

# **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



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

©2014 - 2015 David Shapiro

# Contents

<b>Preface</b>	<b>1</b>
Release Notes	2
Conventions	2
Git Repo, Bugs, and Suggestions	3
<b>Introduction</b>	<b>4</b>
The Problem...	4
A Brief History of Front-End development for the Web	4
Why AngularJS?	6
What this Book Is and Isn't	7
<b>Chapter 1 - Web UI Component Architectures</b>	<b>8</b>
Key Patterns in UI Component Development	12
MVC, MVP, MVVM, MVwhatever	13
Dependency Injection (IOC)	14
Observer and Mediator Patterns	15
Module Pattern	15
Loose Coupling of Dependencies	16
Summary	17
<b>Chapter 2 - AngularJS As a UI Component Framework</b>	<b>18</b>
Why AngularJS? (part 2)	19
Dynamic Views	20
Two Way Data Binding	20
Dependency Injection	21
Test Driven Development (TDD)	22
Don't Repeat Yourself (DRY)	22
POJO Models	23
AngularJS and Other Frameworks in the Same Page	25
Forward Compatibility with W3C Web Components and MDV	26
Summary	27
<b>Chapter 3 - AngularJS Component Primer Part One: The View</b>	<b>28</b>
AngularJS Core Directives	28

## CONTENTS

Scoping Directives ng-app, ng-controller . . . . .	28
Event Listener Directives . . . . .	30
DOM & Style Manipulation Directives . . . . .	31
Rolling Our Own Directives . . . . .	33
Directive Declaration . . . . .	33
Directive Naming . . . . .	34
Directives Definition . . . . .	34
Dependency Injection (API) Strategies . . . . .	45
The Directive Lifecycle . . . . .	47
Testing Directives . . . . .	48
AngularJS 2.0, Web Components and Directive Re-classification . . . . .	49
Summary . . . . .	49
<b>Chapter 4 - AngularJS Component Primer Part 2: Models and Scope . . . . .</b>	<b>50</b>
Data Models in AngularJS . . . . .	50
Data Representation in the View . . . . .	51
\$scope creation and destruction . . . . .	52
Built In \$scope methods and properties . . . . .	55
Decoupled Communication . . . . .	56
Data Binding and Execution Context . . . . .	57
Debugging Scopes and Testing Controllers . . . . .	59
Summary . . . . .	60
<b>Chapter 5 - Standalone UI Components by Example . . . . .</b>	<b>61</b>
Building a <i>Smart Button</i> component . . . . .	63
Directive definition choices . . . . .	66
Attributes as Component APIs . . . . .	67
Events and Event Listeners as APIs . . . . .	70
Advanced API Approaches . . . . .	73
What About Style Encapsulation? . . . . .	79
Style Encapsulation Strategies . . . . .	80
Unit Testing Component Directives . . . . .	81
Setting Up Test Coverage . . . . .	83
Component Directive Unit Test File . . . . .	84
Unit Tests for our Component APIs . . . . .	86
Summary . . . . .	89
<b>Chapter 6 - UI Container Components by Example . . . . .</b>	<b>90</b>
Defining a Set of Navigation Components . . . . .	92
Building the UI Components . . . . .	102
The Menu Item Component . . . . .	102
Menu Item API Unit Test Coverage . . . . .	104
The Dropdown Component . . . . .	107

## CONTENTS

Reusing Existing Open Source Code . . . . .	107
Dropdown Menu Unit Test Coverage . . . . .	122
Global Navigation Bar Container . . . . .	126
NavBar API Documentation . . . . .	139
NavBar Unit Test Coverage . . . . .	140
Summary . . . . .	149
<b>Chapter 7 - Build Automation &amp; Continuous Integration for Component Libs . . . . .</b>	<b>151</b>
Delivering Components To Your Customers . . . . .	151
Source Code Maintenance? . . . . .	152
Tooling . . . . .	153
UI Library Focused . . . . .	154
Node.js and NPM . . . . .	157
Task Runners and Workflow Automation . . . . .	158
Grunt - The JavaScript Task Runner . . . . .	159
Useful Tasks for a Component Lib Workflow . . . . .	164
Gulp.js - The Streaming Build System . . . . .	187
Continuous Integration and Delivery . . . . .	191
Travis-CI . . . . .	193
Bower Package Registry . . . . .	195
Versioning . . . . .	197
Salvaging Legacy Applications with AngularJS UI Components . . . . .	199
Summary . . . . .	201
<b>Chapter 8 - W3C Web Components Tour . . . . .</b>	<b>203</b>
The Roadmap to Web Components . . . . .	203
The Stuff They Call “Web Components” . . . . .	205
Custom Elements . . . . .	206
Shadow DOM . . . . .	211
Using the <template> Tag . . . . .	220
HTML Imports (Includes) . . . . .	226
Some JavaScript Friends of Web Components . . . . .	229
Object.observe - No More Dirty Checking! . . . . .	230
New ES6 Features Coming Soon . . . . .	235
Google Traceur Compiler . . . . .	238
AngularJS 2.0 Roadmap and Future Proofing Code . . . . .	238
Writing Future Proof Components Today . . . . .	239
Summary . . . . .	239
<b>Chapter 9 - Building with Web Components Today . . . . .</b>	<b>240</b>
Platform.js (Webcomponents.js) - Web Components Polyfills . . . . .	241
Polymer.js . . . . .	247
Polymer Based Libraries . . . . .	255

## CONTENTS

Mozilla X-Tags . . . . .	255
<b>Chapter 10 - Integrating Web Components with AngularJS using Angular Custom Element</b>	<b>257</b>

# 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 transferred 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><sup>1</sup>

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

<https://github.com/dgs700/angular-custom-element><sup>2</sup>

---

<sup>1</sup><https://github.com/dgs700/angularjs-web-component-development>

<sup>2</sup><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 server-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. Ember.js, 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](http://en.wikipedia.org/wiki/Component-based_software_engineering)<sup>3</sup>

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)

---

<sup>3</sup>[http://en.wikipedia.org/wiki/Component-based\\_software\\_engineering](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\\_\(computer\\_science\)](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\  
ction 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 it 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 its 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/><sup>4</sup>

<http://requirejs.org/docs/whyamd.html><sup>5</sup>

## 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.

---

<sup>4</sup><http://www.commonjs.org/>

<sup>5</sup><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 2 - AngularJS As a UI Component Framework

AngularJS is a versatile client-side MVC/VM framework since it provides pattern implementations for both controllers and view-models (IMO). It provides most of the structural features necessary for constructing the proverbial single page app (SPA) including client routing, templating, data modeling, app configuration, logic containers and service hooks for interacting with the outside world via AJAX or REST. The AngularJS team is adding more application features with every release, and there are already several books about building browser apps with AngularJS on the market.

The goal here is not to be yet another book about how to build a web app, but rather to discuss some AngularJS “goodies” that transcend the above list of app framework features offered by most client-side frameworks that allow us to construct dynamic, interactive UI components that can be used and reused across contexts, pages, applications, and organizations.

To illustrate, suppose you are a web developer or designer for a large enterprise working on a specific micro-site or application, you likely will have many corporate standards and constraints that must be adhered to such as style guides, uniform look-and-feel, legacy APIs, etc. as part of the development process. Including a simple, standard component such as a search bar or field in a page, can become quite complex when considering what CSS rules to apply plus extra JavaScript for interaction behavior, offering auto-complete, URLs to talk to, error message display, and so on in a way that adheres to corporate standards for consistent styling and behavior. This task can easily result in the creation of hundreds of lines of HTML, CSS, and JavaScript boiler plate code plus many developer hours to implement.

On the other hand, what if the task could be so easy as adding a single, custom HTML tag in a page such as `<search-bar></search-bar>`, and any special configuration could be achieved via custom attributes such as `<search-bar color="blue" autocomplete="false">`. Entire pages and new apps could be built quickly out of such custom UI components for headers, footers, navigation, sign-in widgets and more with simple, declarative markup. Building large parts of a website that adheres to enterprise style guidelines could be as easy as loading a component pallet in your browser, making some configuration choices, and including the custom HTML tag with attribute configurations in your markup.

Developers can be freed to focus on the application to be built rather than all the page boilerplate, code bases can be drastically shrunk, and development time can be drastically reduced. In fact, AngularJS was initially conceived and developed in 2009 by Google employee, Miško Hevery, for just this purpose.

In AngularJS, custom, declarative markup that dynamically extends the static nature of HTML, such as our `<search-bar>` example above, lies at the core of the framework. AngularJS terms these



as **directives**. Angular directives can be used for all kinds of dynamic UI behavior. The framework comes with many built in such as `ng-click` for handling logic upon a mouse click event. But the framework also gives you the ability to define your own, and this becomes a handy tool for encapsulating our own units of business logic with visual representation. In fact, a directive can include an inline HTML template with CSS along with its own controller and data scope.

This is very similar to the idea behind the emerging W3C Web Components standards which are still a ways away. However, we can emulate a web component using AngularJS directives, and the AngularJS team intends to evolve the framework to converge with the Web Components standards.

We will discuss both of the above extensively in this book, but first we need to step back a bit and start with a good foundation of the design features and goals of AngularJS for proper context.

## Why AngularJS? (part 2)

What advantages does it have over Backbone.js or YUI or Extjs? Why would I want to switch? What's the learning curve like, and so on.

These are some of the first questions many UI developers who have been coding in jQuery or other imperative frameworks ask. The creators of the various popular frameworks in use all have opinions as to why their approach is best. Some think that JavaScript is not a real language and the developer needs to be protected from it. Some think that JavaScript should only be an optional addition to a web page. Some think that object oriented programming is the best programming paradigm, so they add all kinds of faux OO to JavaScript, and some are very opinionated about having no "opinion" at all.

In the past, some of these opinions held more weight than they do now. In 2005 a large percentage of web surfers kept their Javascript disabled, so it did not make sense to create pages that were dependent on it for proper functioning. "Progressive enhancement" or "graceful degradation" were important, so non-obtrusive, imperative frameworks like jQuery and Prototype made sense. The Javascript (behavior) should be entirely separate from the content (html). If Javascript were turned off, the surfer would have, at worst, a degraded but functional experience.

This is not true today. JavaScript can still be turned off in browsers, but it is now accepted as safe, ubiquitous, and necessary for browsing any modern website.

The AngularJS team has some strong opinions as to why their approach is best and why you'd want to do things the "Angular way". From the landing page of [Angularjs.org](http://angularjs.org):

*"HTML is great for declaring static documents, but it falters when we try to use it for declaring dynamic views in web-applications. AngularJS lets you extend HTML vocabulary for your application. The resulting environment is extraordinarily expressive, readable, and quick to develop."*

## Dynamic Views

“Other frameworks deal with HTML’s shortcomings by either abstracting away HTML, CSS, and/or JavaScript or by providing an imperative way for manipulating the DOM. Neither of these address the root problem that HTML was not designed for dynamic views.”

This argument would have been quite a stretch in 2005, but it actually makes sense today. The slow evolution of HTML has not kept pace with the way we use the Web. Today the web is highly interactive, and waiting for the server to respond to each interaction is only still seen in some big enterprise, intranet applications that handle payroll, HR, purchasing and other corporate applications.

The fact is that any web UI developer spends an inordinate amount of time writing code that queries a DOM element, attaches an event listener to it, and manipulates it in response. Much of these dynamic behaviors are now ubiquitous across the web and organizations, including UI widgets like carousels, modals, transitions, tab panels, etc.

Ideally we should be able to include these widgets *declaratively* via an HTML element such as `<carousel>`, and these elements should be defined, configurable, and extensible as part of the HTML specification, rather than worrying about including extra JavaScript and CSS.

Along the same lines, representing or modeling the application’s data in the client also requires a lot of boilerplate code. This includes adding custom event listeners and handlers for changes in a data object, the belief that model objects should adhere to object oriented paradigms including inheritance, interfaces, getters, setters, etc. For a change in a datum to be reflected in the view, and visa versa there is a lot to wire up programmatically, and a lot to keep track of. Conceptually, the current (jQuery) way of managing dynamic behavior requires maintaining two parallel tree structures of information, one for HTML and one for *imperative* DOM manipulation.

Including dynamic DOM behavior declaratively as HTML markup is nothing new. Dojo and Bootstrap’s widgets have given the page author the option of declarative inclusion for quite some time.

## Two Way Data Binding

The AngularJS team is of the opinion that data-binding between model and view should be intrinsic. Again, from the AngularJS.org landing page:

*“Data-binding is an automatic way of updating the view whenever the model changes, as well as updating the model whenever the view changes. This is awesome because it eliminates DOM manipulation from the list of things you have to worry about. ...Controllers are the behavior behind the DOM elements. AngularJS lets you express the behavior in a clean readable form without the usual boilerplate of updating the DOM, registering callbacks or watching model changes. ...Unlike other frameworks, there is no need to inherit from proprietary types; to wrap the model in accessor*

*methods. Just plain old JavaScript here. This makes your code easy to test, maintain, reuse, and again free from boilerplate.”*

Behind the two way data binding is the VM or view-model component of MVVM. The view-model is the object representing the state of both the application data and the views representation of it. The flow is bi-directional. A change on one side necessitates a change on the other automatically.

The VM in AngularJS is represented by the **\$scope** object which the framework automatically *injects* into controller functions as a parameter.

## 2.0 AngularJS 2 way data binding

---

```
// any variable attached to $scope can have live representation in the view
myApp.controller('myController', function($scope){
    $scope.myAppModel = {
        firstName: 'Dave',
        occupation: 'UI Architect'
    };
});

<!-- if we type in new values, they will display in real time -->
<div ng-controller="myController">
    <span> {{ myAppModel.firstName }} </span><br>
    <input type="text" name="firstName" ng-model="myAppModel.firstName">
    <span> {{ myAppModel.occupation }} </span><br>
    <input type="text" name="occupation" ng-model="myAppModel.occupation">
</div>
```

---

AngularJS currently accomplishes (view-model) data-binding via “dirty-checking”, a process of recursively comparing current data values for each **\$scope** object with previous values to see what has changed if there has been an associated UI event or “watched” model event.

This can be an expensive process in the currently implemented version of JavaScript (ECMA5). However, the need for “model driven” views (MDV) is well recognized, and the next version of JavaScript will support `Object.observe()` natively, and AngularJS will naturally take advantage of this.

The concept of AngularJS **\$scope** is an important and complex topic, so we will examine it in quite a bit more depth in chapter 4.

## Dependency Injection

Speaking of *injection*, AngularJS encourages a lot of dependency injection (DI), a programming paradigm made popular by the Java Spring Framework several years ago.

Any application, module, or function in JavaScript can resolve dependencies as variables from within that function's scope, a parent scope or the global scope. Direct dependencies outside a function's immediate scope are bad, and global dependencies are considered evil among experienced JavaScript developers and especially the AngularJS team- not just due to global namespace pollution and the potential for naming collisions and overwrites, but because a tight coupling between a function or module dependency and a global variable make its quite hard to write valid unit tests.

## Test Driven Development (TDD)

By building a framework around the idea of explicitly injecting dependencies, the AngularJS team made it quite easy for the developer to isolate individual units of code. Additionally, AngularJS is shipped with several configurable mock services for mimicking dependencies outside the framework such as AJAX calls and results, and on top of that the “official” AngularJS tutorial and the Angular-seed project on GitHub include an integrated testing environment using Jasmine and Karma for unit and end-to-end testing respectively.

<https://github.com/angular/angular-seed><sup>6</sup>

<http://docs.angularjs.org/tutorial><sup>7</sup>

So AngularJS developers are highly encouraged to create unit tests for their functions as they write them, and there really is no excuse for not doing so. This is especially true when building reusable UI components that may be consumed by 3rd parties. It is critical that the APIs of the components we create maintain consistent behavior as the internals of the component are developed from one version to the next.

If JavaScript were a compiled language, the compiler would catch many bugs for us. But JavaScript is a dynamically typed, interpreted language, so comprehensive unit and e2e testing is our first line of defense.

## Don't Repeat Yourself (DRY)

In the AngularJS world, *EVERYTHING* is a module. If you are familiar with asynchronous dependency loaders like **Require.js**, and how they allow us to include library and application dependencies without having to worry about when and where they come from, then AngularJS modules should be easy to understand.

The AngularJS team, like the rest of us, hates to repeat the same blocks of code needlessly. Like a lot of the jQuery binding boilerplate, repeated code is a large contributor to the problem of *code bloat* these days. That's why they decided that all AngularJS code must be encapsulated as AngularJS modules. AngularJS modules can be defined in any order, include dependency modules that haven't

---

<sup>6</sup><https://github.com/angular/angular-seed>

<sup>7</sup><http://docs.angularjs.org/tutorial>

been defined yet, and still work properly. The only prerequisite is that the Angular.js core is loaded first.

