ::content to be replaced with something less general by the time this CSS capability reaches a browser near you. The important takeaways at this point are the concepts not the names.

We will be covering some of the Web Components polyfills in the next chapter, but it is worth mentioning now that shadow DOM style encapsulation is one area where the polyfill shivs fall short. In browsers that do not yet support shadow DOM, CSS will bleed right through in both directions with the polyfills. Additionally, emulating the pseudo classes and elements whose names are still up in the air is expensive. The CSS must be parsed *as text* and the pseudo class replaced with a CSS emulation that does work.

There also appears to be varying levels of shadow DOM support entering production browser versions. For instance, as of mid-2014, Chrome 35 has shadow DOM API support, but not style encapsulation, whereas, Chrome Canary (37) does have style encapsulation. Opera has support. Mozilla is under development, and good old Microsoft has shadow DOM "under consideration".

**8.3 Adding some Shadow DOM to the Custom Element**

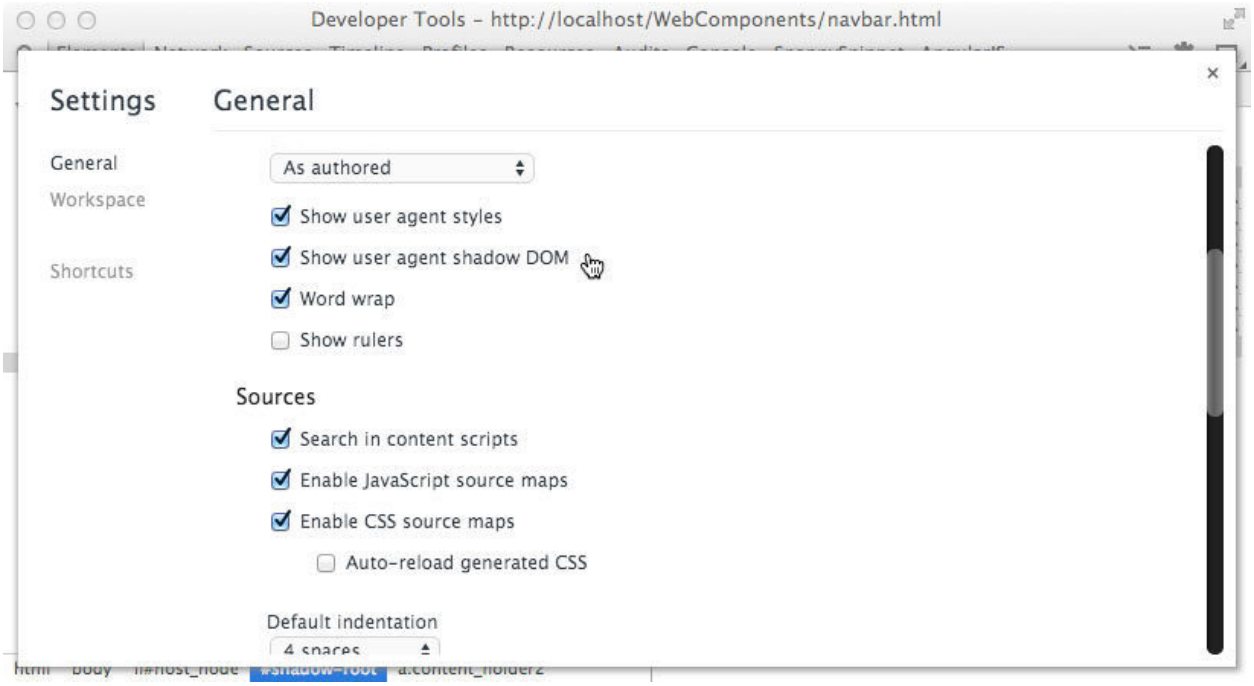
```
<script>
    // Instantiate what will be the custom element prototype
    // object from an element prototype object
    var MenuItemProto = Object.create(HTMLElement.prototype);
    // add a parent component property
    MenuItemProto.parentComponent = null;
    // lifecycle callback API methods
    MenuItemProto.created = function() {
        // perform logic upon registration
        this.parentComponent = this.getAttribute('parent');
        // add some content
        // this.innerHTML = "<a>add a label later</a>";
        // instantiate a shadow root
        var shadowRoot = this.createShadowRoot();
        // move the <a> element into the "dark"
        shadowRoot.innerHTML = "<a>add a label later</a>";
        // attach the shadow DOM
        this.appendChild( shadowRoot );
    };
    // the remaining custom element code is truncated for now
</script>

<menu-item parent="drop-down"></menu-item>
```

This is a minimalist example of creating, populating, and attaching shadow DOM to a custom element. The created or createdCallback (depending on final name) function is called when the instantiation process of a custom element is complete which makes this moment in the lifecycle ideal for instantiating and attaching any shadow roots needed for the component.