Just like AMD loaders, AngularJS registers all module definitions before resolving dependencies in the module definitions. So there is no concern about the order that your AngularJS module dependency files are loaded into the browser, as long as, they are loaded after *angular.js* and before the *DOMContentLoaded* event when AngularJS normally bootstraps. This, of course, solves the problem of having to repeat blocks of code in order to satisfy the dependency on that code in different modules.

Repeated JavaScript code that finds its way into the browser or codebase repository is a pervasive problem in larger enterprise organizations where front-end development may take place among different groups spread out over continents, and the AngularJS module system can be quite helpful in avoiding or preventing this type of bloat.

### 2.1 AngularJS Asynchronous Module Definition and Retrieval

---

```
// Define an AngularJS module with dependencies referenced by "MyApp".  
// Dependencies can be loaded in any order before the "bootstrap" process and  
// AngularJS takes care of resolving in the correct order.  
angular.module('MyApp', ['MyServiceDeps', 'ThirdPartyPlugin', 'AnotherDep']);  
  
// A module with no dependencies is created like so  
angular.module('MyApp', []);  
  
// We can then retrieve the module for configuration like so:  
angular.module('MyApp');  
// Notice the only difference is the presence or absence of an array as the 2nd  
// argument.
```

---

## POJO Models

It amazes me how many UI developer job descriptions I see that require someone to be skilled in “object oriented” JavaScript when JavaScript was never really meant to be an “object oriented” language like Java or C# to begin with. It’s safe to say that most JavaScript frameworks out there have some sort of system for emulating Java style object inheritance hierarchies using JavaScript’s prototype object chains, and many try to emulate object encapsulation with unnecessary “getter” and “setter” functions that don’t really prevent direct access. This is another source of needless boilerplate bloat due to a general lack of understanding of the JavaScript language by traditional application engineers and engineering managers.

POJO is an acronym for plain old JavaScript meaning no getter/setter, or inheritance boilerplate- just functions and mix-ins. The AngularJS team’s definition of MVC uses just plain JavaScript objects for

its models, a conscious decision to again minimize bloat and complexity while isolating functionality for the sake of both testability and maintainability.

Below is a contrived comparison of a Backbone.js model to an AngularJS model. Backbone.js, like a lot of other frameworks, requires an inheritance hierarchy and encapsulates the model properties behind getters and setters to mimic the style and syntax of classical object oriented languages like Java. This is a natural result of engineers trained on Java, Ruby, or C++ who've later switched to JavaScript but haven't been able to fully embrace the different programming paradigms. JavaScript is terse. Most common functionality can be directly *mixed in* to the object that needs it. Objects can be instantiated with simple *object literal* syntax, and object attributes can be accessed and modified directly. For the sake of source code reduction, the AngularJS team has forgone all the unnecessary OO syntactic sugar.

## 2.2 Example of AngularJS and Backbone.js Models

---

```
// Backbone.js model
// step 1 - extend the base Model class
var MyModelConstructor = Backbone.Model.extend({});
// step 2 - create a new model instance
var myModelInstance = new MyModelConstructor();
// step 3 - add model attributes
myModelInstance.set({
    firstName: 'Dave',
    lastName: 'Shapiro',
    title: 'UI architect'
});
// step 4 - manually register attribute listeners with change callbacks
// to update the DOM
myModelInstance.on('change:title', function(model, title){
    // callback function for a jQuery style DOM update
});

/*****/

// AngularJS model
// no required boilerplate including set, get, or extend
// no event listeners, changes reflected in DOM automatically
// simple object literal notation
$scope.myModel = {
    firstName: 'Dave',
    lastName: 'Shapiro',
    title: 'UI architect'
};
```

---

## AngularJS and Other Frameworks in the Same Page

As mentioned before, all JavaScript frameworks are opinionated even if the opinion is to be un-opinionated. Some JavaScript frameworks are opinionated to the point of bordering on self-righteousness. The authors are so convinced that their way is the “right” way they cannot conceive as to why a web developer would need or want to make use of any other toolkit or framework. These frameworks typically allow only one instance to be attached to the root of the page in a monolithic, all or none fashion, and happily generate massive amounts of DOM elements with inline CSS in such a fashion that other tools cannot access. I won’t name any names here, but many web developers in larger enterprise organizations know who I’m talking about.

AngularJS, on the other hand, is both non-exclusive and non-intrusive when loaded in the same page as other frameworks. While it is often “bootstrapped” in declarative fashion with ng-app in the <html> or <body> tag when used as a single page app framework, the ng-app directive can also be placed in the DOM hierarchy at any level. Alternatively, AngularJS can be bootstrapped in the page imperatively at any level of the DOM, more than once, and at any point of the page life cycle without affecting code outside above its level in the DOM.

### 2.3 AngularJS Application Bootstrap Options

---

```
<!DOCTYPE html>
<html lang="en" xmlns="http://www.w3.org/1999/html" ng-app="DeclarativeApp">
<head>
  <meta charset="utf-8">
  <title>AngularJS Bootstrapping Demo</title>
</head>
<body>

  <!-- do this to run an AngularApp anywhere in the page -->
  <div id="imperative_app">
    ...
  </div>

  <script>
    // on DOM ready find a DOM node and start an AngularJS app
    angular.element(document).ready(function () {
      angular.bootstrap('#imperative_app', ["ImperativeApp"]);
    });
  </script>
</body>
```

---

This allows the developer the flexibility to use AngularJS as both a single page app (SPA) framework, or as a framework to support declarative UI components within a page that may or may not be using

another framework for SPA functionality such as Backbone.js. The latter is, of course, the subject of this book.

## Forward Compatibility with W3C Web Components and MDV

This is what makes AngularJS particularly attractive to this author. The W3C Web Component specifications when fully implemented by the major browsers will most certainly cause a major paradigm shift in the way web page construction is viewed.

Building websites and web apps will no longer be seen as a monolithic engineering endeavor. Creating a web app and adding functionality, will become more of a process of adding and extending off-the-shelf UI components represented as HTML markup. The Web Component specs describe how we will be able to encapsulate and scope JavaScript, CSS, with an HTML fragment and have it load, attach/detach and activate/deactivate at any point in the page lifecycle. Also, when `Object.observe()` is made available as the major browsers implement the next version of ECMA Script, model driven views (MDV) will become a basic part of any web page functionality.

### 2.4 Some of the new web components goodies

---

```
<!-- import html directly into a page -->
<link rel="import" href="include.html">

<!-- the new template tag -->
<template id="commentTemplate">
  <!-- innert DOM -->
  <style> /* scoped css rules */ </style>
  <div></div>
</template>

<!-- extend existing or define new elements -->
<element extends="button" name="big-button">
  <script>
    // custom element script API
  </script>
</element>

<script>
  // create a "light weight" iframe
  var s = element.createShadowRoot();
  var t = document.querySelector('#commentTemplate');
  r.appendChild(t.content.cloneNode(true));
```



```
// all object will have observe() methods to facilitate
// model driven views
Object.observe(callback);
</script>
```

---

However, in the snail's pace of web standards development, full Web Component and MDV support may still be a ways off, but AngularJS directives along with AngularJS data binding (the VM of MVVM) allow us to emulate this type of component encapsulation and functionality now. What's even better is that the AngularJS team will actively evolve the framework toward full Web Component and MDV compatibility and usage. UI component code written in AngularJS will likely be quite a bit more forward compatible with evolving web standards than components created using Extjs or jQuery UI. This translates to a longer shelf life and easier upgrade path for the code we write today. What UI developer is ok with their code becoming obsolete after a year or two?

## Summary

In this chapter we discussed some of the major attributes of AngularJS that are making it quite popular and widely adopted by the web development community including:

- dynamic views
- code reduction
- automatic data binding
- dependency injection
- TDD friendly architecture
- POJO as the framework's controllers
- non-intrusiveness or exclusiveness
- Web Component and MDV forward compatibility

In the next chapter we will explore the mechanics of AngularJS core directives, as well as rolling our own custom directives with a focus on component encapsulation.

# Chapter 3 - AngularJS Component Primer Part One: The View

The AngularJS.org landing page succinctly summarizes the framework as “HTML enhanced for web apps!” A directive is the AngularJS term for a unit of this *extended* HTML. Directives may encapsulate other directives and more than one directive may be active on an HTML element at a time. AngularJS has many built-in directives that form the nucleus of the declarative aspect of the framework.

The core directives cover the most common dynamic functionality for the view, but as we get beyond the core where we need to define our own dynamic view functionality, we do so by creating our own directives with a directive definition object.

## AngularJS Core Directives

The first place to learn about AngularJS’ core directives is the AngularJS documentation. Our discussion aims to expand on the documentation with a bent towards componentization. The API docs should be consulted for full API usage of any core directives discussed.

[<sup>8</sup>](http://docs.angularjs.org/api/ng)

However, there are several core directives that are of particular importance in understanding how the framework may be used to transform an HTML document from static to dynamic, and the core directives also fall into a few different categories. It is important to understand these as well. Our goal is to provide some insight to the core directives that is complimentary to the official documentation.

## Scoping Directives ng-app, ng-controller

### ng-app

An AngularJS application within an HTML document is scoped or demarcated with ng-app by placing this directive in the element that will serve as the root of the application. Most often this would be the root of the document, either `<html>` or `<body>` if we are creating a single page app. But it can be located further down the DOM hierarchy if in situations where we may want to run an AngularJS app within a single page app of another framework.

Example usage would be:

---

<sup>8</sup><http://docs.angularjs.org/api/ng>

```
<anyElement ng-app="myAppModule">
```

The dependencies for `ng-app` would be the core `angular.js` and a user defined AngularJS module that will serve as the application module.

Also, a reminder that for exact syntax and a simple usage example of any core directive, please refer to the API documentation at <http://docs.angularjs.org/api/><sup>9</sup>.

## ng-controller

AngularJS will automatically instantiate a default `$rootScope` for any declared application. But this is hardly useful to us. To give an application life, we need to define controller(s) for our application's business logic, and we need to “declare” the DOM scope of our controllers by placing the `ng-controller` directive in an element within our application boundary:

```
<anySubElement ng-controller="myAppController">
```

For both of the above to work, and to avoid the dreaded “`myAppModule` does not exist” error we need code in our HTML document that accomplishes the following:

### 3.0 Basic AngularJS app outline

---

```
<!doctype html>
<html ng-app="myAppModule">
  <head>
    <script src="angular.min.js"></script>
    <script>
      angular.module('myAppModule', [dep_1, dep_2, dep_3])
        .controller('myAppController', ['$scope', function($scope) {
          $scope.someData = 'Hello World!';
        }]);
    </script>
  </head>

  <body>
    <div ng-controller="myAppController">
      ...
    </div>
  </body>
</html>
```

---

The most common source of JavaScript console errors (or lack thereof) for AngularJS newbies is caused by forgetting to include either the declaration or matching definition for a module or controller.

---

<sup>9</sup><http://docs.angularjs.org/api/>

Every controller has an associated `$scope` object which as you may recall is the VM of the MVVM. Controller directives may be nested within other controller directives which is how we would go about creating a `$scope` hierarchy akin to the DOM hierarchy. We will discuss `$scope` in depth in chapter 4.

`ng-app` has one major drawback in that there can only be one of these per HTML document. So what happens if we have more than one AngularJS application that needs to reside in the same HTML document? Such may be the case if we need to include more than one unrelated AngularJS UI component of which each would technically be considered an “app” from the AngularJS point of view. Fortunately we can instantiate multiple AngularJS apps imperatively using:

```
angular.bootstrap(element[, modules]);
```

where `element` is a jQuery or jQlite reference to the element in the DOM. This code should only be called when all the necessary dependencies have loaded such as on `domReady` or `load` events.

## Event Listener Directives

Remember the days back in the 90’s, before jQuery when adding JavaScript to a page consisted primarily of something like this:

```
<a onclick="alert('hello world')" href="http://angularjs.org/">
```

If you don’t, then you are lucky. We had to stop doing that because around the time AJAX became the buzzword-of-the-day, it came to be considered “obtrusive JavaScript” since we were “mixing” presentation with behavior. So we had to maintain our JavaScript in separate files and reference an associated DOM element like so:

### 3.1 Unobtrusive JavaScript - the jQuery way

---

```
<!-- presentation.html -->
<a id="some_anchor" href="http://angularjs.org/">

/* behavior.js */
$('#some_anchor').click(function(){
    alert('hello world');
});
```

---

Now we have presentation and behavior cleanly separated, but as the *presentation* and *behavior* grew to thousands of lines of code, managing the points of association or bindings became very messy and loaded with boilerplate code. Even worse, it lead to JavaScript that was very difficult to unit test since the DOM reference became a *hard* external dependency.

The AngularJS team gave the two issues above some thought, and their solution was to try and find some middle ground while incorporating the advantages of both approaches. The result was

to essentially move the event listeners back into the DOM while still keeping the business logic separate. This effectively removes the binding boilerplate and management headaches.

So the core AngularJS module offers a number of directives to match, more or less, the old DOM attribute listeners. Currently, in version 1.2, the list includes:

ngBlur, ngChange, ngClick, ngCopy, ngCut, ngDbclick, ngFocus, ngKeydown, ngKeyPress, ngKeyUp, ngMouseDown, ngMouseenter, ngMouseleave, ngMouseMove, ngMouseover, ngMouseup, ngPaste, ngSubmit

Very similar to the old listener attributes, an AngularJS listener directive can be included as an element attribute and execute an AngularJS expression on the event.

### 3.2 The AngularJS way

---

```
<!-- view.html -->
<a ng-click="popup(message)" href="http://angularjs.org/">

/* controller.js */
function demoCtrl($scope, $window){
    $scope.message = "hello world";
    $scope.popup = function(msg){
        $window.alert(msg);
    };
}
```

---

“popup()” and “message” would map to a function and a variable defined on the current scope respectively. More on this in the next chapter. But suffice to say, we are NOT executing a function on the *global scope* as we would be with JavaScript event listeners.

Also please note that attribute directive names in the form of ng-\* are automatically camel-cased when mapped to the directive definition in our module JavaScript. Directives in the form of HTML attributes are the preferred way of inclusion, but there are other ways (comments, elements, data-\*) of including directives in markup explained in the AngularJS documentation.

## DOM & Style Manipulation Directives

Most of the remainder of the core directives are utilized to dynamically modify the DOM in one way or another.

ngShow and ngHide are the jQuery equivalents to .show() and .hide().

ngClass, ngClassEven, ngClassOdd, ngStyle are used to manipulate styling based on the state of the associated \$scope.

`ngIf`, `ngRepeat`, `ngSwitch` are used to add some conditional logic for template inclusion/exclusion in the DOM. It's a best practice to avoid logic in templates where possible. However, in cases where it makes the source code significantly DRYer, or it is not desirable to have asset URLs evaluate, then its warranted.

`ngInclude`, `ngBindHtml`, `ngBindTemplate` are used to include template HTML into the document.

## Form Element Directives

AngularJS provides a very rich set of directives for manipulating forms, form elements, and form data. Several of the directives are mapped directly to common form elements themselves to augment with AngularJS behavior like form, input, select, textarea.

`ngForm` allows for nested forms.

`ngModel` binds the value of any form element to a property on the scope.

AngularJS' form directives enable extensive declarative functionality including validations, filtering, and persistence. Two or three chapters could be devoted to form directives alone, but unfortunately are not in the \$scope (pun intended) of this book.

## Miscellaneous Directives

There are a number of core directives that have special purposes.

Some have been created to prevent the momentary flash of unstyled AngularJS text (FOUT) that can happen in the milliseconds between DOM paints and AngularJS evaluation including `ngHref`, `ngSrc`, `ngCloak` and `ngBind`. `ngBind` is used in place of the standard AngularJS expression delimiters `{{ }}`.

`ngInit` is used when it is desired to declaratively initialize scope variables. Care should be used in applying `ngInit` as it can be easy to allow *controller* code to bleed into the *view*.

`ngNonBindable` is used to prevent AngularJS from evaluating any expressions within the containing element. It can be thought of as applying commenting delimiters like `<!-- -->` would be used to prevent HTML evaluation and rendering. This comes in handy for things like code snippets in HTML text, or if you find yourself in certain, odd situations where there may be conflicts between AngularJS delimiters `{{ }}` in templates that also may serve as a Handlebars.js or html.twig templates since they use the same characters.

`ngTransclude` is a special directive that allows merging the content of a parent element content into a child directive. This one has particular relevance to building custom directives and will be discussed in depth in subsequent sections.

The list of AngularJS core directives is constantly growing and evolving, so please refer to the official documentation for the latest.

## Rolling Our Own Directives

Understanding the dynamic behavior that can be achieved through inclusion of AngularJS' core directives gives us a good basis for understanding how we can encapsulate our own custom behavior wrapped as a directive.