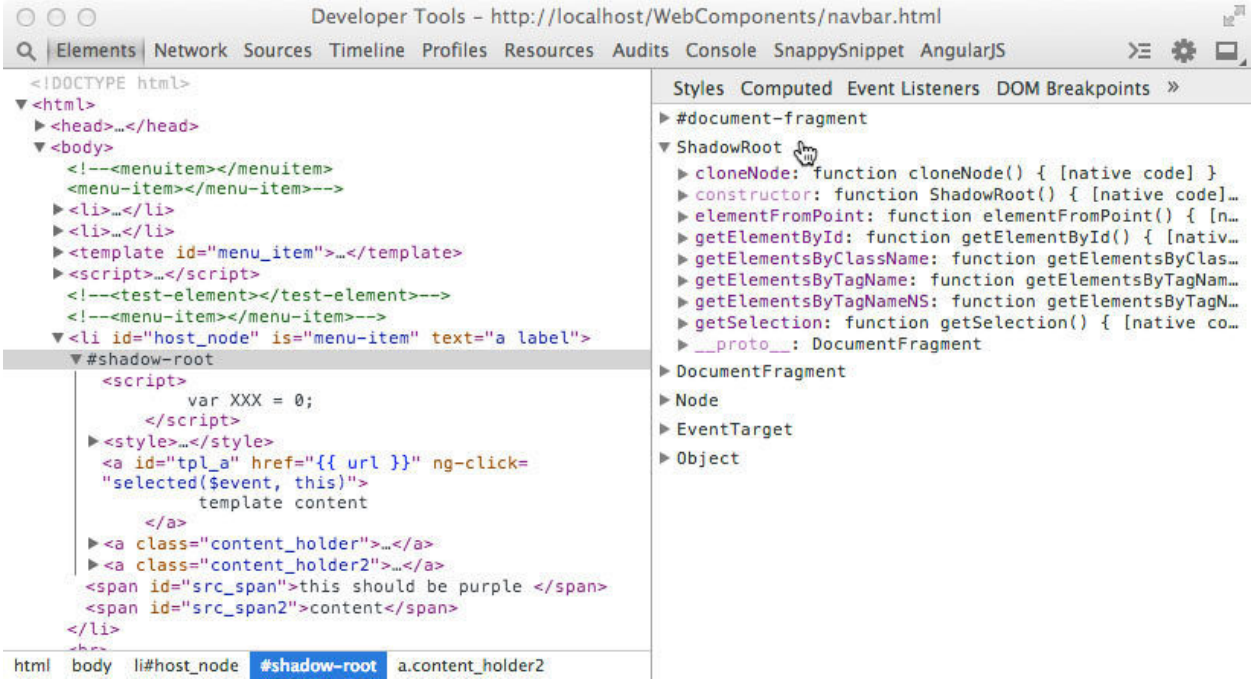
Rather than getting into a bunch of programmatic DOM creation for the sake of more indepth shadow DOM API examples, we will defer until the next section where we cover the `<template>` tag. `<template>` tags are the ideal location to stick all the HTML, CSS, and JavaScript that we would want to encapsulate as shadow DOM in our web component definition.

## Shadow DOM Inspection and Debugging

As of Chrome 35, the developer tools settings have a switch to turn on shadow DOM inspection under General -> Elements. This will allow you to descend into any shadow root elements, but the view is the *logical* DOM, not the composed (rendered) DOM.




**If you check this box, you can inspect shadow DOM**




**A view of the logical DOM (left) and a shadow DOM object (right)**

What you see in dev tools likely won't match what you see in the browser window. It's a pretty good bet that by the time Web Components are used in production web development, the Chrome team will have added a way to inspect the composed DOM into dev tools. For the time being, Eric

Bidelman of the Polymer team at Google has a shadow DOM visualizer tool, based on d3.js, that can be accessed at:

http://html5-demos.appspot.com/static/shadowdom-visualizer/index.html[20]

It's helpful for resolving content insertion points in the logical DOM for a single shadow host element, so you can get a feel for the basic behavior for learning purposes

# Using the <template> Tag

The <template> tag is already an established W3C standard. It is meant to replace <script> tag overloading for holding inert DOM fragments meant for reuse. The only reason it has not already replaced <script> tag overloading in JavaScript frameworks is because, surprise, surprise, Internet Explorer has not implemented it. As of mid-2014, its status for inclusion in Internet Explorer is "under consideration". This means we'll see it in IE 13 at the earliest. So that is the reality check. We will pretend that IE does not exist for the remainder of this section.

http://www.w3.org/TR/html5/scripting-1.html#the-template-element[21]

## Advantages and Disadvantages

Today, if we want to pre-cache an AngularJS template for reuse, we overload a script tag with the contents as the template with the initial page load:

```
<script type="text/ng-template">
    <!-- HTML structure and AngularJS expressions as STRING content -->
</script>
```

The advantage of this is that the contents are inert. If the type attribute value is unknown, nothing is parsed or executed. Images don't download; scripts don't run; nothing renders, and so on. However, disadvantages include poor performance since a string must be parsed into an innerHTML object, XSS vulnerability, and hackishness since this is not the intended use of a <script> tag. In AngularJS 1.2 string sanitizing is turned on by default to prevent junior developers from inadvertently allowing an XSS attack vector.

The contents of <template> tags are not strings. They are actual document fragment objects parsed natively by the browser and ready for use on load. However, the contents are just as inert as overloaded script tags until activated for use by the page author or component developer. The "inertness" is due to the fact that the ownerDocument property is a transiently created document object just for the template, not the document object that is rendered in the window. Similar to shadow DOM, you cannot use DOM traversal methods to descend into <template> contents.

[20] http://html5-demos.appspot.com/static/shadowdom-visualizer/index.html
[21] http://www.w3.org/TR/html5/scripting-1.html#the-template-element

The primary advantages of <template> tags are appropriate semantics, better declarativeness, less XSS issues, and better initial performance than parsing <script> tag content. The downside of using <template> tags include the obvious side effects in non-supported browsers, and lack of any template language features. Any data-binding, expression or token delimiters are still up to the developer or JavaScript framework. The same goes for the usual logic including loops and conditionals. For example, the core Polymer module supplies these capabilities on top of the <template> tag. There is also no pre-loading or pre-caching of assets within <template> tags. While template content may live in <template> tags instead of <script> tags for supported browsers, client-side templating languages are not going away anytime soon. Unfortunately for non-supporting browsers, there is no way to create a full shiv or polyfill. The advantages are supplied by the native browser code, and cannot be replicated by JavaScript before the non-supporting browser parses and executes any content.

What will likely happen in the near future is that JavaScript frameworks will include using <template> tags as an *option* in addition to templating the traditional way. It will be up to the developer to understand when using each option is appropriate.

**8.4 Activating <template> Content**

```
<template id="tpl_tag">
    <script>
        // though inline, does not execute until DOM attached
        var XXX = 0;
    </script>

    <style>
         /* does not style anything until DOM attached
        .content_holder{
            color: purple;
        }
        .content_holder2{
            color: orange;
        }
    </style>
    <!-- does not attempt to load this image until DOM attached -->
    <img src="http://imageserver.com/some_image.jpg"/>
    <a class="content_holder">
        <content select="#src_span"></content>
    </a>
    <a class="content_holder2">
        <content select="#src_span2"></content>
    </a>
</template>

<script>
```

```
    // create a reference to the template
    var template = document.querySelector( '#tpl_tag' );

    // call importNode() on the .content property; true == deep
    var clone = document.importNode( template.content, true );

    // activate the content
    // images download, styling happens, scripts run
    document.appendChild( clone );
</script>
```

As you can see, activating template content is simple. The main concept to understand is that you are importing and cloning an existing node rather than assigning a string to innerHTML. The second parameter to .importNode() must be true to get the entire tree rather than just the root node of the template content.

<template> tags work great in combination with custom elements and shadow DOM for providing basic encapsulation tools for UI (web) components.

#### 8.5 <template> Tag + Shadow DOM + Custom Element = Web Component!

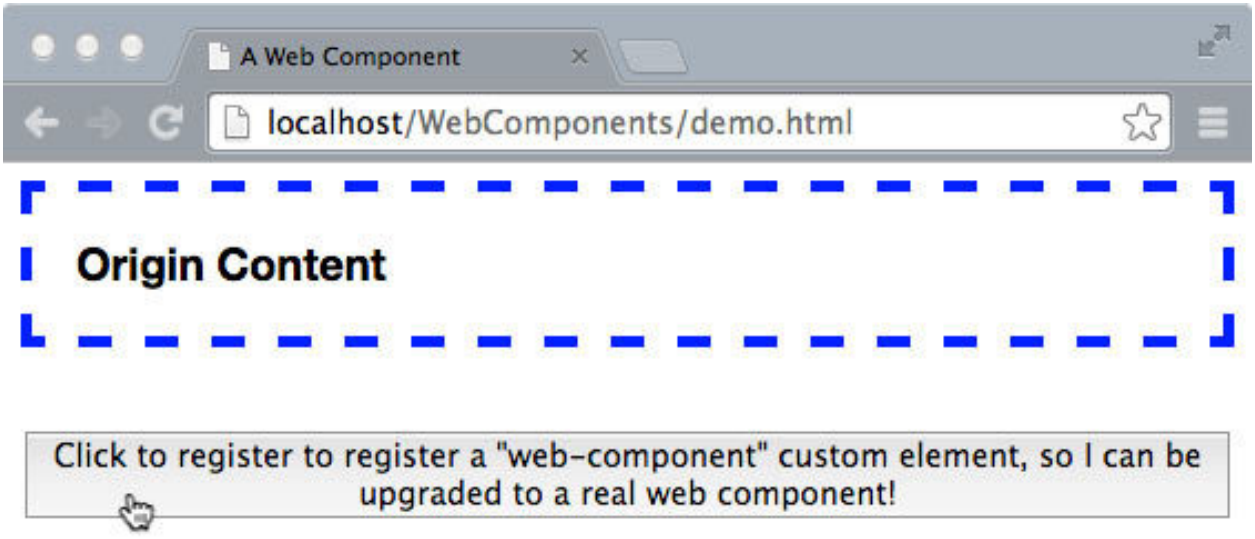
```
<!doctype html>
<html>
<head>
    <title>A Web Component</title>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible"
          content="IE=edge,chrome=1">
    <style>
        /* this styling will still apply even after
        the node has been "redistributed" */
        web-component div {
            border: 2px blue dashed;
            display: flex;
            padding: 20px;
        }
        /* uncomment this new pseudo class to prevent FOUC
        (flash of unstyled content) */
        /* :unresolved {display: none;} */
    </style>
</head>

<body>
```

```
<!-- this displays onload as an "unresolved element"
     unless we uncomment the CSS rule above -->

<web-component>
    <div>origin content</div>
</web-component>

<template id="web-component-tpl">
    <style>
        /* example of ::content pseudo element */
        ::content div {
            /* this overrides the border property in page CSS */
            border: 2px red solid;
        }
        /* this pseudo class styles the web-component element */
        :host {
            display: flex;
            padding: 5px;
            border: 2px orange solid;
            overflow: hidden;
        }
    </style>

    The origin content has been redistributed
    into the web-component shadow DOM.

    The blue dashed border should now be
    solid red because the styling in the template
    has been applied.

    <!-- the new <content> tag where shadow host element
         content will be "distributed" or transposed -->
    <content></content>

    <!-- template tag script is inert -->
    <script>
        // alert displays only after the template has been
        // attached and activated
        alert('I\'ve been upgraded!');
    </script>
</template>
```