There are a number of important directive building blocks and life-cycle states that need to be fully understood before we can master the art of rolling our own directives. We will take a look at each of these from the perspective of our goal to be able to use AngularJS' directives as a tool for creating custom widgets or UI components. One of the requirements of component architecture is that the component should be able to exist independently of its containing environment and not need to know anything about it. We must draw this distinction because AngularJS directives have many other uses in providing custom behavior for web apps which do not require them to be entirely self-contained, and not understanding the distinction is where noobs often let outside dependencies bleed in.

For a full, general understanding of directive creation, which is beyond the scope of the book, I recommend first the section in the AngularJS docs on directives, and also chapters 8-9 in *Mastering Web Application Development with AngularJS* by Darwin and Kozlowski.

<http://docs.angularjs.org/guide/directive><sup>10</sup>

[http://docs.angularjs.org/api/ng.\\$compile](http://docs.angularjs.org/api/ng.$compile)<sup>11</sup>

<http://www.packtpub.com/angularjs-web-application-development/book><sup>12</sup>

## Directive Declaration

For our purposes, a minimal UI component directive definition that replaces a custom HTML tag in a page might be something like:

### 3.3 Minimal component directive

---

```
var myAppModule = angular.module('myApp', []);
myAppModule.directive('myHello', function() {
  return {
    template: '<div>Hello world!</div>',
    restrict: 'E',
    replace: true
  };
});
```

---

<sup>10</sup><http://docs.angularjs.org/guide/directive>

<sup>11</sup><http://docs.angularjs.org/api/ng.%5Cprotect%5Cchar%220024%5Crelaxcompile>

<sup>12</sup><http://www.packtpub.com/angularjs-web-application-development/book>

Including in pre-parsed, pre-compiled markup:

```
<html lang='en' ng-app='myApp'>
...
<body>
  <my-hello></ my-hello>
</body>
...
```

Viewing in AngularJS parsed and compiled markup:

```
<body>
  <div>Hello world!</div>
</body>
```

---

## Directive Naming

Our directive name is myHello. The my part is to provide a namespace for our widget should we want to include and distinguish it as part of widget library separate from others- especially the AngularJS directive library (ng). Name-spacing custom directive is considered a *best practice* in AngularJS development.

When the name myHello is registered with AngularJS, AngularJS will try to match that name when it parses the DOM looking for “my-hello” and performing an auto camel case between markup and code. By default, AngularJS searches for an HTML attribute such as <div my-hello>, but in our case, the line of code above, restrict:’E’, tells the AngularJS parser to search only for custom named elements <my-hello> rather than element attributes.

AngularJS actually gives four options for directive declaration as: elements, attributes, classes, and HTML comments denoted by E, A, C, and M respectively. The options provide for situations when supporting legacy browsers such as Internet Explorer 8 or where the HTML must be validated. Neither of these situations are particularly relevant for current HTML5 compliant browsers, and we want to be as *forward compatible* as possible with W3C Web Component specifications, so we will only be declaring our directives as HTML elements throughout this book.



The W3C spec for custom elements requires that custom element names include a hyphen in them. It’s good practice to include a hyphen in the names of any element directives defined with AngularJS 1.x for Web Component forward compatible code. The prefix before the hyphen should be your library’s three letter namespace.

## Directives Definition

The restrict:’E’ line above is part of a return statement which returns what’s called a directive definition object from our directive registration function. The definition object is where we configure



our directives to do all sorts of things including overriding certain configuration defaults.

As of AngularJS 1.2, the current directive configuration options include: **priority**, **restrict**, **template**, **templateUrl**, **replace**, **transclude**, **scope**, **controller**, **require**, **link** and **compile**. In order to be thoroughly fluent in rolling our own directives, we need to be intimately familiar with all of these configuration parameters including their options, defaults, and life-cycle order if pertinent. Again, for a comprehensive reference, please see the documentation links above since the discussion here is meant to expand upon that knowledge for our purposes.

### 3.4 Options and Defaults for the Directive Definition Object

---

```
myModule.directive('myHello', function factory(injectables) {
  var directiveDefinitionObj = {
    priority: 0,

    // HTML as string or a function that returns one
    template: '<div></div>',

    // or URL of a DOM fragment
    templateUrl: 'directive.html',
    replace: false,
    transclude: false,

    // directive as attribute name only
    restrict: 'A',
    scope: false,
    controller: function($scope, $element, $attrs, $transclude,
      otherDependencies) { ... },

    // string name of directive on same element
    require: 'directiveName',
    compile: function compile(tElement, tAttrs, transclude*) {
      return {
        pre: function preLink(scope, iElement, iAttrs, controller) { ... },
        post: function postLink(scope, iElement, iAttrs, controller) { ... }
      }

      // or
      // return function postLink( ... ) { ... }
    },

    // or
    link: {
      pre: function preLink(scope, iElement, iAttrs, controller) { ... },
```

```
    post: function postLink(scope, iElement, iAttrs, controller) { ... }
  },

  // or
  link: function postLink(scope, iElem, iAttrs, ctrl, transcludeFn)
    { ... }
};
return directiveDefinitionObj;
});
```

---

## template, templateUrl, and replace

In our example above we also included configuration values for `template` and `replace`. `template` tells our directive to use the string or function return value as an *inline* HTML template for our directive, and `replace` tells our directive to replace the parent element with the template rather than appending it. The `template` configuration is mutually exclusive with the `templateUrl` configuration. The latter is used to load an external template file, or include a previously loaded template as `<script>` tag. There is some debate as far as when to use which template inclusion style that is very pertinent to our goal of encapsulated UI components.

The motivation for loading an external DOM fragment includes:

- ease of maintenance and editing of HTML source code longer than 1-2 lines
- less cluttering of directive definition source code
- easier replacement

The motivations for including inline templates include:

- self-contained, encapsulated widget code analogous to how we might package Web Component code
- fewer files to manage
- better load performance than a separate external file
- being the only option for certain browser security restrictions such as loading cross-domain or cross protocol

For our purposes, the ideal situation would be a combination of the above. For development we maintain a separate template file. For production, we stringify the template source and inline it into the directive source via a build script prior to minification. We will explore a potential build script for this process in Chapter 7.

## AngularJS Templates are Real HTML, not Strings

A quick tangent on AngularJS templates is relevant. AngularJS templates, unlike other front-end template systems like Handlebars.js, are composed of real HTML fragments rather than text strings that undergo a compile and token replacement step. This is because AngularJS operates directly on the static HTML in a page, and the HTML in the page is the AngularJS template. For our purposes, what this means is that our template HTML must be an actual hierarchy of nodes including all opening and closing tags with a root node. Failures to include a closing tag, mixing tags, or including things like HTML comments or `<style>` tags before or after the root node in templates are very common cause of fatal errors for AngularJS as of version 1.2. Hopefully in future releases the AngularJS compiler will be more permissive, as it is a drawback for building fully self contained JavaScript, HTML, and CSS not to be able to include a `<style>` tag prior to and at the same level as the root node in the template.

### 3.5 Valid AngularJS HTML Template

---

```
<div class="single-root-element">
  <!-- a text node -->
  <h1>my template</h1>
  <div> {{ myContent }} </div>
</div>
```

---

### 3.6 Invalid AngularJS HTML Template

---

```
<!-- a text node -->
<h1>my template</h1>
<div> {{ myContent }} </div>
```

---



08/18/14 update - In AngularJS 1.2+ both templates above are now valid as the templating engine is more permissive towards the structure of the HTML fragment.

The final directive definition configuration in our previous example is `replace`. AngularJS' default is `false` (boolean value) meaning that any contents of our directive will be appended to the parent element rather than replacing it. This behavior is often desirable for directives that are part of AngularJS *applications*, however for packaged UI components and widgets, it is much cleaner if the parent element functions as a declaration and set of parameters to be entirely replaced by the processed directive. The only exception might be when we want to set up a live binding between a parent attribute and a directive scope value. However, this can usually be achieved in a cleaner fashion using event `$broadcast` and `$emit` to be discussed in the next chapter.

### 3.7 When replace equals true vs. false

---

```

<div class="myDirectiveTemplate">
  <span> some inner content </span>
</div>
...
<my-directive>
  <span> some outer content </span>
</my-directive>

<!-- when replace = true becomes: -->
<div class="myDirectiveTemplate">
  <span> some inner content </span>
</div>
...
<!-- when replace = false becomes: -->
<my-directive>
  <span> some outer content </span>
  <div class="myDirectiveTemplate">
    <span> some inner content </span>
  </div>
</my-directive>

```

---



08/18/14 update - For the purposes of Web Components and custom element forward compatibility, it probably is best now to stick with `replace == false` which more closely models how custom elements will work with `<template>` tags and shadow DOM.

## require, priority, and terminal

**require:** 'directiveName' **OR** **require:** [directive names]

is used to require that other directive(s) be present on the same element as ours. Prefix the “^” to the directive name to require the directive be present on any ancestor element. AngularJS apps may make extensive use of multiple directives on elements to ensure that any presentation or behavioral dependencies are present. When using AngularJS directives as a tool to create discreet UI components or widgets, especially those of a style congruent with Web Components, we really want the top level directive that represents our custom HTML non-dependent on other modules. That said, we are free to style any inner directives as we see fit since we may have the complexity of a miniature application nested within our component. By *Requiring* another directive, our directive is requesting methods and properties from the *required* directive’s controller instance be available

to its controller instance. This is analogous to the “mixin” way of reusing functionality in JavaScript, and a way that we can keep our source code DRY.

Required directives can be made optional if we prepend a “?” (“?” for searching parents) to the directive name, require: ‘?directiveName’. Otherwise AngularJS will throw an error if the required directive is missing from the element.

When we require a directiveName its controller instance becomes available as the fourth parameter on our link function. If we need to require multiple directives, then the fourth parameter would be an array of controller instances:

### 3.8 require Options for the Directive Definition Object

---

```
myModule.directive('myHello', function factory(injectables) {
  var directiveDefinitionObj = {

    // directive controller must be found on current element; throws error
    require: 'directiveName',

    // directive controller optional on current element; no error if missing
    require: '?directiveName',

    // directive controller must exist on any ancestor element; throws error
    require: '^directiveName',

    // directive controller optional on any ancestor element; no error
    require: '?^directiveName',

    // the controllers associated with the required directive(s) are injected
    link: function postLink(scope, iElem, iAttrs, controller/[controllers]) {
      ...
    }
  };
  return directiveDefinitionObj;
});
```

---

For UI component level directive definitions we should think twice about using the require option as it implies either hard or soft dependencies on the outside DOM. Dependent functionality should be injected as a service, and necessary configuration or data are better injected via attributes. At the time of this writing, if we want to be sure our UI components are forward compatible with the W3C Web Components specification, then we don’t want to have multiple directives residing on the same component element since that is not supported as a part of the custom element spec.

priority is a number used to specify execution order of multiple directives on an element. Higher numbers are applied earlier. The default is 0. As with require, we probably want to omit this for

our encapsulating directive while being free to use it for any inner directives. `terminal` (boolean) specifies that the current priority is the last to execute on an element.

The remaining five fields of the definition object deserve more extensive discussion and need their own sections.

## scope

The `scope` option is where we declare what, if any, parent scope data will be accessible to our directive component. It is also where we may declare any defaults for the data bound to our view. The scope will become for all intents and purposes, the ViewModel if we see our component as an MVVM implementation. It's the glue for binding our data together with our view.

Discussion of scope, data, and models are quite important so we will defer to and devote the entire next chapter to this topic. For now, just know that `scope: false` means use whatever scope object instance is the element level of our directive. `scope: true` says to create a new scope object instance for our directive but inherit the properties and methods of the parent scope to our directive element. `scope: {...}` says to create a new *isolate* scope object instance that does *not* inherit from any parent scopes, but still allows us to selectively import properties and methods from parent scopes. At the component level, we will almost always want to create an isolate scope since our components should not know about the world outside their boundaries,

## transclude

`transclude` is a word made up by the AngularJS team. Transclusion allows us to compile the DOM contents of the directive element and insert into the directive template on a node with the core directive `ngTransclude`. By default the contents are transferred as is, but AngularJS also gives us the option to perform some intermediate manipulation. The interesting thing about as is, is that this also includes the scope of the original DOM fragment along with the markup. The analogy is much like that of a closure in JavaScript where the return value retains access to scope values in the originating function.

The default value of `transclude` is `false`. If set to `true`, we can transfer the DOM fragment, original scope and all, into any `<element ng-transclude>` node in our directive template. The transcluded DOM and other nodes in the template may contain the same data-binding identifiers such as `{{name}}`, but they can hold different values as set on their respective scopes.

We may also set the value to `'element'`, in which case the entire directive element including any other directives on it, may be transcluded.

Since this book is focused on UI components the desire to inject DOM fragments may seem a bit puzzling. However, consider that there is no requirement that a UI component supply all of its own HTML in its template. Many UI components are simply just containers such as tab panels, headers, or footers. Suppose in a large organization with many micro-sites our goal is to propagate the official corporate look and feel. We can create a DOM container component with the look and feel, and

allow the site developer to import their contents, scope and all, via transclusion. If there is a need to perform any manipulation on the transcluded DOM, AngularJS passes a `$transclude` function as the fifth parameter to a directive's linking function. This can be useful for validating the passed in content, but care should be taken that we are not setting up a situation where our component is forced to make assumptions about what's passed in a way that it needs to know about the outside world.

Transclusion is one of the more advanced AngularJS concepts to grasp, so it is best explained by an example of a useful scenario below. However, suffice to say that transclusion can be a very powerful way to inject DOM fragment dependencies into our UI components.

### 3.9 transclude Example

---

```
// simplified for this example, never hard code URLs in your controller code
myApp.controller('myFooter', function($scope){
    $scope.faceBookUrl = 'https://www.facebook.com/dgs';
    $scope.googlePlusUrl = 'https://plus.google.com/10081026737705520/videos';
    $scope.twitterUrl = 'https://twitter.com/dgs';
    $scope.faceBookImg = 'https://cdn.dgs.com/img/fb.jpg';
    $scope.googlePlusImg = 'https://cdn.dgs.com/img/gp.jpg';
    $scope.twitterImg = 'https://cdn.dgs.com/img/twitter.jpg';
});

<!-- before compilation and transclusion-->
<footer ng-controller="myFooter">
    <!-- we have provided a social icon container component the page
    author can fill with links and images -->
    <social-links-container>
        <a href="{{ faceBookUrl }}">  </a>
        <a href="{{ googlePlusUrl }}">  </a>
        <a href="{{ twitterUrl }}">  </a>
    </social-links-container>
</footer>

myApp.directive('socialLinksContainer', function factory(){
    return {
        template: '<div ng-transclude id="social-container"' +
            '<class="corporate-styling"></div>',
        transclude: true,

        // replace the directive element
        replace: true,

        // isolate scope
```

```

scope: {},
controller: function($scope, $element, $attrs, $transclude){

    // this won't be applied to the transcluded template
    $scope.facebookUrl = 'http://facebook.com/dgs';

    // $transclude here is identical to transcludeFn below
},
link: function(scope, elem, attrs, ctrls, transcludeFn){
    transcludeFn([scope], function(clone){
        //clone is the transcluded contents we can manipulate
        //perform any processing on transcluded DOM
        //the optional scope parameter would be a replacement
        //for the default
        //transclusion scope which is the parent of the directive scope

    });
}
};
});

<!-- after compilation and transclusion -->
<footer ng-controller="myFooter">
    <div id="social-container" class="corporate-styling">

        <!-- notice from which scope the href value is applied -->
        <a href="https://www.facebook.com/dgs">
            
        </a>

        <a href="https://plus.google.com/100810267337987705520/videos">
            
        </a>

        <a href="https://twitter.com/dgs">
            
        </a>
    </div>
</footer>

```

---



## controller

The controller option of our directive definition object is a constructor function that gets instantiated before any pre-linking functions are executed. This is the place to attach any methods and properties to be shared among all instantiated directives of the type we are defining. In MVC terms, it would be the “C”.

### 3.10 Controller Option of a Directive Definition Object

---

```
controller: function('myHelloCtrl', ['$scope', '$element',
    '$attrs', '$transclude', 'otherDeps',
    function($scope, $element, $attrs, $transclude, otherDeps){
        ...
    }]),
```

---

Common injectables include the instance scope, element, attributes, and sometimes transcluded DOM. As with the other options for declaring controllers, the declaration syntax in our directive supports the “long hand” version which allows it to survive minification.

Naturally our controller function is where we would want to define any common business logic for all instances of our directive definition object. This is very important to distinguish from the directive’s linking function where any instance specific properties and methods should be defined especially where we may have many instantiated component directives in the same document. This is also where we would define any API methods and properties that may be required in by other directives. A very common mistake for developers unfamiliar with the directive lifecycle is to declare methods and properties here that are really intended to be instance specific.