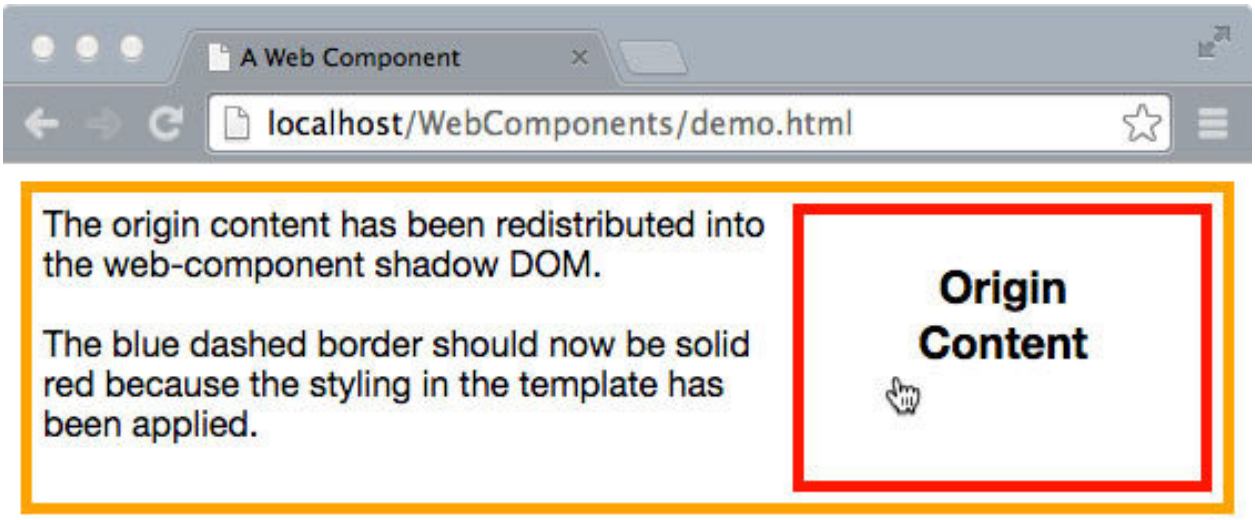
```
<script>
    // create a Namespace for our Web Component constructors
    // so we can check if they exist or instantiate them
    // programatically
    var WebCompNS = {};
    // creates and registers a web-component
    function createWebComponent (){
        // if web-component is already registered,
        // make this a no-op or errors will light up the
        // console
        if( WebCompNS.WC ) { return; }
        // this is actually not necessary unless we are
        // extending an existing element
        var WebCompProto = Object.create(HTMLElement.prototype);
        // the custom element createdCallback
        // is the opportune time to create shadow root
        // clone template content and attach to shadow
        WebCompProto.createdCallback = function() {
            // get a reference to the template
            var tpl = document.querySelector( '#web-component-tpl' );
            // create a deep clone of the DOM fragments
            // notice the .content property
            var clone = document.importNode( tpl.content, true );
            // call the new Element.createShadowRoot() API method
            var shadow = this.createShadowRoot();
            // activate template content in the shadow DOM
            shadow.appendChild(clone);
            // set up any custom API logic here, now that everything
            // is in the DOM and live
        };
        // document.registerElement(local_name, prototypeObj) API,
        // register a "web-component" element with the document
        // and add the returned constructor to a namespace module
        // the second (prototype obj) arg is included for educational
        // purposes.
        WebCompNS.CE = document.registerElement('web-component', {
            prototype: WebCompProto
        });
        // delete the button since we no longer need it
        var button = document.getElementById('make_WC');
        button.parentNode.removeChild(button);
    }
```

```
</script><br><br>

    <button id="make_WC" onclick="createWebComponent()">
        Click to register to register a "web-component" custom element,
        so I can be upgraded to a real web component!
    </button>

    <!-- concepts covered: basic custom element registration, <template>,
    <content> tags, single shadow root/host/DOM, :unresolved,
    :host pseudo classes, ::content pseudo element -->

    <!-- concepts not covered: <shadow>, multiple shadow roots,
     /deep/ combinator, element extensions -->
        </body>
    </html>
```

Above is a bare bones example of using template tags, and shadow DOM in combination with custom elements to create a nicely encapsulated web component. You can load the above into Chrome Canary and play around with it today (mid-2014). You can probably load the above in Internet Explorer sometime in 2017, and it will work there too! It illustrates most of the important CSS and DOM APIs.




**Screen grab of code listing 8.5 on browser load.**




**Screen grab after instantiating the web component including content redistribution and application of template styles to the content node.**

The portion of code contained in the <template> tag, could also be wrapped in an overloaded <script> tag if that is preferable for some reason. The only difference in the code would be to grab the <script> content as text, create an innerHTML object and assign the text content to it, and then append it to the shadow root.

The CustomElement.createdCallback(), can be used for much, much more than grabbing template content and creating shadow roots. We could grab attribute values as component API input. We could register event bindings or listeners on the shadow host, and even set up data-binding with Object.observer() when that is available in ES7. We could emit custom events to register with global services such as local storage or REST. The created callback plus any template script is where all of our custom web component behavior would live.

In this *contrived* example, we are clicking a button to run the code that creates and registers the custom element web component. There is a more practical option for packaging and running this code, which is in a separate HTML file that can be imported into the main document via another new W3C standard proposal called HTML imports. We will cover this in the next section, as well as, the rest of the new CSS and DOM APIs.

# HTML Imports (Includes)

The final specification proposal of the group that comprises "web components" is HTML import. It is essentially the ability to use HTML "includes" on the client-side. It is similar, but not identical, to the

sever-side template includes we've been doing for 20 years (remember SSIs?). It is also similar to the way we load other page resources on the client side such as scripts, CSS, images, fonts, videos, and so on. The difference is that an entire document with all the pieces can be loaded together, and it can be recursive. An HTML import document can, itself, have an import document. This makes HTML imports an ideal vehical for packaging and transporting web components and their dependencies to a browser since web components, by definition, include multiple resources. It also adds a nice option for keeping component code organizationally encapsulated.

The mechanism proposed is to add a new link type to the current HTML link tag.

```
<link rel="import" href="/components/demo.html">
```

http://www.w3.org/TR/html-imports/[22]

During the parsing of an HTML document, when a browser encounters an import link tag, it will stop and try to load it. So imports are "blocking" by default, just like <script> tags. However, they would be able to be declared with the "async" attribute to make them non-blocking like script tags as well. Import link tags can also be generated by script and placed in the document to allow lazy loading. As with other resource loaders, import links have onload and onerror callbacks.

## Security and Performance Considerations

The same-origin security restrictions as iframes apply for the obvious reasons. So importing HTML documents from a different domain would require CORS headers set up by the domain exporting the web component document. In most cases, this would likely be impractical unless both domains had some sort of pre-existing relationship. In the scenario where both domains are part of the same enterprise organization, this is very doable. However if the web components are exported by an unrelated, third party organization, it would likely make more sense just to prefetch and serve them from the same server (be sure to check the licensing first). Also, for best performance, it may make sense to prebundle the necessary imports as part of the initial download similar to the way we pre-cache AngularJS templates.

Since these are HTML documents, browser caching is available which should also be considered when developing the application architecture. An HTML import document will only be fetched once no matter how many link elements refer to it assuming browser caching is turned on. Apparently any scripts contained in an import are executed asynchronously (non-blocking) by default.

One scenario to avoid at all costs is the situation where rendering depends on a complex web component that itself imports web components as dependencies multiple levels deep! This would mean multiple network requests in sequence. An "asynchronous document loader" library would certainly be needed. Another option is to "flatten" them into a single import which is what the NPM module, **vulcanize** from the Polymer team, does.

[22] http://www.w3.org/TR/html-imports/

## Differences from Server-side Includes

Using includes on the server-side, whether PHP, JSP, or ERB assembles the page which the client considers a single, logical document or DOM. This is not the case with HTML imports. Having HTML imports in a page means that there are multiple, logical documents or DOMs in the same *window* context. There is the primary DOM associated with the browser frame or window, and then there are the sub-documents.

This is where loading CSS, JavaScript, and HTML together can get confusing (and non-intuitive if you are used to server-side imports). So there are two things to understand which will help speed development and debugging.

1. CSS and JavaScript from an HTML import executes in the window context the same as non-import CSS and JavaScript.
2. The HTML from an HTML import is initially in a separate logical document (and is not rendered, similar to template HTML). Accessing and using it from the master document requires obtaining a specific reference to that HTML imports document object

Number two can be a bit jarring since includes on the server-side which we are most used to are already inlined. The major implication to these rules are that any JavaScript that access the included HTML will not work the same way if the HTML include is served as a stand-alone page. For example, to access a <template id=”web-component-tpl”> template in the imported document we would do something like the following to access the template and bring it into the browser context:

```
var doc = document.querySelector( 'link[rel="import"]' );
if(doc){
    var template = doc.querySelector( '#web-component-tpl' );
}
```

This holds true for script whether it runs from the root document or from the import document. With web components it will be typical to package the template and script for the custom element together, so it is most likely that the above logic would execute from the component document.

However, consider the scenario in which we want to optimize the initial download of a landing page by limiting rendering dependencies on subsequent downloads. What we do is inline the content on the server-side, so it is immediately available when the initial document loads. If the component JavaScript tries the above when inlined it will fail.

Fortunately, since “imported” JavaScript runs in the window context it can first check to see if the ownerDocument of the template is the same as the root document and then use the appropriate reference for accessing the HTML it needs. Specifically,

**8.6 HTML Import Agnostic Template Locator Function**

```
// locate a template based on ID selector
function locateTemplate(templateId){
    var templateRef = null;
    var importDoc = null;
    // first check if template is in window.document
    if(document.querySelector( templateId )){
        templateRef = document.querySelector(templateId);
    // then check inside an import doc if there is one
    } else if ( document.querySelector('link[rel="import"]') ){
        // the new ".import" property is the imported
        // document root
        importDoc = document.querySelector('link[rel="import"]').import;
        templateRef = importDoc.querySelector(templateId);
    }
    // either returns the reference or null
    return templateRef;
}

var tpl = locateTemplate('#web-component-tpl');
```

The above function can be used in the web component JavaScript to get a template reference regardless of whether the containing document is *imported* or *inlined*. This function could and should be extended to handle 1) multiple HTML imports in the same document, and 2) HTML imports nested multiple levels deep. Both of these scenarios are all but guaranteed. However, this specification is still far from maturity, so hopefully the HTML import API will add better search capability by the time it becomes a standard.

A more general purpose way to check from which document a script is running is with the following comparison:

```
document.currentScript.ownerDocument === window.document
```

The above evaluates to true if not imported and false if imported.

# Some JavaScript Friends of Web Components

We have covered the three W3C standards proposals, plus one current standard, that comprise what's being called Web Components, templates, custom elements, shadow DOM, and HTML imports. There was one more called "decorators", but that mysteriously disappeared.

In this section we will take a look at some new JavaScript features on the horizon that aren't officially part of "Web Components" but will play a major supporting role to web component architecture

in the areas of encapsulation, modularity, data-binding, typing, type extensions, scoping, and performance. The net effect of these JavaScript upgrades will be major improvement to the "quality" of the components we will be creating. Most of these features, while new to JavaScript, have been around for a couple decades in other programming languages, so we won't be spending too much time explaining the nuts and bolt unless specific to browser or asynchronous programming.

# Object.observe - No More Dirty Checking!

`Object.observe(watchedObject)` is an interface addition to the JavaScript Object API. This creates an object watcher (an intrinsic JavaScript object like scope or prototype) that will asynchronously execute callback functions upon any mutation of the "observed" object without affecting it directly. What this means in English is that for the first time in the history of JavaScript we can be notified about a change in an object of interest natively instead of us (meaning frameworks like AngularJS and Knockout) having to detect the change ourselves by comparing current property values to previous values every so often. This has big implications for data-binding between model and view (the VM in MVVM).

What's awesome about this is that crappy data-binding performance and non-portable data wrappers will be a thing of the past! What sucks is that this feature will not be an official part of JavaScript until ECMA Script 7, and as of mid 2014, ECMA 5 is still the standard. Some browsers will undoubtedly implement this early including Chrome 36+, Opera, and Mozilla, and polyfills exist for the rest including the NPM module "observe-js".

Observables are not actually part of W3C web components because they have nothing to do with web component architecture directly, and the standards committee is a separate organization, ECMA. But we are including some discussion about them because they have a big influence on the MVVM pattern which is an integral pattern in web component architecture. AngularJS 2.0 will use `Object.observe` for data-binding in browsers that support it with "dirty checking" as the fallback in version 2.0. According to testing conducted, `Object.observe` is 20 to 40 times faster on average. Likewise, other frameworks like Backbone.js and Ember that use object wrapper methods such as set(datum), get(datum), and delete(datum) to detect changes and fire callback methods will become much lighter.

http://wiki.ecmascript.org/doku.php?id=harmony:observe[23]

This proposal actually has six JavaScript API methods. The logic is very similar to the way we do event binding now.