## compile and link

These options are a rich source of confusion among AngularJS noobs. Understanding their differences and relationship really requires one to understand the basic architecture underlying AngularJS and how it performs its data-binding magic. Part of the confusion can be traced to all the different ways that compile and link syntax is displayed in examples of AngularJS code. Understanding the purpose and relationship of each, as well as, the *directive lifecycle* (explained in a subsequent section) is necessary to know which syntax to use.

You would specify either one or the other option in your directive definition. As with compiled software languages, naturally compiling happens before linking. In the AngularJS world, the *compilation* step during the birth of a directive instance is where any necessary manipulation of the template DOM prior to use would take place. Any manipulations or transformations would be common to all directive instances. While the need to manipulate a template before marrying with a scope and inserting in the DOM is not common, there are situation where we may need to use this step. Since AngularJS uses real HTML for its templates, any situation where the template may have

strings that need to be transformed into HTML elements first would be a use-case, as would any DOM cloning or conditional inclusion.

The compile function will always return a function or an object that will be used for the linking step where the template is cloned and married with its designated scope including any *instance* methods or properties. When returning a function, the function will execute during the post-link step. When returning an object it would include functions to execute during the pre-link step and the post-link step.

Which syntax is chosen should depend on whether or not there is a need to perform a task before any child elements or directives are processed. The pre-link function on a directive-element executes before the pre-link function of a child directive-element and so on until the bottom of the directive hierarchy is reached. Conversely, the post-link function of the lowest directive in any hierarchy will execute *before* the post-link function of any directive-element above it. For those familiar with the history of the event propagation model where event-capturing down the DOM hierarchy happens before event bubbling back to the root of the document the analogy here is quite similar. Just as we almost always process browser events during the *bubbling* phase, so do we usually perform any directive *instance* processing during the post-link phase. You can be sure that you are only affecting the instantiated directive-element when adding or manipulating any methods, properties or injectables in the post-link function.

### 3.11 Compile and Link Syntax Options for Directive Definitions

---

```
// use when logic is needed in the compile, pre, and post link phases
compile: function compile(tElement, tAttrs) {
    ... //compiler logic
    return {
        pre: function preLink(scope, iElement, iAttrs, ctrl, transcludeFn){...},
        post: function postLink(scope, iElement, iAttrs, ctrl, transcludeFn){...}
    }
},

// use when logic is needed in only the compile and post link steps
compile: function compile(tElement, tAttrs) {
    ... //compiler logic
    return function postLink(scope, iElement, iAttrs, ctrl, transcludeFn){...}
},

// use when logic is needed only during the pre and post link phases
link: {
    pre: function preLink(scope, iElement, iAttrs, ctrl, transcludeFn) {...},
    post: function postLink(scope, iElement, iAttrs, ctrl, transcludeFn) {...}
},
```

```
// use when we only need to perform logic in the post link step - most common
link: function postLink(scope, iElement, iAttrs, ctrl, transcludeFn) {...}
```

---

A side note on avoiding accidental memory leaks is to always make sure any event listeners within our directive-element are attached only in the post-link function, and always utilize the \$destroy event to cleanup any bindings on a directive-element to be removed.

You may notice in the AngularJS documentation that the injectable parameters for compile functions include tElement and tAttrs, and the injectable parameters for the link functions include iElement and iAttrs. The “t” denotes template, and the “i” denotes instance to help keep the difference of what’s actually being injected clear. Actually all the parameters to a link function are either instances or functions that operate on an instance. The \*Attrs parameters in both cases provide a normalized list of camel-cased attribute names to the function. Also, it is the order of the parameter injectables that is important, not the name.

## Dependency Injection (API) Strategies

Now that we have an overview of the AngularJS API for configuring directives let’s discuss how we can leverage these configuration options to create our own API’s for our directives that would act as reusable and portable UI components.

### Static Attributes

The first thing to consider is who the consumer of your component is going to be. Will they be a designer who has little if any knowledge of JavaScript and AngularJS, or will they be a senior front-end developer? If they are the former, then we likely want to create any configuration APIs as *declarative* and *plain English* as possible. This would essentially restrict us to utilizing *static* element attributes as the way to pass in any configuration information. A UI component directive would look something like:

```
<mySearchBox size="small" auto-complete="true"></mySearchBox>
```

In our directive definition we could access the attribute values in either the scope or link options:

```
scope: { size: @, autoComplete: @} //or
link: function(scope, elem, attrs){
  scope.size = attrs.size || 'small';
  scope.autoComplete = (attrs.autoComplete === 'true') ? true : false;
}
```

Pulling in attribute values in the link function allows for a bit more flexibility to set defaults or perform any processing.

## Dynamic Attributes, Functions & DOM Fragments

If our component customers are more JavaScript savvy we have more options in terms of how the interaction with our components can be set up. In addition to static attribute values, we could also permit scoped functions, AngularJS DOM fragments, and dynamic attribute values to be passed into our components. Along with the additional flexibility, we also need to be wary that we are not setting up any undesirable tight dependency couplings at run-time. Since we can inject live, two-way bindings including full DOM fragments, a best practice would be to make sure that our component can either function without the dependency or that a default is provided internally. A worst practice would be to rely on try-catch blocks for robustness, one that is epidemic in enterprise JavaScript.

Full DOM fragments can be included as innerHTML in our custom component element via transclusion:

```
<component>
  <inner-html></inner-html>
</component>
transclude: true,
<template-html ng-transclude>...</template-html>
```

Any HTML passed in, includes the original scope context. Alternatives for injecting any scope context as an API parameter include the “=” and “&” prefixes in the scope definition for two-way binding and scoped functions respectively. These are discussed further in the next chapter when we talk about scope.

We could also inject full AngularJS controllers via the require option, but then we have to ask whether or not we’ve designed our directive to be a discreet component to begin with.

If we happen to be including our AngularJS UI component inside of another application level framework such as Backbone.js with Handlebars.js for templating we can set directive attribute values dynamically at runtime as template tokens and then lazily instantiate our AngularJS UI component. This is a very advanced topic, but also the key to slowly transforming bloated jQuery based web applications to concise AngularJS apps from the inside out, with out the need to perform a full rewrite from scratch!

```
// this could be a Handlebars.js template
<mySearchBox size="{{small}}" auto-complete="{{auto}}"></mySearchBox>
```

What was mentioned in the last couple paragraphs will get *extensive* treatment by example in the next part of this title since these are core concepts to understand for building robust UI components and solving real world problems in existing web applications such as code bloat and excessive boilerplate.



08/18/2014 update - For AngularJS UI component source code that is web component ready, it's now best to try to limit input APIs to attributes and events, and to limit output APIs to events where possible. UI components that are developed with these restrictions will be much more cross-compatible with other web components when the UI components created with AngularJS are, themselves, upgraded to web components.

## The Directive Lifecycle

Understanding the steps in the lifecycle of a directive is important for knowing what can be manipulated when. We particularly want to avoid performing operations intended for a single directive instance on all instances by mistake.

Consider the run-time of a browser page, and assume all AngularJS bootstrap prerequisites have been loaded and included. When the initial DOM has been flowed and painted, the `onDomReady` event is fired and AngularJS begins the bootstrap process. After all modules are registered, AngularJS traverses the DOM under its control and matches any names to registered directives. When a name is matched, any associated HTML template is *compiled* into a template function. Keep in mind that the very first DOM traversal is actually a *template compilation* since AngularJS uses real HTML nodes (not strings) as templates.

If there are multiple directives attached to the same DOM node, the compiler sorts them by priority. Then the compile functions of matched directives are called making any compile step potentially a recursive operation. During the compilation step is where any necessary DOM manipulation takes place. As every compile function is run a linking function is returned which includes injectables for scope, element, attributes, any additional *required* controllers plus any developer added functionality.

Note that directive controller functions are constructors that are instantiated prior to any linking function execution. Therefore, any properties or methods in the constructor are copied to the instance and any prototype properties or methods are shared. As with basic JavaScript inheritance, any initial controller or scope properties should be included in the constructor. This is a good place to perform any general initialization. Methods including common functionality to all directives of the same type should be on the prototype object to avoid needless copies.

Linking functions are by default executed from the bottom of any controller hierarchy up, unless they are designated specifically as pre-link functions. Pre-link functions are executed top down and where any preprocessing should take place. When the link function is executed, the compiled template is bound with the associated scope instance where `$watches` and listeners are set up. The linking function is where we have access to the live scope and would perform any final initialization. Any AngularJS expressions in the linking function are equivalent to any included in an `ng-init` directive.

At this point, any AngularJS interactive app or component directives are ready for human interaction, and remain so until destroyed by a page refresh or client-side operation. AngularJS operations that destroy templates are usually pretty good about removing associated binding, but

not always. If heavy use of a custom directive results in a memory leak, then `$destroy()` (discussed later) needs to be applied manually.

This above description doesn't suffice as a complete, illustrative or definitive explanation of the directive life-cycle process. I recommend reading about directive compilation from multiple sources including the docs and general AngularJS books out there. The purpose here is to create awareness of what developer tasks should and can be performed at what part of the directive creation process.

## Testing Directives

If your goal in reading this book is to create AngularJS UI components that other developers will want to use, then comprehensive testing is a must, both unit and end-to-end. There are many approaches and frameworks for creating test cases. Jasmine for behavioral unit testing, Karma as a test runner, and Protractor for end-to-end are the preferred frameworks in the AngularJS community. Karma and Protractor were actually created by the AngularJS team.

There are also many good examples of creating unit tests in the tutorials, the best being those for the AngularJS ng-core directives at:

<https://github.com/angular/angular.js/tree/master/test/ng/directive/><sup>13</sup>

These should be used as the definitive set of unit testing examples.

We are not going to get into code examples for testing until the comprehensive examples in later sections. For now, the point should be made that if you create a pallet or library of UI components using AngularJS directives, the APIs for your components will be the contract between you and your customer. Your unit tests will be the means of verifying that contract during the lifetime of your library as you upgrade the internal source code.

For those unfamiliar with research methodology there are two important concepts with regard to the results that your tests provide:

- reliability
- validity

Validity refers to the degree that your test measures what it is supposed to measure. Reliability is the degree to which your test consistently measures from one run to the next. A lot of code shops require developers to provide unit tests along side their source code, and a lot of these tests are crap because they do not measure the primary piece of logic in the function. They measure something inconsequential and are written to always pass. These shops are often the source of excessively bloated and poor quality code to begin with.

If you do bother to provide tests with your source code, then make sure your tests are measuring the primary piece of logic in your function. It should paraphrase in code the documentation for the function.

---

<sup>13</sup><https://github.com/angular/angular.js/tree/master/test/ng/directive/>

Also, in the *ideal* world, we as developers have set requirements that we can create our TDD code against. The *reality* in many enterprise organizations is that a byproduct of a dysfunctional organization is developers being forced to code against fuzzy, changing requirements. If your product managers cannot deliver set specifications before your engineering manager starts cracking the coding whip, then test driven development is pointless since the logic to be tested will likely change right up until the QA process or even production. My two cents in this situation is to wait until the requirements are reasonably stable before writing tests, and consider any source code produced up until that time to be only of *prototype* quality.

## AngularJS 2.0, Web Components and Directive Re-classification

Earlier in this chapter we lumped the AngularJS 1.x.x core set of directives into a few categories. Then we previewed the many options that must be accounted for in defining our own directives. One of the biggest complaints among the AngularJS community is how complex defining a directive can become when we get to the level of complete UI components.

The core AngularJS team has heard these complaints and agrees that the extra time required to master directives is not ideal for making the best part of AngularJS as accessible as it should be.

## Summary

In this chapter, we discussed AngularJS directives, both off the shelf and custom. We also touched on specific ways that AngularJS directives can be defined to function as robust UI components of a quality that can be reused, exported, imported, published, etc.

In a web architecture sense, AngularJS directives encapsulate and represent the aspects of our application that relate to the View in an MVC, MVVM, or MVwhatever framework since AngularJS really has both controllers and view-models. Our view code, along with all of our code, should be decoupled of global and application level dependencies. If we are creating directives to serve as UI component encapsulation, then they should be created in such a way that no knowledge of the outside page or application is required. Likewise, they should be set up so that the containing page, app, or container can inject all the necessary configuration and data.

In the next chapter we discuss UI component architecture from the **Model** and **Controller** perspectives.

# Chapter 4 - AngularJS Component Primer Part 2: Models and Scope

In the last chapter we discussed directives, which are created by AngularJS developers to encapsulate *custom* HTML, custom extensions to HTML, and create *almost* fully self-contained UI components including presentation and behavior. We focused primarily on the presentation aspect of AngularJS. However, web applications and widgets are useless without the data they are built for. In this chapter we will discuss how data is represented, packaged and manipulated in the AngularJS universe with a focus towards loose coupling and componentization.

## Data Models in AngularJS

Developers new to AngularJS might be somewhat confused when the landing page of [angularjs.org](http://angularjs.org) mentions that AngularJS models are plain POJOs or plain old JavaScript objects, but then there is this thing called `$scope` which is the data that is bound to the HTML in AngularJS' data-binding. The distinction is that the last sentence was not entirely correct. It should be “which is what holds” the data that is bound to the HTML. To be clearer, the data in AngularJS is modeled in a basic JavaScript object, which may be converted to JSON and visa versa. That *model* is then attached to an AngularJS `$scope` object which provides a context for the data in the DOM hierarchy.

### 4.0 A model and a context in AngularJS

---

```
// Model is a plain old JavaScript object- no getters, setters, or inheritance
// It may start life as a JSON string from a REST call or from form field inputs
var aDataModel = {
  field_0: 'a string',
  field_1: 5
  field_2: false,
  field_3: {
    subField_0: ['an array']
  }
};

// $scope is a contextual container for a model
function ourController($scope){
  // Our data is now bound to the view, and so is another property and method
  // not a part of our data.
  $scope.data = aDataModel;
```



```
$scope.method_0 = function(){};  
$scope.property_0 = '';  
};
```

---

Models in AngularJS do not need to be part of any inheritance hierarchy or have OO style setters and getters, or other methods. On the other hand, scopes in AngularJS usually (but not always) are part of an inheritance hierarchy, and have a number of built-in methods and properties. Any developers with Backbone.js experience know that data *models* in that toolkit are inherited and have OO style encapsulation methods.

The reasoning behind using POJOs for models in AngularJS is for increased ease of testing, maintenance, and reusability. Tomorrow if we decide to switch to another framework or even *Web Components*, untangling the data will require minimal effort.

## Data Representation in the View

The `$scope` object in AngularJS is the container that allows both the *view* and the *controller* access to the data. When we attach our data object to a `$scope` object in AngularJS we have effectively created a *ViewModel* as most developers would define MVVM. The AngularJS team refer to AngularJS as an MVC framework perhaps for marketing purposes or for attractiveness to back-end developers, but the distinction between VM and C is fuzzy at best and probably depends more on whether it is used for web apps or components.

## Expressions and Bindings

In compiled template HTML, or the *view*, `$scope` data fields, methods, and properties are normally accessed and manipulated via AngularJS expressions. AngularJS expressions are snippets of JavaScript with a few distinctions. Expressions evaluate against the scope of the current element or directive, whereas, JavaScript expressions embedded in HTML always evaluate against the global (usually window) context. Attempting to evaluate a property that is null or undefined fails silently rather than throwing a reference error, and control flow logic (if, else, while, etc.) is not allowed since computational logic belongs in controllers, not templates.

#### 4.1 Difference between AngularJS and JavaScript expressions

---

```
// setting and accessing a scope context
<div id="expExample"
      ng-controller="ourController"
      ng-click="angularExpression()"
      onclick="globalExpression()">
  <span> {{ aScopeProperty }} </span>
  <input type="text" ng-bind="aScopeProperty"/>
</div>
```

---

The code above is short, but represents much of a developer's everyday use of AngularJS. We are applying a scope to `div#expExample`, and binding properties and methods to elements on or within the scope.

The default AngularJS binding delimiters are `{{}}`. They can be changed if there is conflict with other client or server template delimiters. `{{}}` is the read-only version of `ng-bind`. `{{}}` is typically used when displaying a scope property as output text in HTML. `ng-bind` is typically used to give two-way binding to form input elements, although it can also be used in cases where a delay in the JavaScript execution thread might cause an undesired momentary flash of the raw `"{{ var }}"`. Behind the scenes, the expressions are passed to the `$eval()` method on the `$scope` object analogous to JavaScript's `eval()` method with the exception that `$eval()` evaluates against the `$scope` context. It is a best-practice to keep statements for `$eval()` simple as in a property name or a function call that returns a value.

## \$scope creation and destruction

### Automatic Scopes

The concept of scope, scope hierarchies and other inter-scope relationships are a source of massive hair loss among AngularJS noobs. "Why does a bound property change when I do this, but not when I do that?" The answer is almost always because of an attempt to access the same property, but within different, unexpected scope contexts due to a scope popping up out of nowhere for some reason. So it is important to understand the various AngularJS activities that result in automatic scope creating- one of the downsides to a massive reduction in boilerplate.

So starting from the top, when an AngularJS app module is defined and attached to the DOM via `ng-app`, a root scope is automatically created on the `ng-app` element and is accessible from any sub scope via `$rootScope`. That said, my advice is to avoid at all costs directly referencing the `$rootScope` property from an inner scope as it leads to highly coupled dependencies. Accessing a `$rootScope` from a UI *component* directive scope should *never* be done.

In AngularJS single page apps, scopes are most often created when `ng-controller="myController"` is placed in an element declaration. Any data-bindings within that DOM fragment will reference that scope until an inner element or directive that instantiates a new scope is reached.

Scopes are automatically created when a new template is inserted into the DOM via `ng-route` or `ng-include`. Same goes for a form element when given a name attribute, `<form name="name">`, and also for any directive that clones HTML templates, a new scope is created for each cloned template when attached to the DOM as with `ng-repeat`.

## Manual Scope Creation

Behind the scenes, any scopes that are automatically created via one of the avenues above are done so with the `$new()` method, available on any existing `$scope` object. We may also purposely call `$scope.$new()` to create one manually, but the use cases are rare and is usually a clue that we are not doing something *the AngularJS way*.

Finally new scopes may be granted existence on purpose when a custom directive is instantiated, and we almost always want to create a new scope if our directive encapsulates a UI component. Recall the `scope` option in the directive definition object from the previous chapter. For the purpose of using directives to create re-usable UI components, we will focus closely on this type of scope creation.

## Scope Inheritance Options for Directives

When defining a directive for the purpose of encapsulating a UI component, we typically want to do a few special tasks with any new scope that is created. If we omit the `scope` option in our directive definition object, the default is no new scope, the same as if we declare `scope:false`. On the other hand, if we declare `scope:true`, a new scope is instantiated when the directive executes, but it inherits everything, prototypal style, from the parent scope. This is the default for automatic scope creation, and the reason for the hair pulling when the property names are the same, but the values are not. *Manipulating a value on a sub-scope creates a new variable overwriting the inherited value.*

The third value for the `scope` option is `scope:{}.` This creates an isolated scope for the directive instance with no inheritance from any parent scope. Because UI components should not *know* about state outside their boundaries, we typically want to use this option for our component directives. One side note with *isolate* scopes is that only one isolate scope can exist on an element, but this is usually not an issue with elements designated as component boundaries.

The value for the `scope` option does not need to be an empty object literal. We can include default properties, values and methods for the isolated scope using object literal notation. Also, the previous statement of no inheritance was not entirely correct. We can also *selectively* and purposely inherit specific properties from a parent or an ancestor scope, and we can also selectively accept values of the directive element attributes. Both of these abilities create some interesting options for defining APIs on our custom UI components.

## Isolate Scopes as Component APIs

While directives are a great tool for creating re-usable UI components, the primary focus of the framework is toward being comprehensive for web application functionality. Therefore, there is no

official AngularJS best-practice for defining UI component APIs beyond choosing a method that allows for loose coupling and dependency injection. The framework gives us multiple options for dependency injection, configuration, and API communication from the containing DOM. The option used should be the best fit for the use case and the user.

## Directive API as HTML Attributes

Suppose we have some epic user-story that requires we create a pallet of UI widgets and components that might be used by designers or junior developers to create websites or webapps. The consumers of our pallet might not have a high level of programming skill, so we would want to give them a method to include any configuration or data declaratively, or as close to plain English as possible.

For example, we could create a custom search bar directive with our organization's look-and-feel that a page author could include via adding a custom element:

```
<my-search-bar></my-search-bar>
```

But this is of limited use since search bars are often needed in different contexts, locations, and with certain configurations.

If we were using jQuery UI widgets, any customization or configuration would need to happen by passing in a configuration object to the imperative statement that creates the widget on an element:

```
$('#elementId').searchBar({autoComplete:true});
```

This limits the accessibility of widget use to developers with an adequate knowledge of JavaScript.

However, via AngularJS directives we could give the page author the ability to pass in configuration information declaratively via element attributes:

```
<my-search-bar auto-complete="true"></my-search-bar>
```

removing the need to touch any JavaScript. This is accomplished by configuring the isolate scope options in the directive definition object to use the values of matched attributes on our directive element.

### 4.2 Isolate scope attribute options

---

```
<my-search-bar auto-complete="true"></my-search-bar>
```

```
myModule.directive('mySearchBar', function factory(injectables) {
  var directiveDefinitionObj = {
    template: ...,
    restrict: 'E',
    scope: {
      autoComplete: '@', // matches an attribute of same name
      anotherModelName: '@autoComplete' // if using a different name
    },
  },
```

```
    controller: ...,
    link: function postLink( ... ) { ... }
  };
  return directiveDefinitionObj;
});
```

---

## Directive API as Selective Scope Inheritance

The downside of UI component APIs using static attribute values like the example above is the restriction that only string values may be passed in since that is what all HTML attribute values are, and the value is not a live binding.

If our use-case requires that the component consumer have the ability to pass in configuration information of all types and dynamically at run-time, then we more options available if the consumer is familiar with JavaScript and AngularJS at least somewhat.

By including a variable within AngularJS tags,

```
<my-search-bar auto-complete="{ { scopeVar } }"></my-search-bar>
```

as the value of an attribute we can create a live, read-only binding to a value on the parent scope. If `scopeVar` on the parent scope changes during the life of the directive instance, then so will the corresponding property on the directive scope.

Additional variable and method inheritance options for isolate scopes exist. Using `'=` instead of `'@` creates a live two-way binding between a value on a parent element scope and the directive scope, and using `'&` allows the scope to call a function within the parent element's context. Both of these options can be useful for inner directives, but we should avoid using these options as UI component APIs since they allow a component directive to directly affect state outside its boundaries. If the world outside a component's boundaries needs to react to some change in component state, its is much better to use the publish-subscribe pattern made available to AngularJS scopes via:

```
$scope.$emit(evtName, args) //propagates up the scope hierarchy
```

```
$scope.$broadcast(evtName, args) //propagates down the scope hierarchy
```

```
$scope.$on(evtName, listenerFn)
```

Any scope with the same `$rootScope` can communicate via events whether isolate or not. Note that while the publish-subscribe pattern is discouraged in favor of data-binding in the official AngularJS documentation, for maintaining loosely coupled UI components, it is necessary.

## Built In \$scope methods and properties

There are several methods included by AngularJS with every instantiated `$scope` object. They fall into a few different categories including:

- events for inter-scope communication
- data-binding and execution context facilitation
- programmatic creation and destruction

## Decoupled Communication

As mentioned above, AngularJS provide three scope methods for inter-scope communication: `$broadcast()`, `$emit()`, `$on()`.

`$on()` is an event listener function analogous to those provided by browser DOMs and jQuery. `$on()` takes two default arguments, a string representing an event name to listen for and an event handler function or reference to fire upon detection of the named event. The event handler function receives an AngularJS event object as its first parameter, and any parameters subsequently passed as additional arguments from `$emit` or `$broadcast`. The event object contains properties typical of JavaScript event objects such as `currentScope`, `targetScope`, `stopPropagation`, `preventDefault`, etc. See the docs at:

[http://docs.angularjs.org/api/ng.\\$rootScope.Scope](http://docs.angularjs.org/api/ng.$rootScope.Scope)<sup>14</sup>

...for the full list.

`$on()` allows any scope object in the page hierarchy to function as an event bus. The `ng-app` `$rootScope` and the root scopes of UI component directives are typical places to maintain an event bus. For our UI component directives we might wish to set up an event bus for any internal directives or controllers that need to communicate in some way when direct data-binding is not desirable or possible such as when inner directives are themselves UI components.

`$broadcast()` and `$emit()` are both used to *fire* events. Their arguments are identical, but their actions are in opposite directions. `$broadcast()` will propagate an event downwards through a scope hierarchy, and `$emit()` will propagate the event upwards.

The first argument to both is a custom event name (string). While any string may be used, it is advisable to follow what has become a JavaScript convention by including name-spacing with the event name separated by colons to help prevent any potential naming collisions.

```
$scope.$emit( 'myComponent:myController:eventName', args);
```

The any additional arguments to either `$broadcast()` or `$emit()` will be passed directly as parameters to any events handlers referenced in the matching `$on()` statements.

`$emit()` has special value when constructing directives that will function as UI components. Since it is considered very poor form to create UI components that can directly manipulate or influence any other component, page, or application state outside its boundaries. Publishing benign events that communicate a UI component's state outside of its boundaries that another component may listen for and optionally react to is an acceptable alternative. It does not require a UI component to have any specific knowledge of the outside world.

---

<sup>14</sup>[http://docs.angularjs.org/api/ng.\\$rootScope.Scope](http://docs.angularjs.org/api/ng.$rootScope.Scope)

In a sense, creating a UI component with a *published* set of events that it can emit can serve as a reverse API for a page application to react to any change in a component's internal state without needing to know what really goes on inside the component.

### 4.3 Component Scope Pub-Sub Communication

---

```
// contrived example of an application reacting to a state change of an
// inner UI component
```

```
<div ng-class="searchBarSize" ng-controller="myAppController">
  <my-search-bar compact="true"></my-search-bar>
</div>
```

```
myModule.directive('mySearchBar', function factory(injectables) {...
  }).controller('searchBarCtrl', function($scope){
    // a maximize button in our scope is toggled
    $scope.on('maximizeBtn:on', function(){
      // ask the outer application to make me larger
      $scope.$emit('mySearchBar:pleaseExpandMe');
    });
    // the same button is toggles again
    $scope.on('maximizeBtn:off', function(){
      // ask the outer application to make me smaller
      $scope.$emit('mySearchBar:pleaseContractMe');
    });
  });
});
```

```
myApp.controller('myAppController', function($scope){
  $scope.$on('mySearchBar:pleaseExpandMe', function(evtObj){
    $scope.searchBarSize = 'large-search-bar';
  });
  $scope.$on('mySearchBar: pleaseContractMe', function(evtObj){
    $scope.searchBarSize = 'small-search-bar';
  });
});
```

---

## Data Binding and Execution Context

`$apply()`, `$digest()`, `$watch()`, `$watchCollection()`, `$eval()`, and `$evalAsync()` are the built-in AngularJS `$scope` methods that perform the heavy lifting for the automatic data-binding which lies at the core of AngularJS' popularity. During the normal course AngularJS application development, these functions are usually called automatically by the built in directives and run in the background.

However, not everything fits neatly into the AngularJS universe, and there are instances when we will need to include these functions explicitly in our source or unit test code.

`$apply(non_angular_expression)` is used to wrap a function with code from outside the AngularJS context to execute within the AngularJS lifecycle and context. `$apply` ends up being utilized quite often when importing functionality from other toolkits and frameworks, browser DOM event callbacks, and window functions such as `setInterval`, `setTimeout`, XHR calls, etc. Behind the scenes, `$apply()` calls `$eval()` and `$digest()` to provide AngularJS context and lifecycle. It's not always clear up front when some code should be wrapped in `$apply()`, but if there is a scope value that is not being updated *when* it should be or at all then an `$apply()` wrapper is likely needed. A typical scenario where `$apply()` is needed is if we use an external AJAX library and need to set a scope value from within the the `onSuccess` or `onError` callback function.

#### 4.4 Explicit `$apply()` wrapper needed

---

```
// scenario where $apply must be called explicitly
myModule.directive('mySearchBar', function factory() {...
    }).controller('searchBarCtrl', function($scope, $extAjaxLib){
        $scope.ajaxCall = function(ajaxObj){
            $extAjaxLib.send({ajaxObj}, function successCallback(data){
                //this does not update $scope as expected
                $scope.data = data;
                //this does update as expected
                $scope.$apply(function(){
                    $scope.data = data;
                });
            });
        });
    });
});
```

---

Because AngularJS hasn't been around long enough to benefit from a large library of third party plugins at the same level as something like jQuery, we will often want to wrap third party plugins from other toolkits rather than recoding the needed functionality from scratch, and `$apply()` will need to be applied (pun intended) quite often in this case.

Besides `$apply()`, `$watch(watchExpression, listener, objectEquality)` is another built-in scope function that sometimes needs to be called explicitly, though not nearly as often. `$watch()` is used to register a listener callback function that fires when a `$scope` expression changes. A typical use-case for explicit use of `$watch` is when we wish to run some code in response to a change in a `$scope` value.

`$watchCollection(object, listener)` performs the same function as `$watch()` with the exception that it watches at the first level properties (shallow) of an object for any changes.



Extreme care must be exercised when explicitly calling `$watch()` that the listener function is able to complete quickly. It needs to be efficient, and avoid any long-running operations such as DOM access since it will likely be called *many* times during a single `$digest()` cycle.

`$digest()` is the scope function that kicks off AngularJS' system of recursive *dirty-checking* of all scope properties that have registered `$watch`'s. `$watch` listeners execute and "watched" properties are rechecked until all "watched" properties on the scope have stabilized. There are actually about 30 pages of important AngularJS information in the previous two sentences. A complete discussion of everything that happens during the `$digest` cycle while very important to understand, is unfortunately beyond the scope of this title. The AngularJS developer's guide (<http://docs.angularjs.org/guide/scope>) gives a good overview of all you should know about the internals of the `$digest` loop. Suffice to say that you should not need to ever call `$digest()` directly except perhaps as part of some unit test code. You might, however, want to detect that a `$digest` loop has occurred which can be done via a `$watch` listener function.

On a side note, it will be quite welcome when the ECMA Script 6 `Object.observe()` is finally implemented by all major browsers. `Object.observe()` will obsolete the need for `$watches` and dirty-checking in AngularJS removing the major performance issue inherent in AngularJS' data-binding system. Experimental versions of Chrome with `Object.observe()` enabled show digest loop performance increase by a factor of 40-60.

`$eval(angularExpression)` and `$evalAsync(angularExpression)` are used to explicitly evaluate AngularJS expressions. The latter uses the `setTimeout(0)` trick to defer the run till the end of the current JavaScript execution queue. They are called automatically when `$apply()` runs and usually are only needed explicitly in test case code.

## Creation and Destruction

`$new(isolate)`, and `$destroy()` are scope methods that are used to explicitly create and remove scopes. We rarely need to use these explicitly in our source code as they are called automatically whenever we insert or remove compiled templates in the DOM via built-in directives or client-side routing. There may be times when we do need these functions while writing test code. Also, do note that `$destroy()` removes all child scopes, data and bindings to prevent memory leaks similar to jQuery's `remove()` function, and it also *broadcasts* a `$destroy` event that may be useful for performing any additional cleanup tasks such as removing non-AngularJS bindings on the DOM hierarchy to be removed.

## Debugging Scopes and Testing Controllers

As new `$scope` instances are created in every AngularJS app, they are given an `$id` property that is numbered in ascending sequence as the scope begins life. We can then inspect each scope in the app or component hierarchy referenced by `$id` with AngularJS Batarang for Chrome which can be got at the Chrome Web Store.

Batarang adds some quite useful extensions to the Chrome Developer Tools. The most useful is the ability to inspect the AngularJS scope properties for any element in the DOM the same as we can for styling, event listeners, and DOM properties. We can also inspect the hierarchy of scopes in the AngularJS app or component. Both of these are very helpful in uncovering where those *mystery scopes* came from or where those scope values went since AngularJS does a lot of automatic scope creation under the covers especially when it comes to forms, includes, etc.

Approaches to testing controllers vary widely, and it is beyond our *scope* to present a comprehensive discussion of testing. The AngularJS developer guide, tutorial, and recommended technical books for AngularJS application development cover controller testing thoroughly. However, there are some key points to remember.

First, it cannot be stated enough that any and all external dependencies must be injected into any controller function as parameters. Never reference anything in the DOM directly such as class or id names. The same goes for all global functions and properties. All of these either have a preexisting AngularJS wrapper such as `$location`, `$window`, `$document`, `$log`, or it is a simple matter to create your own wrapper as an AngularJS service. Ideally we want all of our functions in our AngularJS source code to exhibit the highest degree of *functional transparency* as possible.

Second, it is highly recommended that controller tests inject both `$rootScope`, and `$controller` in order to manually instantiate a new scope using `$new()` and manually instantiate a new controller instance immediately after.

## Summary

In the previous two chapters we discussed using the AngularJS framework to create and interact with the view via built-in and custom directives (chapter 3), and to interact with and enhance the model (chapter 4). Along the way we discussed strategies and best-practices for constructing UI component APIs and interfaces through creative use of AngularJS built-in binding and injection schemes.

We learned that we can create UI component APIs using element attributes, transclusion, and event publish and subscribe that can allow our components to exist and function without any direct knowledge of any outside DOM.