## Proposed API

- Object.observe(watchedObject, callbackFn, acceptList) - The callback function automatically gets an array of **notification objects** (also called **change records**) with the following properties:

[23] http://wiki.ecmascript.org/doku.php?id=harmony:observe

- type: type of change including “add”, “update”, “delete”, “reconfigure”, “setPrototype”, “preventExtensions”
- name: identifier of the mutated property
- object: a reference to the object with the mutated property
- oldValue: the value of the property prior to mutation

The type of changes to watch for can be paired down with the optional third parameter which is an array contaning a subset of the change types listed above. If this is not included, all types are watched.

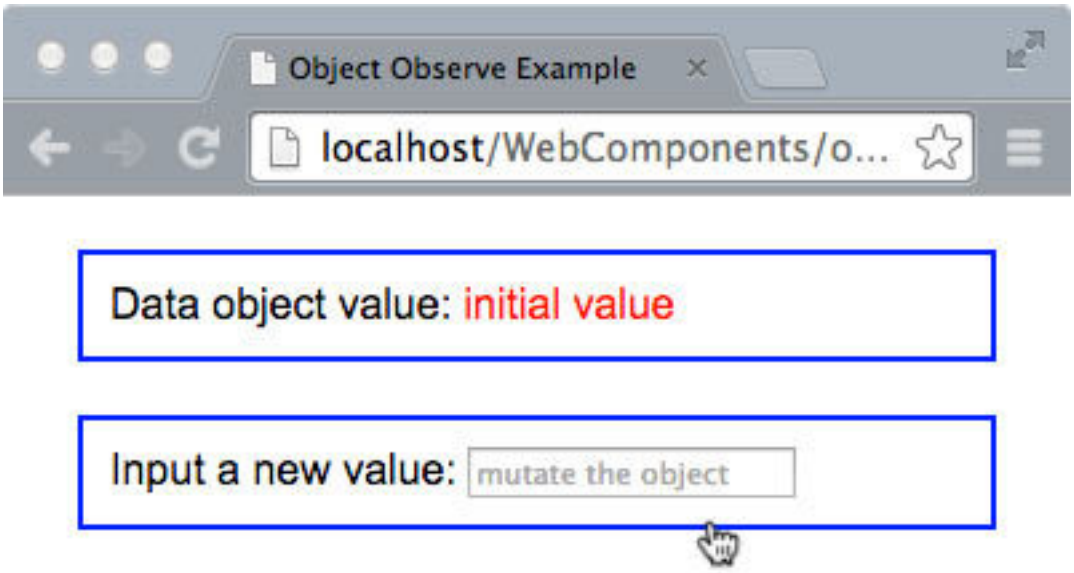
- Object.unobserve(watchedObject, callbackFn) - Used to unregister a watch function on an object. The callback must be a named function that matches the one set with Object.observe. Similar to unbinding event handlers, this is used for cleanup to prevent memory leaks.
- Array.observe(watchedArray, callbackFn) - Same as Object.observe but has notification properties and change types related specifically to arrays including:
- type: type of array action including as **splice**, **push**, **pop**, **shift**, **unshift**
- index: location of change
- removed: and array of any items removed
- addedCount: count (number) of any items added to the array
- Array.unobserve(watchedArray, callbackFn) - This is identical to Object.unobserve
- Object.deliverChangeRecords(callback) - Notifications are delivered asynchronously by default so as not to interfere with the actual “change” sequence. Since a notification is placed at the bottom of the call stack, all the code surrounding the mutation will execute first, and there is no guarantee when exactly the notification will arrive. If a notification must be delivered at some specific point in the execution queue because it is a dependency for some logic, this method may be used to deliver any notifications *synchronously*.
- Object.getNotifier(O) - This method returns the notifier object of a watched object that can be used to define a custom notifier. It can be used along with Object.defineProperty() to define a set method that includes retrieving the notifier object and adding a custom notification object to its notify() method other than the default above.

#### 8.7 An Example of Basic Object.observe() Usage

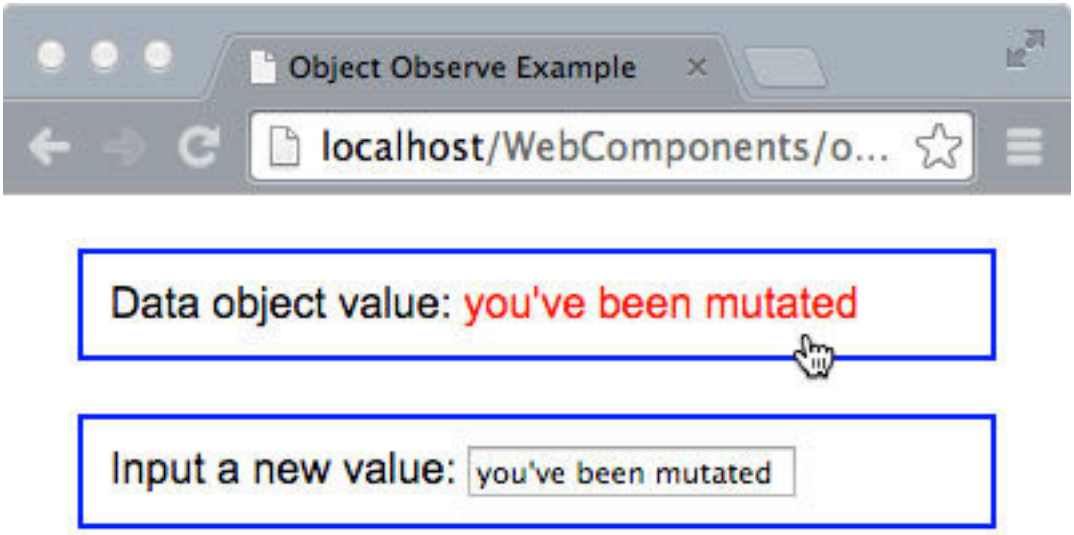
```
<!DOCTYPE html>
<html>
<head lang="en">
    <meta charset="UTF-8">
    <title>Object Observe Example</title>
    <style>
        body{font-family: arial, sans-serif;}
        div {
```

```
            padding: 10px;
            margin: 20px;
            border: 2px solid blue;
        }
        #output {
            color: red;;
        }
    </style>
</head>

<body>
<div>
    <label>Data object value: </label>
    <span id="output">initial value</span>
</div>

<div>
    <label>Input a new value: </label>
    <input
            type="text"
            name="input"
            id="input"
            value=""
            placeholder="new value">
</div>

<script>
// the data object to be "observed"
var dataObject = {
    textValue: 'initial value'
};
// user input
var inputElem = document.querySelector('#input');
// this element's textValue node will be "data bound"
// to the dataObject.textValue
var outputElem = document.querySelector('#output');
// the keyup event handler
// where the user can mutate the dataObject
function mutator(){
    // update the data object with user input
    dataObject.textValue = inputElem.value;
}
```

```
// the mutation event handler
// where the UI will reflect the data object value
function watcher(mutations){
    // change objects are delivered in an array
    mutations.forEach(function(changeObj){
        // the is the actual data binding
        outputElem.textContent = dataObject.textValue;
        // log some useful change object output
        console.log('mutated property: ' + changeObj.name);
        console.log('mutation type: ' + changeObj.type);
        console.log('new value: ' + changeObj.object[changeObj.name]);
    });
}
// set a keyup listener for user input
// this is similar to AngularJS input bindings
inputElem.addEventListener('keyup', mutator, false);
// set a mutation listener for the data object
// notice how similar this is to setting event listeners
Object.observe(dataObject, watcher);
</script>

</body>
</html>
```