In the next section we will explore some examples of real-world user stories where UI components created with AngularJS can solve enterprise level problems such as excessive source code bloat, lack of consistent look and feel, and inability to share, export and reuse UI code across a large organization.

# 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.

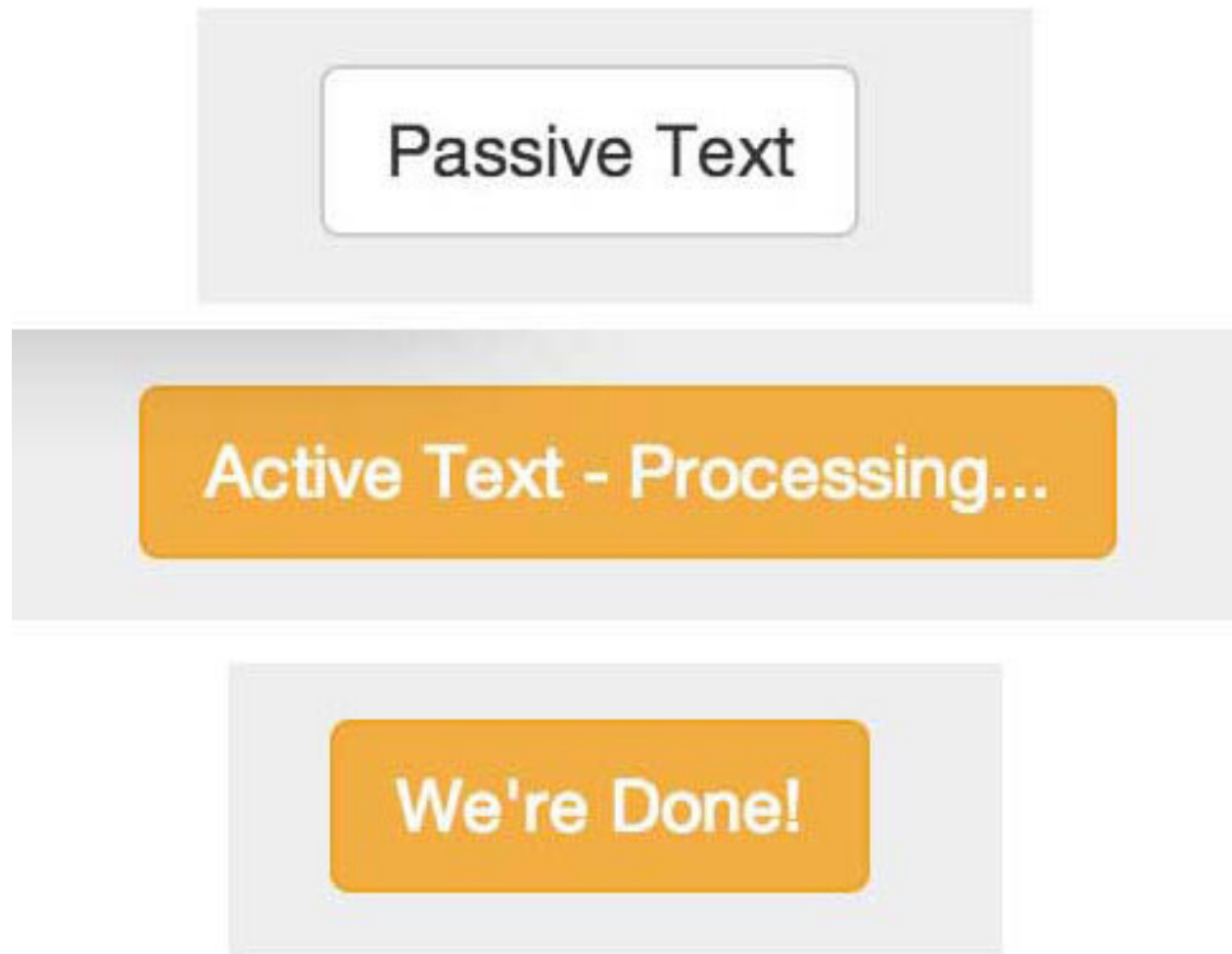
A proprietary framework often used for the above is ExtJS. Underneath ExtJS is a lot of monolithic JavaScript and DOM rewriting. It works great if used as is out of the box, but breaks down severely when styling or behavior modifications are needed. It also has serious performance problems and does not 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="btnClass">{{defaultText}}</a>';
        return {
            // use an inline template for increased
            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.btnClass = '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.

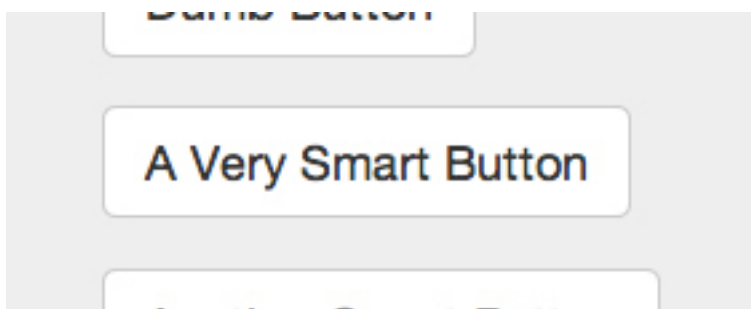
Our smart button component really isn't all that smart just yet, let's start building our API to add some usefulness for the consumers of our component.

### 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="btnClass" '
      + 'ng-click="doSomething()">{{btnText}}</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.btnClass = 'btn btn-default';
        $scope.btnText = $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.btnText = $scope.activeText;
          $timeout(function(){
            $scope.btnText = "We're Done!";
          }, 3000);
        };
      },

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

```

```

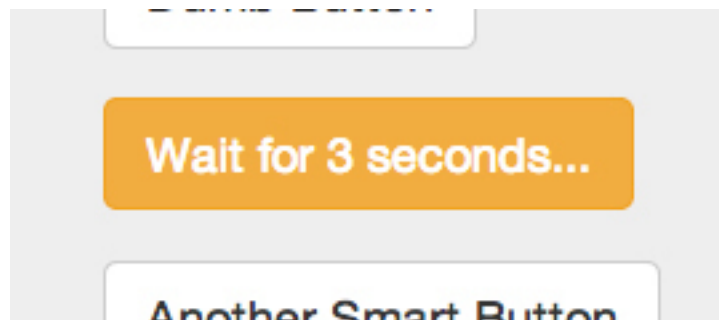
        if(iAttrs.defaultText){
            scope.btnText = scope.defaultText = iAttrs.defaultText;
        }
        // <string> button text to display when active
        if(iAttrs.activeText){
            scope.activeText = iAttrs.activeText;
        }
    }
};
});
})();

```

---

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 `setTimeout()`, 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="btnClass" '
            + 'ng-click="doSomething(this)">{{btnText}}</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.btnClass = 'btn btn-default';
    $scope.btnText = $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.btnText = $scope.activeText;
      $timeout(function(){
        $scope.btnText = "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.btnClass = 'btn ' + command.setClass;
        }
      }
    });
  },

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

```

```

        scope.btnText = scope.defaultText = iAttrs.defaultText;
    }
    // <string> button text to display 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 AngularJS 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="btnClass" '
            + 'ng-click="doSomething(this);debug()">{{btnText}}</a>';

        return {
            template: tpl, // use an inline template for increased
            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.btnClass = 'btn btn-default';
                $scope.btnText = $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.btnText = $scope.activeText;
                    $timeout(function(){
                        $scope.btnText = "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.btnClass = 'btn ' + command.setClass;
                }
            }
        });
    },

    link: function(scope, iElement, iAttrs, controller){
        // <string> button text
        if(iAttrs.defaultText){
            scope.btnText = scope.defaultText = iAttrs.defaultText;
        }
        // <string> button text to display 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

---

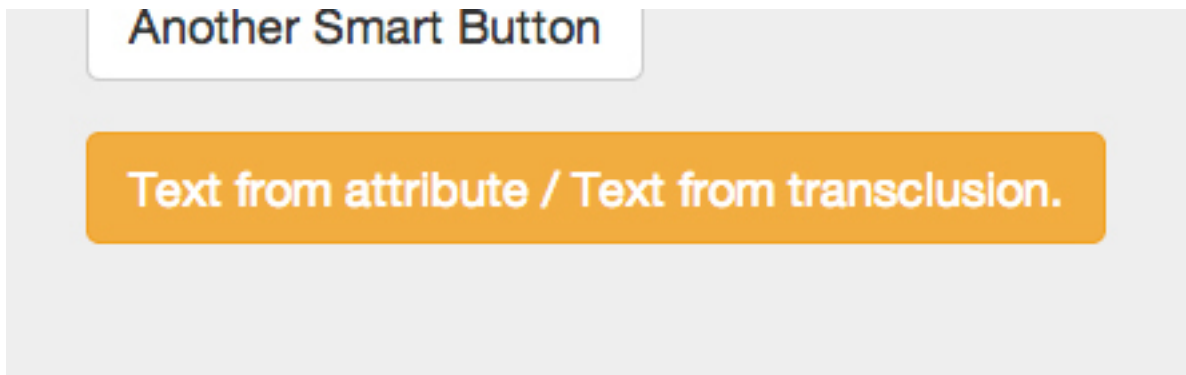
```
<!-- index.html -->
<smart-button
  default-text="Text from attribute"
  active-text="Wait for 5 seconds..."
>
  Text from transclusion.
</smart-button>

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

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

    return {
      template: tpl, // use an inline template for increased
      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'	<code>\$emit()</code>	ngClick	element
'smart-button-command'	<code>\$on()</code>		

## 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.

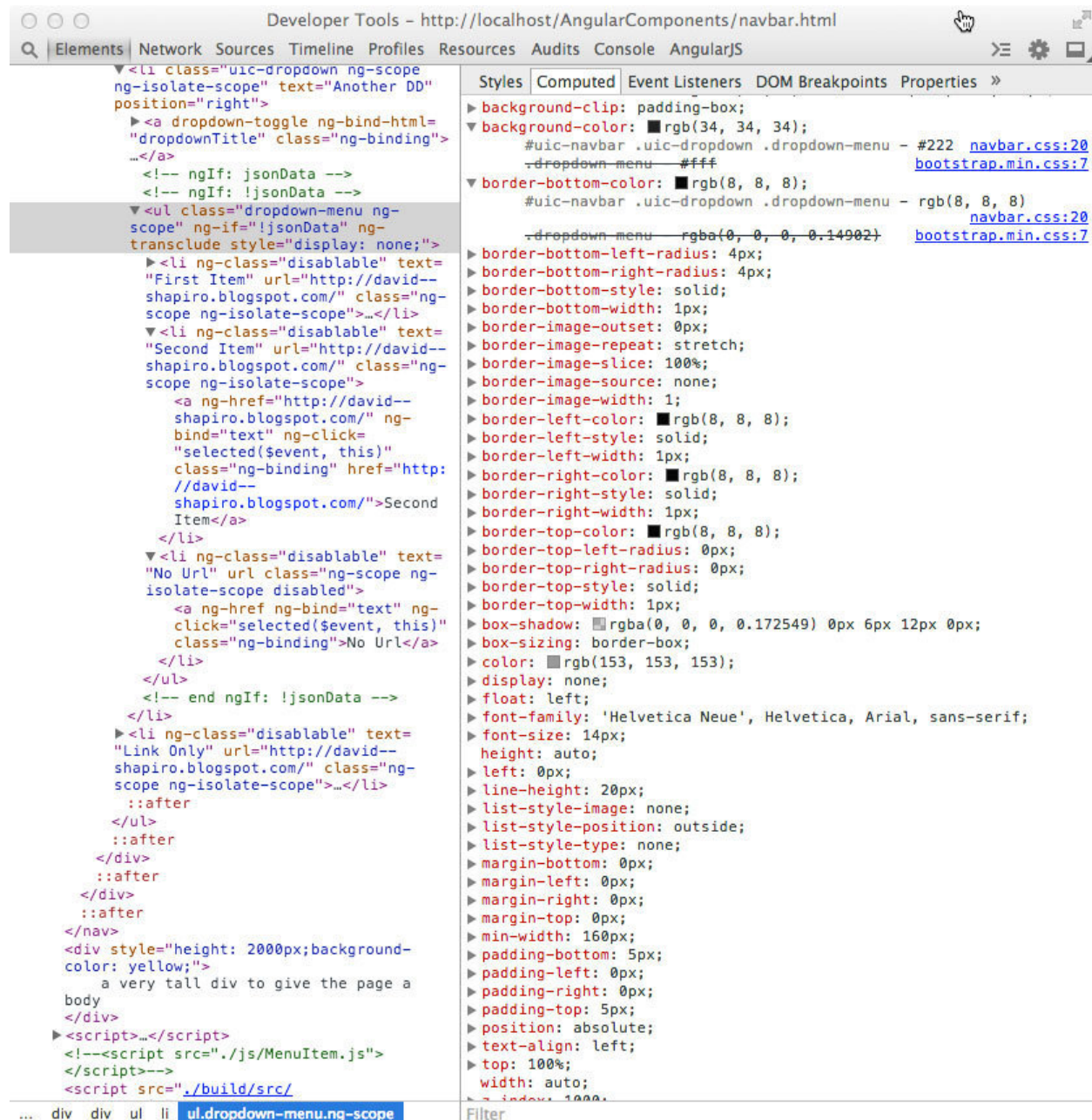
## Style Encapsulation Strategies

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( $injectorDependencyFn ){
    $scope.aScopedFunctionRef= $injectorDependencyFn;
});
```

## 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/><sup>15</sup>

<http://karma-runner.github.io/><sup>16</sup>

<https://github.com/angular/protractor><sup>17</sup>

<https://code.google.com/p/selenium/wiki/WebDriverJs><sup>18</sup>

<http://phantomjs.org/><sup>19</sup>

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

---

<sup>15</sup><http://jasmine.github.io/>

<sup>16</sup><http://karma-runner.github.io/>

<sup>17</sup><https://github.com/angular/protractor>

<sup>18</sup><https://code.google.com/p/selenium/wiki/WebDriverJs>

<sup>19</sup><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><sup>20</sup>

<https://github.com/yeoman/generator-angular><sup>21</sup>

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

<https://github.com/angular/angular-seed><sup>22</sup>

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 in-lined 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_) {
```

---

<sup>20</sup><https://github.com/yeoman/generator-karma>

<sup>21</sup><https://github.com/yeoman/generator-angular>

<sup>22</sup><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\')"'
    + '>{{btnText}} 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 btn text',
      function(){
        expect(element).toContain('A Very Smart Button');
      });
  });

```

```

// API: activeText
it('should display the value of "active-text" when clicked',
function(){
    expect(scope.btnText).toBe('A Very Smart Button');
    scope.doSomething();
    expect(scope.btnText).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).toHaveBeenCalled('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.btnClass).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, then 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 6 - UI Container Components by Example

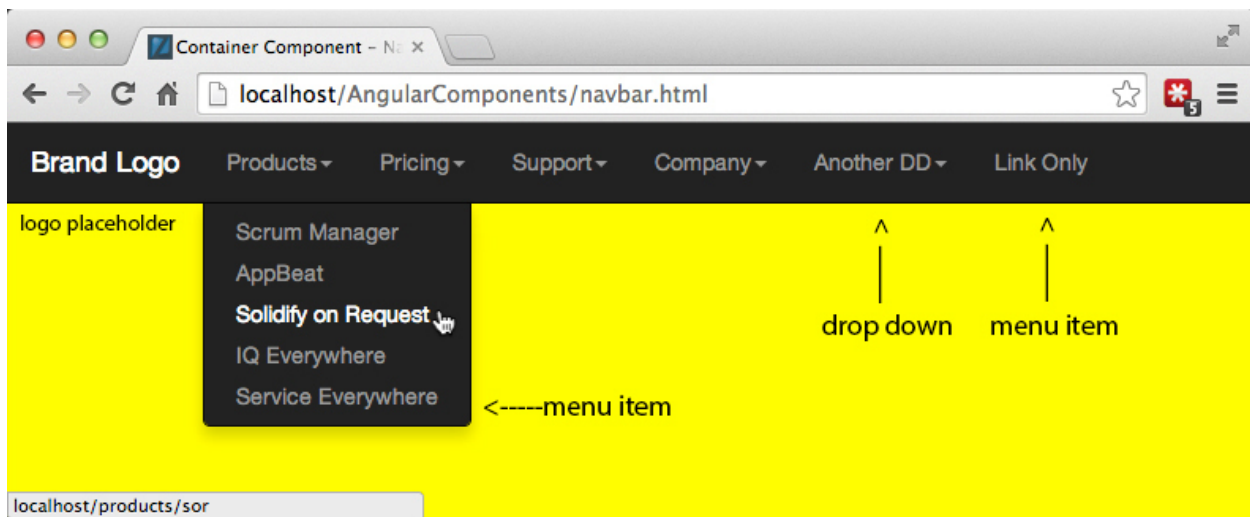
In the first half of this book we've extensively discussed the attributes of a "high-quality" UI component. It should be reusable, portable, encapsulated, and have no direct outside dependencies. In other words, *a good UI component should not know about anything beyond its element boundaries*. Using AngularJS as our toolbox, we built an example component that adheres to the best practices discussed, and learned a lot about the AngularJS APIs and the declarative approach in the process.