**The `Object.observe()` example upon load**




**After the user has changed the data object value.**

This example demonstrates a very basic usage of `Object.observe()`. One data property is bound

to one UI element via the interaction of an event handler with a mutation observer. In real life an event handler alone is sufficient for the functionality in this example.

The actual value of `Object.observe` are situations where data properties are mutated either indirectly from user events or directly from data updates on the server, especially for real-time connections. There will no longer be a need for comparing data values upon events to figure out what might have changed, and there will no longer be a need for data wrapper objects such as Backbone.model to fire change events.

That said, in practice there will be a lot of boiler plate involved in creating custom watcher and notifiers. So it is likely that this API will be abstracted into a data-binding libraries and frameworks or utilized be existing frameworks for significantly better UI performance. This addition to JavaScript will allow web components to be quite a bit more user responsive, less bulky codewise, and better enabled for real-time data updates.

# New ES6 Features Coming Soon

Object mutation observers are still a ways off, but some other great updates to JavaScript that enhance UI component code will be here a lot sooner, if not already. In fact, ES6 includes a number of enhancements that will help silence those high-horse computer scientists who look down on JavaScript as not being a real programming language. Features include syntax and library additions such as arrow functions, destructuring, default parameter values, block scope, symbols, weak maps, and constants. Support is still spotty among major browsers, especially for the features listed below that are of most use for component development. Firefox has implemented the most features followed by Chrome. You can probably guess who supports the least number of features. A nice grid showing ES6 support across browers can be found at:

http://kangax.github.io/compat-table/es6/[24]

Our coverage is merely a quick summery of these JavaScript features and how they benefit UI component architecture. You are encouraged to gather more information about them from the plethora of great examples on the web. Plus, you are probably already familiar with and using many of these features via a library or framework.

## Modules and Module Loaders

Modules and imports are a functionality that is a standard feature of almost all other languages, and has been sorely missing from JavaScript. Today we use AMD (asynchronous module definitions) via libraries such as Require.js on the client-side. Most of these handle loading as well. CJS (CommonJS) is primarily used server-side, especially in Node.js.

These libraries would be obsoleted by three new JavaScript keywords:

[24] http://kangax.github.io/compat-table/es6/

- module - would precede a code block scoped to the module definition.
- export - will be used within module declaration to expose that module's API functionality
- import - like the name says, will be for importing whole modules or specific functionality from modules into the current scope

Modules will be loadable with the new Loader library function. You will be able to load modules from local and remote locations, and you will be about to load them asynchronously if desired. Properties and methods defined using var within a module definition will be scoped locally and must be explicitly exported for use by consumers using export. For the most part, this syntax will obsolete the current pattern of wrapping module definitions inside self-executing-anonymous-functions.

Just like shadow DOM with HTML, ES6 modules will help provide solid boundaries and encapsulation for web component code. This feature will likely take longer to be implemented across all major browser given that in mid-2014 the syntax is still under discussion. Regardless, it will be one of the most useful JavaScript upgrades to support UI component architecture.

EXAMPLE??

## Promises

Promises are another well known pattern, supported by many JavaScript libraries, that will finally make its way into JavaScript itself. Promises provide a lot of support for handling asynchronous events such as lazy loading or removing a UI component and the logic that must be executed in response to such events without an explosion of callback spaghetti. The addition to native JavaScript allows promise support removal from libraries lightening them up for better performance. The Promise API is currently available in Firefox and Chrome.

## Class Syntax

Though JavaScript is a functional prototyped language, many experienced server language programmers have failed to embrace this paradigm. This can be reflected by the many web developer job posting that include "object oriented JavaScript" in the list of requirements.

To help placate this refusal to embrace functional programming, a minimal set of class syntax is being added. It's primarily syntactic sugar on top of JavaScript's prototypal foundation, and will include a minimal set of keywords and built-in methods.

You will be able to use the keyword "class" to define a code block that may include a "constructor" function. As part of the class statement, you will also be able to use the keyword "extends" to subclass an existing class. Accordingly, within the constructor function, you can reference the constructor for the super-class using the "super" keyword. Just like Java, constructors are assumed as part of a class definition. If you fail to provide one explicitly, one will be created for you.

The primary benefit of class syntax to UI component architecture will be a minor improvement in JavaScript code quality coming from developers who are junior, "offshore" contractors, or weekend dabblers who studied object-oriented programming in computer science class.

EXAMPLE??

## Template Strings

Handling strings has always been a pain-in-the-ass in JavaScript. There has never been the notion of multi-line strings or variable tokens. The current string functionality in JavaScript was developed long before the notion of the single page application where we would need templates compiled and included in the DOM by the browser rather than from the server.

The "hacks" that have emerged to support maintaining template source code as HTML including Mustache.js and Handlebars.js are bulky, non-performant, and prone to XSS attacks since they all must use the evil "eval" at some point in the compilation process.

http://tc39wiki.calculist.org/es6/template-strings/[25]

With the arrival of template strings to JavaScript, we can say goodbye to endless quotes and "+" operators when we wish to inline multi-line templates with our UI components. We will be able to enclose strings within back ticks ( `multi-line string` ), and we will be able to embed variable tokens using familiar ERB syntax ( `Hello: ${name}!` ).

## Proxies

The proxy API will allow you to create objects that mimic or represent other objects. This is a rather abstract definition because there are many use-cases and applications for proxies. In fact, complete coverage for JavaScript proxies including cookbook examples could take up a full section of a book.

Some of the ways proxies can benefit UI component architecture include:

- Configuring component properties dynamically at runtime
- Representing remote objects. For example, a component could have a persistence proxy that represents the data object on a server.
- Mocking global or external objects for easier unit testing

Expect to see proxies used quite a bit with real-time, full duplex web applications. A proxy API implementation is currently available to play with in Firefox, and more information can be gleaned at:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy[26]

[25] http://tc39wiki.calculist.org/es6/template-strings/

[26] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy

# Google Traceur Compiler

If you are itching to start writing JavaScript code using ES6 syntax, Google's Traceur compiler can be used to translate it into ES5 which all major bowsers understand.

https://github.com/google/traceur-compiler/wiki/GettingStarted[27]

http://google.github.io/traceur-compiler/demo/repl.html[28]

Before using this for any production projects make sure you are well aware of any ES6 syntax or APIs that are still not completely "standardized" or likely to be implemented by certain browsers. Using these features can result in code that is not maintainable or prone to breakage. Also understand that with any JavaScript abstraction, the resulting output can be strange or unintelligible, therefore, difficult to debug.

# AngularJS 2.0 Roadmap and Future Proofing Code

Web components and ES6 will be fantastic for encapsulating logic, functionality, styling and presentation for reusable pieces of a user interface. However, we will still need essential non-UI services that operate in the global or application level scope to turn a group of web components into an application. While old school widget libraries will disappear, application frameworks that supply services for things like REST, persistence, routing, and other application level singletons are not going anywhere anytime soon.

Now that these DOM and JavaScript additions are on the horizon, some of the more popular and forward leaning JavaScript frameworks including AngularJS and Ember are looking to incorporate these capabilities as part of their roadmap. The plan for AngularJS 2.0, is to completely rewrite it using ES6 syntax for the source code. Whether or not it is transpiled to ES5 with Traceur for distribution will depend on the state of ES6 implementation across browsers when 2.0 is released.

Another major change will be with directive definitions. Currently, the directive definition API is a one-size-fits-all implementation no matter what the purpose of the directive is. For 2.0, the plan is to segment directives along the lines of current patterns of directive use. The current thinking is three categories of directive. Component directives will be build around shadow DOM encapsulation. Template directives will use the <template> tag in some way, and decorator directives will be for behavior augmentation only. Decorator directives will likely cover most of what comprises AngularJS 1.x core directive, and they will likely be declared only with element attributes.

One of the hottest marketing buzz words in the Internet business as of mid-2014 is "mobile first", and AngularJS 2.x has jumped on that bandwagon just like Bootstrap 3.x. So expect to see things like touch and pointer events in the core. The complete set of design documents for 2.0 can be viewed on Google Drive at:

[27] https://github.com/google/traceur-compiler/wiki/GettingStarted
[28] http://google.github.io/traceur-compiler/demo/repl.html

https://drive.google.com/?pli=1#folders/0B7Ovm8bUYiUDR29iSkEyMk5pVUk[29]

Brad Green, the developer relations guy on the AngularJS team, has a blog post that summarizes the roadmap.

http://blog.angularjs.org/2014/03/angular-20.html[30]

# Writing Future Proof Components Today

There are no best practices or established patterns for writing future-proof UI components since the future is always a moving target. However, there are some suggestions for increasing the likelihood that major portions of code will not need to be rewritten.

The first suggestion is keeping the components you write today as encapsulated as possible- both as a whole and separately at the JavaScript and HTML/CSS level. In this manner eventually you will be able to drop the JavaScript in a module, and the HTML/CSS into template elements and strings. Everything that has been emphasized in the first 2/3 of this book directly applies.

The second suggestion is to keep up to date with the specifications process concerning Web Components and ES6/7. Also watch the progress of major browser, and framework implementations of these standards. Take time to do some programming in ES6 with Traceur, and with Web Components using Chrome Canary, and the WC frameworks that we shall introduce in the next chapter. By making this a habit, you will develop a feel for how you should be organizing your code for the pending technologies.

# Summary

Web Components will change the foundation of web development, as the user interface for applications will be built entirely by assembling components. If you are old enough to remember and have done any desktop GUI programming using toolkits like Java's Abstract Windowing Toolkit (AWT), then we are finally coming full circle in a way. In AWT you assembled GUIs out of library components rather than coding from scratch. Eventually we will do the same thing with Web Components.

In the next chapter we will visit the current set of frameworks and libraries build on top of Web Component specifications including Polymer from Google, and X-Tag/Brick from Mozilla.

[29] https://drive.google.com/?pli=1#folders/0B7Ovm8bUYiUDR29iSkEyMk5pVUk

[30] http://blog.angularjs.org/2014/03/angular-20.html