However, this only gets us so far in the real world. The best selling book of all time has a saying, "...give a man a fish, and he eats for a day. Teach a man to fish, and he eats for a lifetime". As is always the case with hot new web technologies, there's the rush to publish books about it, and these books always end up "giving the man a fish" rather than teaching him *or her* to fish. Our goal is to teach the sport of fishing, and focus on the component development concepts that can be applied to situations well beyond the set of examples we cover in this chapter.

A good percentage of UI components and widgets are meant to be part of a group such as tab panels, carousel slides, and menu items. These components need to exist side by side with some kind of common containment and management; otherwise we end up in a situation where we are trying to herd cats on the web page.

This is where the use-case requires a container component such as a tab group, slide deck, or menu bar. All the major client side widget toolkits including jQueryUI, Extjs, Bootstrap.js, Dojo and more have container components. Often the behavior on the inner components are such that when clicking on one component to toggle the active state means that the other components must be toggled to a passive or disabled state such as hiding a tab panel when another tab is clicked.





Nav Bar, Dropdown, &amp; Menu Item Components

Container components allow us to provide and ensure common styling and behavior at the next level up in the DOM hierarchy from a discreet UI component. When creating UI components in AngularJS that are meant to exist as part of a group, populating a container component with them allows us to use the container component as the central event bus for the individual components. This way AngularJS UI components can have minimal or no dependency on the `$rootScope` as the mediator since `$rootScope` is analogous to the window scope in the DOM.



**Leave `$rootScope` alone** Why do we want to minimize component dependency on the `$rootScope`? After all, there are many examples on the web in tech blogs, Stack Overflow or GitHub that show you how to attach event listeners, data, or even access `$rootScope` in view templates. But for precisely this reason, and the fact that AngularJS is rapidly becoming the most popular JavaScript toolkit since jQuery, is why we want to avoid coupling the proper functioning of our UI components with the `$rootScope`. Who knows who or what might be messing with it, especially given that most web developers have too much to do in too little time, and using the `$rootScope` rather than figuring out the proper way, is the path of least resistance. Note that while we used `$rootScope` for the button component example, this was purely to illustrate how to decouple dependency via the publish-subscribe pattern using AngularJS event listeners and handlers. In the real world, we would prefer attaching event listeners to the appropriate scope such as an application or component container scope.

<< diagram of container scope or service as event bus vs. `$rootScope` >>

Container components can and should know exactly what exists and is happening within their element boundaries including full knowledge of their child components. However, as with any high quality component, the opposite is not the case. Container components, just like any other UI components should have zero knowledge of anything that exists at a higher level of the DOM or AngularJS controller hierarchy.

<< diagram ??? >>

This chapter focuses on six goals:

1. Building robust UI component directives meant to exist along side instances of the same
2. Building container components for the above
3. Population of container components with child UI components
4. Interaction between container components and children components
5. Which AngularJS tools, methods, and conventions should be used for what and when
6. Understanding *why* we choose to make an implementation a certain way compared to all of the alternative ways of accomplishing the same result. Every function and line of code in the examples to follow was written to adhere to a best practice. Nothing is arbitrary.

This is the chapter where everything we've discussed in previous chapters comes together. We will stretch the limits of the tools AngularJS provides us with as we create a global navigation header container. Our header will be fillable with either dropdown menus or individual menu item components. The dropdown menus will also be fillable with the same menu item components making them both container and contained components. In keeping with the DRY principal, the dropdowns and menu items will have the ability to exist outside of the container components we create for this example. All of the components we create will have both declarative and imperative methods of creation, rich and flexible APIs, and to top things off, rather than reinvent the wheel, we will borrow and extend existing functionality from the Bootstrap, AngularJS core, and Angular-UI projects.

Does this sound overwhelming? It should, so we will start with the most basic pieces of any web application- the business requirements and the data, and proceed in a logical fashion towards the more complex inter-component interactions.

## Defining a Set of Navigation Components

As in the previous chapter's example, we will again imagine that we are tasked with creating a set of container and contained components that would be suitable for use by many different teams throughout a large, enterprise organization.

### Basic Requirements

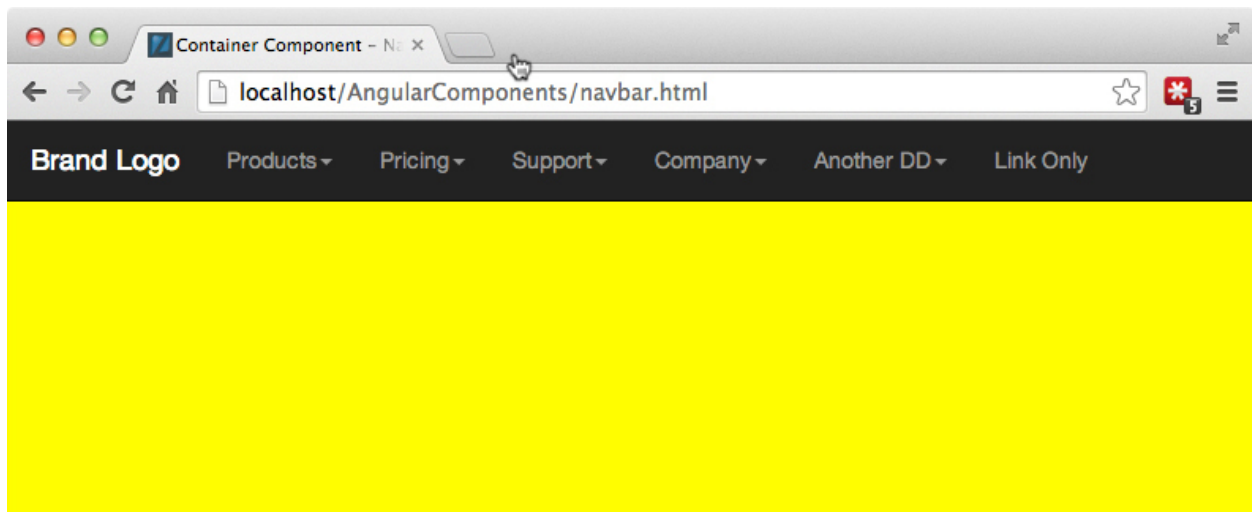
- Must be viewport responsive - same code for desktop and mobile
- Must allow dynamic or runtime creation of menu labels, URLs, etc.
- Must be able to dynamically switch between minimal and maximal display states
- Have active and passive states for dropdowns and menu items
- Must allow for limited set of styling or color choices for the consumer
- Able to be included with the simplest HTML markup as possible, or rather only a limited knowledge of HTML is required to create a full featured navigation header
- Full set of API documentation and full unit test coverage

## Nice-To-Haves

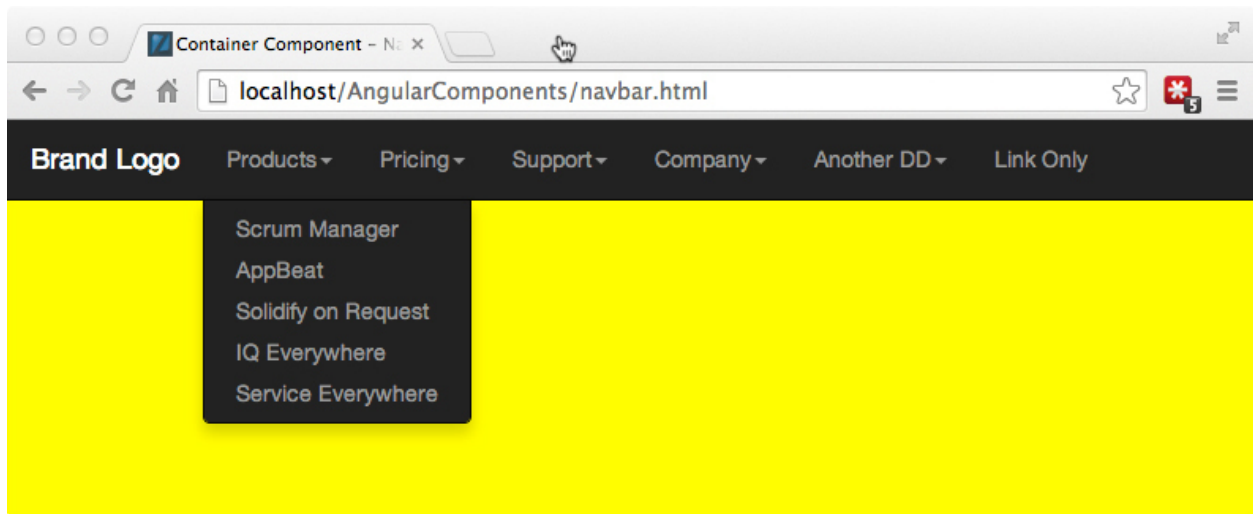
- For DRY purposes, UI components should be viable whether in or outside of a container component
- Ability to populate the Navigation Bar both declaratively with markup, *and* imperatively with JSON data in the same instance.
- Source code organized in a way that is easily upgradable to web components

## Design Comps

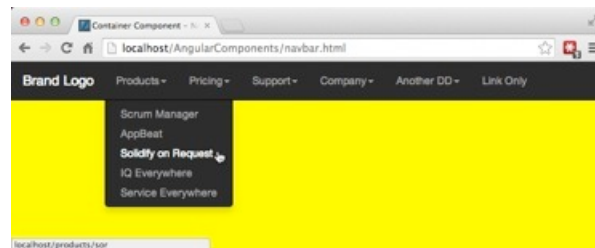
In most organizations product or project managers will collaborate with UX (user experience) and visual designers to create wireframes and example images of how the engineering deliverables should like and behave. The following are the visuals from our imaginary design team:



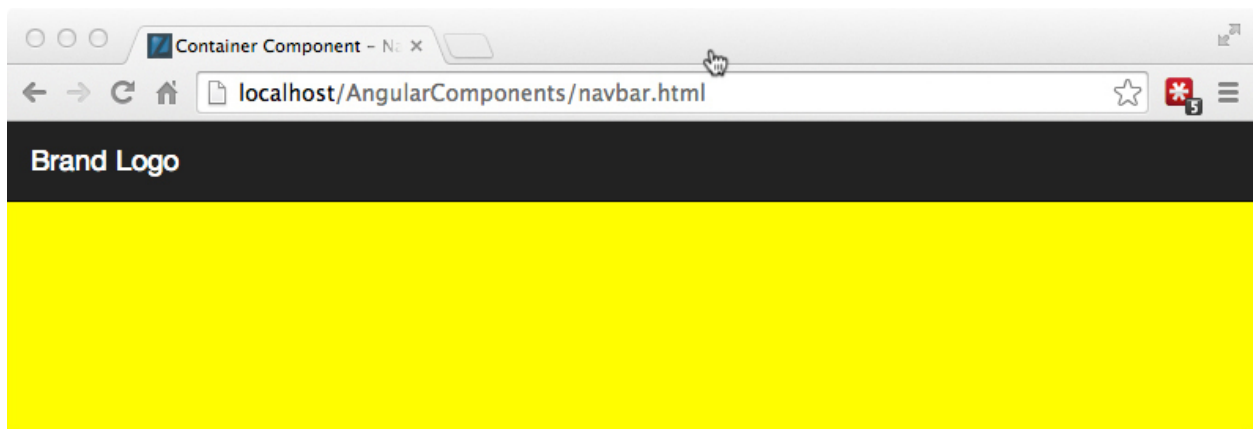
Navigation Bar - Default State



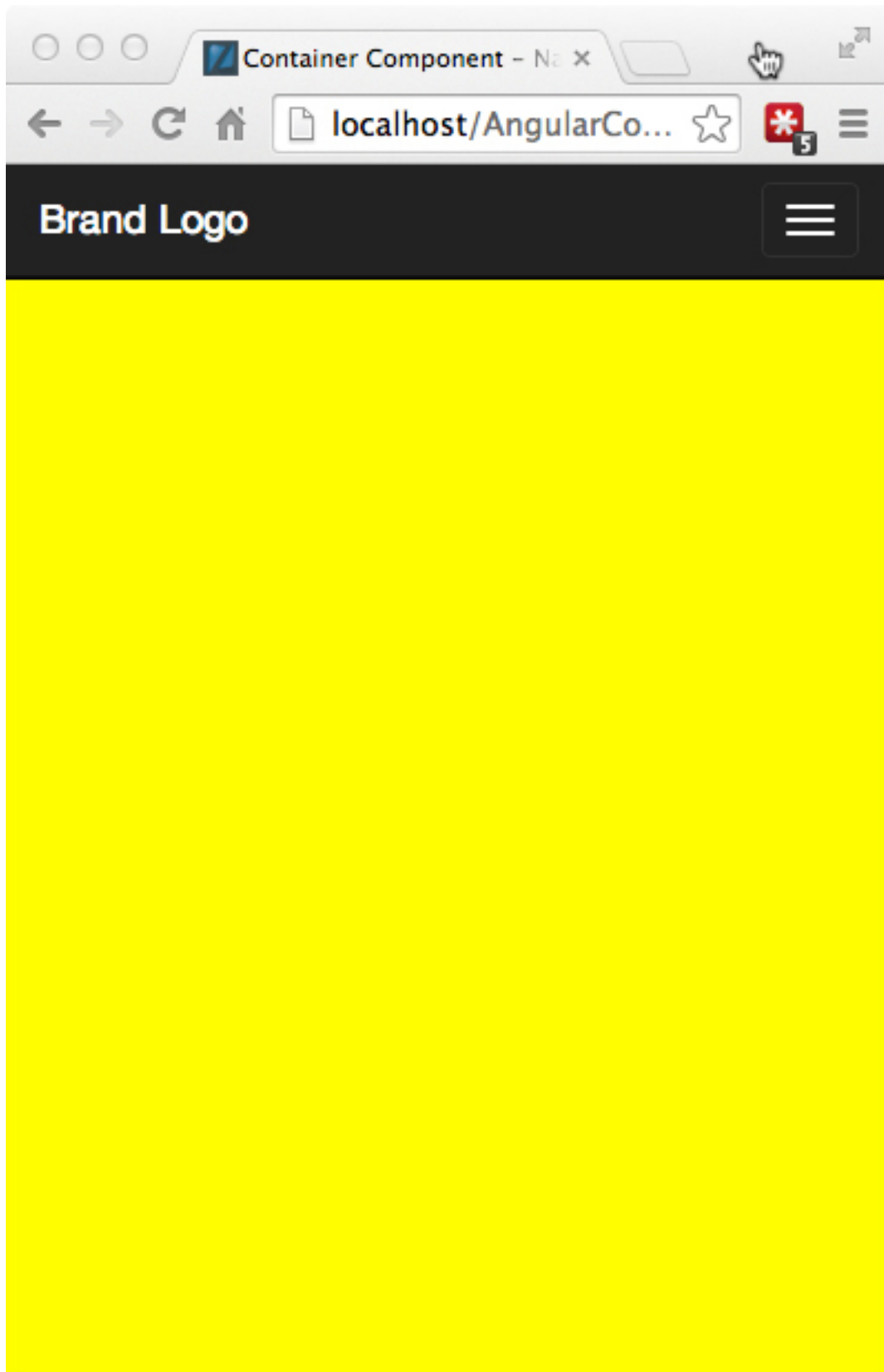
Navigation Bar - Dropdown Open State



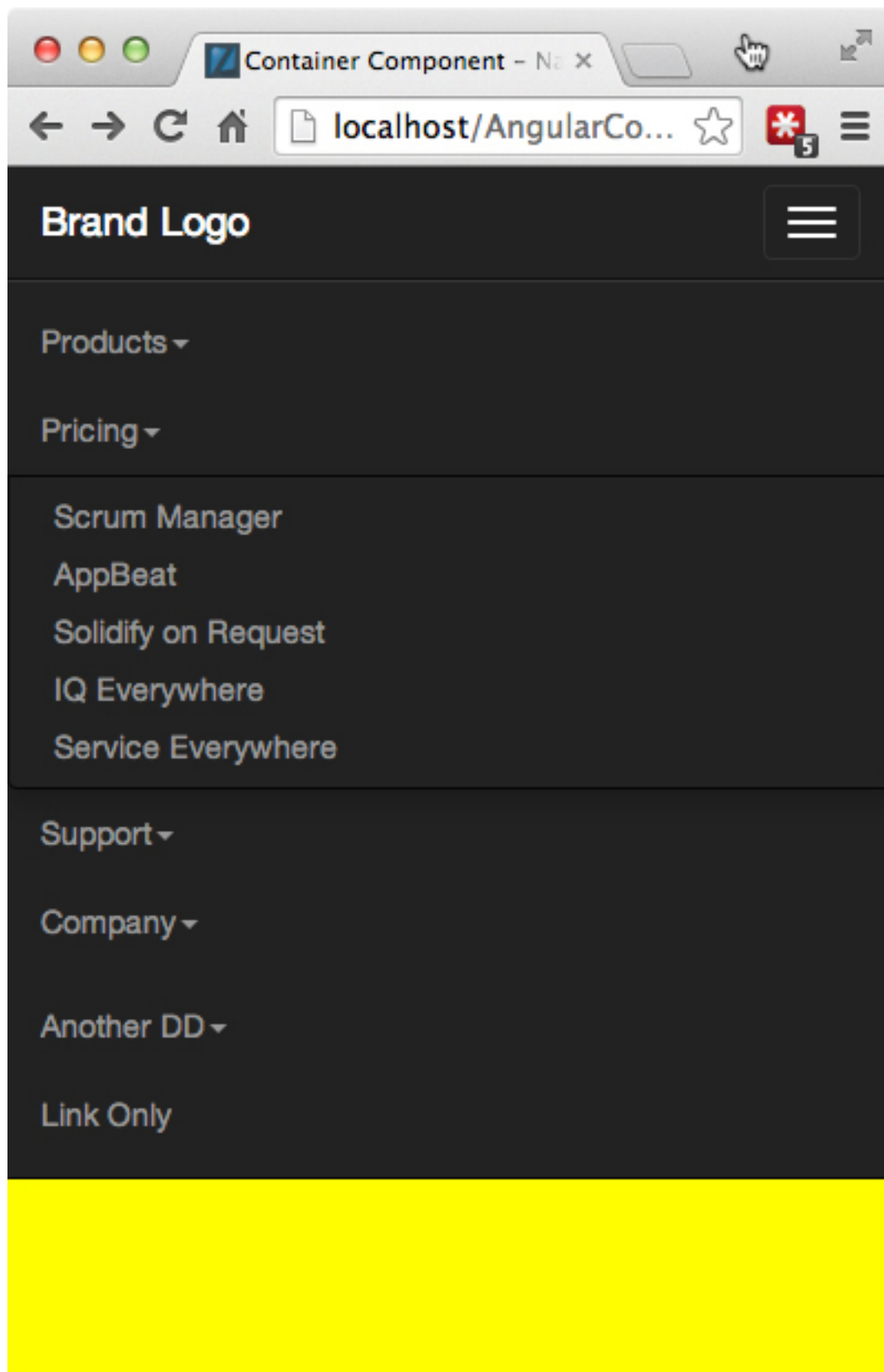
Navigation Bar - Dropdown Open - Menu Item Hover State



Navigation Bar - Minimal or Contents Hidden State (for less distraction while the user performs an important task)



Navigation Bar - Responsive / Mobile Contracted (left) and Expanded (right) States



Navigation Bar - Responsive / Mobile Contracted (left) and Expanded (right) States

If these images look like they are using Bootstrap “inverse” navigation header bar styling, it is because they are. As of this writing, Bootstrap 3.xx is the most widely used set of base styles for common HTML page components including navigation. But we are not endorsing Bootstrap because we use it as the styling base here. Some alternatives to Bootstrap for “responsive first” css frameworks include Foundation, InK, Zimit, and HTML KickStart among others. What we do endorse is using *a* CSS framework, along with a preprocessor like SASS or LESS, because it prevents engineering time waste in creating the same base functionality, provides a common language for UI engineering to communicate with visual and UX design teams, and provides presentable styling fallbacks for any components not covered by the organization’s style guides or pattern libraries.

## Data Model

For the purposes of setting up the context of our real-world example, not only are we working with a set of images from an imaginary design team, but we are also working with an imaginary back-end or server-side engineering team that have provided us with the following example JSON data object that represents the data attributes and structure that our navigation bar component can expect to find on page load or after an AJAX call.

### 6.0 Example JSON Data Object from AJAX or Initial Page Load

---

```
// global header json
{
  "header": [
    /* This data object translates to a dropdown container component */
    {
      "Products": [
        /* This data object translates to a menu item component */
        {
          "text": "Scrum Manager",
          "url": "/products/scrum-manager"
        }, {
          "text": "AppBeat",
          "url": "/products/app-beat"
        }, {
          "text": "Solidify on Request",
          "url": "/products/sor"
        }, {
          "text": "IQ Everywhere",
          "url": "/products/iq-anywhere"
        }, {
          "text": "Service Everywhere",
          "url": "/products/service-everywhere"
        }
      ]
    },
    {
      "Pricing": [
```

```

    {
      "text": "Scrum Manager",
      "url": "/pricing/scrum-manager"
    }, {
      "text": "AppBeat",
      "url": "/pricing/app-beat"
    }, {
      "text": "Solidify on Request",
      "url": "/pricing/sor"
    }, {
      "text": "IQ Everywhere",
      "url": "/pricing/iq-anywhere"
    }, {
      "text": "Service Everywhere",
      "url": "/pricing/service-everywhere"
    }
  ]],
  {"Support": [
    {
      "text": "White Papers",
      "url": "/support/papers"
    }, {
      "text": "Case Studies",
      "url": "/support/studies"
    }, {
      "text": "Videos",
      "url": "/support/videos"
    }, {
      "text": "Webinars",
      "url": "/support/webinars"
    }, {
      "text": "Documentation",
      "url": ""
    }, {
      "text": "News & Reports",
      "url": "/learn/news"
    }
  ]],
  {"Company": [
    {
      "text": "Contact Us",
      "url": "/company/contact"
    }
  ]
}
```



```
    }, {
      "text": "Jobs",
      "url": "/company/jobs"
    }, {
      "text": "Privacy Statement",
      "url": "/company/privacy"
    }, {
      "text": "Terms of Use",
      "url": "/company/terms"
    }
  ]
}
```

---

The items of note in the example data object are two levels of object arrays. The first level array represents data that will map to dropdown components in the navigation bar with the titles Product, Pricing, Support... The second level array represents the menu items that will populate each dropdown. The objects must be wrapped in arrays in order to guarantee order in JavaScript. Menu items in the above object include the minimum useful attributes: text label and URL. But they could also include attributes for special styles, states, alignment and other behavior.

Also, you may have noticed that the images contain some navigation bar items including an extra dropdown and a simple navigation link that are not found in the data object. These items have been added, not from data, but from custom HTML elements that we will create for designers that want to fill the navigation container via markup instead of data, a pretty nice bit of flexibility for our navigation bar component that requires some AngularJS trickery to accomplish.

## Simple Markup and Markup API

Looking once again at the list of requirements for our deliverables, we see that one requirement is to allow inclusion and configuration of the global navigation bar and its contents with the simplest markup we can get away with. This topic doesn't get much coverage in the greater AngularJS bogging universe, as that community is primarily experienced web developers.

However, in the context of large enterprise organizations that need to ensure a common corporate branding across all of its microsites, the ease or difficulty for individual business units to integrate the common styling and behavior makes a lot of difference in time needed to comply, the quality of the compliance, or whether there is any compliance at all beyond a corporate logo image. The difficulty is almost always compounded due to lack of competent web development skills across enterprise organizations.

As we've mentioned before, a direct result of excess time and technical difficulty in adhering to branding standards are microsites across business units of enterprise organizations the look and feel like they represent entirely different companies. This is very easy to notice if you browse different areas of almost any billion-dollar company.

## Getting Declarative

So our goal is to encapsulate the branding look and feel in some very simple markup that can be easily consumed by groups with minimal technical aptitude beyond basic HTML.

### 6.1 Component Encapsulation via Custom Elements and Attributes

---

```
<!-- example component markup usage -->
<body>

    <!-- include the global nav header with JSON menus or no contents using our
         custom element / directive -->
    <uic-nav-bar></uic-nav-bar>
    <!-- same as above using custom attributes as configuration APIs to define
         fixed position when the user scrolls and an available theme to use -->
    <uic-nav-bar
        sticky="true"
        theme="default">
    </uic-nav-bar>
    <!-- navigation header with one dropdown menu component and one menu item
         component as children -->
    <uic-nav-bar>
        <!-- Custom element for a dropdown menu container with custom attributes
             for display text and alignment APIs -->
        <uic-dropdown-menu
            position="right"
            text="A Dropdown Title">
            <-- custom element and attributes representing a single menu item
                 component and APIs link display text and URL -->
            <uic-menu-item
                text="First Item"
                url="http://david--shapiro.blogspot.com/">
            </uic-menu-item>
            <uic-menu-item
                text="Second Item"
                url="http://david--shapiro.blogspot.com/">
            </uic-menu-item>
            <uic-menu-item
                text="No Url"
                url="">
            </uic-menu-item>
        </uic-dropdown-menu>

        <-- a menu item component that is a child of the navigation bar
```

```
    component, and therefore appears directly on the navigation
    bar instead of in a dropdown container to illustrate robustness
    and versatility -->
    <uic-menu-item
      text="Link Only"
      url="http://david--shapiro.blogspot.com/">
    </uic-menu-item>
  </uic-nav-bar>
</body>
```

---

Notice how these custom elements and attributes we created via AngularJS directives are both very descriptive or and easy to understand for those without a PhD in computer science. After we construct the guts of our three components, anyone who includes this markup in a page is really including a lot of HTML, JavaScript, and CSS that they don't need to understand or worry about. Not only that, but the few hours it will take us to code the components will allow the consumer to have a complete navigation header with only five minutes of work.

When we publish the APIs and downloads of the corporate branding styles and pattern library to our, the bar for corporate look and feel compliance will be significantly lowered, and the excuses for non-compliance removed.

From the perspective of the component developer, we now have two new UI components, a dropdown and a menu item, that can be used anywhere in the page or application whether in a navigation bar or not.



**A History Tidbit** - Web development veterans should notice that the custom presentational aspect of what we are doing is nothing new. The ability to define specialized elements, attributes, and contents was one of the ideas behind XML (extensible markup language), XSL (extensible style sheet language) and XSLT (style sheet transformations) many years ago. Given that one Internet year is roughly equivalent to one dog year, this would have been about 85-90 years ago. Despite a major push by technical evangelists to popularize the use of XML/XSL for presentation in the greater web community, effective use of XSLT did require significant programming ability; whereas, CSS was much simpler and more accessible by hobbyists. CSS eventually won out leaving XML to be used as a (now obsolete) data transmission and configuration language. Even those uses have largely been replaced by much more concise JSON and YML.

One best practice we are borrowing from XML is the idea of name spacing. We are prefixing the component directives in our library with uic- for “user interface component”. It is strongly suggested to use a namespace to avoided naming clashes. Never prefix your components with already established prefixes like ng- or ui- since that means you have now effectively forked AngularJS or Angular-UI for your library- a maintenance nightmare. Although you may think of it as filling in the missing pieces of the core AngularJS directive offerings, calling your component ng-scroll, may cause breakage when the AngularJS team creates an ng-scroll.



**BEST PRACTICE** always name space your component directive libraries

## Building the UI Components

When planning and building out a hierarchical set of UI components, it almost always makes sense to start with the most granular and work up to the most general. Likewise, when building out a front-end application, it's often easiest to start from both ends (data and view) and work towards the middle.

In this case, we will start with the navigation item component and template, then proceed to the drop down menu, and finally to the navigation header. Along the way, we will get the opportunity to get some advanced AngularJS under our belt including cannibalizing or extending existing directives from AngularJS core and Angular-UI teams, working with events, event listeners, transclusion, watches, directives requiring other directives, and creating some supporting directives and services.

## The Menu Item Component

Below is the code for our menu item component, `<uic-menu-item>`. It is completely independent and self-contained except for the CSS.

### 6.2 Menu Item Directive

---

```
// As AngularJS and the example UI components built upon it are continuously
// evolving, please see the GitHub repo for the most up to date code:
// https://github.com/dgs700/angularjs-web-component-development
// example component markup usage, things to note in bold
// a simple menu item component directive
.directive('uicMenuItem', [function(){
  return {

    // replace custom element with html5 markup
    template: '<li ng-class="disablabable">' +
    // note the use of ng-bind vs. {{{}} to prevent any brief flash of the raw
    // template
    '<a ng-href="{{ url }}" ng-bind="text"' +
    ' ng-click="selected($event, this)"></a>' +
    '</li>',
    replace: true,
    // restrict usage to element only since we use attributes for APIs
    restrict: 'E',
```

```

// new isolate scope
scope: {
  // attribute API for menu item text
  text: '@',
  // attribute API for menu href URL
  url: '@'
},
controller: function($scope, $element, $attrs){
  // the default for the "disabled" API is enabled
  $scope.disableable = '';
  // called on ng-click
  $scope.selected = function($event, scope){
    // published API for selected event
    $scope.$emit('menu-item-selected', scope);
    // prevent the default browser behavior for an anchor element click
    // so that pre nav callbacks can execute and the proper parent
    // controller or service can handle the window location change
    $event.preventDefault();
    $event.stopPropagation();
    // optionally perform some other actions before navigation
  }
},
link: function(scope, iElement, iAttrs){
  // add the Bootstrap "disabled" class if there is no url
  if(!scope.url) scope.disableable = 'disabled';
}
};
})();

```

---

So here we have the code that executes anytime AngularJS finds a match with `<uic-menu-item>` in the page markup. This is a relatively simple component directive that has no hard outside dependencies. It replaces the custom element with normal HTML markup, checks attribute values for display text and URL, sets `class="disabled"` if not URL is provided, and emits a selected event when clicked. The template markup structure is generally compatible with Bootstrap styling, and default browser navigation behavior is prevented in favor of programmatic handling.

When it comes time to add the `uicMenuItem` component to our published component pallet we might add some API documentation like the following:

---

COMPONENT DIRECTIVE: `uicMenuItem`, `<uic-menu-item>`

**DESCRIPTION:**

A menu item component conforming to corporate branding and style standards that can include a URL and display text and be used to populate a `uicDropdownMenu` or `uicNavBar` container.

**USAGE AS ELEMENT:**

```
<uic-menu-item
  text="[string]" display title of dropdown
  url="[string]" URL for compiled anchor tag
>
</uic-menu-item>
```

**ARGUMENTS:**

Param	Type	Details
text	string	display text
url	string	

**EVENTS:**

Name	Type	Trigger	Args
'menu-item-selected'	\$emit()	click	Event Object

---

## Menu Item API Unit Test Coverage

Along with the menu item code, of course, comes the unit test coverage. The set up for the tests is almost identical to the set up for the smart button tests earlier, but we are including them this time as a refresher. The nice thing about Jasmine and most other unit test frameworks is that the code is mostly self-documenting, therefore, should be self-explanatory. The `describe()` and `it()` calls in bold, above should read in plain English to be consistent with the AngularJS practice of being as declarative and descriptive as possible.

## 6.3 Menu Item Unit Tests

---

```

describe('My MenuItem component directive', function () {
  var $compile, $rootScope, $scope, $element, element, $event;
  // manually initialize our component library module
  beforeEach(module('UIComponents'));
  // make the necessary angular utility methods available
  // to our tests
  beforeEach(inject(function (_$compile_, _$rootScope_) {
    $compile = _$compile_;
    $rootScope = _$rootScope_;
    // 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 menu item component in their page that covers all of
  // the API options
  var tpl = '<uic-menu-item text="First Item" ' +
    'url="http://david-- shapiro.blogspot.com/"></uic-menu-item>';
  // 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('Menu Item Component API coverage', function(){
    var scope;
    beforeEach(function(){
      compileDirective();
      // get access to the actual controller instance
      scope = $element.data('$scope').$$childHead;
      // create a fake event
      $event = $.Event( "click" );
    });
  });
}

```

```

    });
    it('should use the attr value (API) of "text" as the menu label',
        function(){
            expect(element).toContain("First Item");
        });
    it('should use the attr value (API) of "url" for the href url',
        function(){
            expect(element).toContain("http://david--shapiro.blogspot.com/");
        });
    it('should emit a "menu-item-selected" event (API) when selected',
        function(){
            spyOn(scope, '$emit');
            scope.selected($event, scope);
            expect(scope.$emit)
                .toHaveBeenCalled('menu-item-selected', scope);
        });
    });
});

// template with no URL attribute
var tplNoUrl = '<uic-menu-item text="First Item"></uic-menu-item>';
describe('Menu Item Component other functionality', function(){
    var scope;
    beforeEach(function(){
        compileDirective(tplNoUrl);
    });
    it('should add a "disabled" class if there is no URL provided',
        function(){
            expect($element.hasClass('disabled')).toBeTruthy();
        });
    });
});

```

---

In our unit tests we want to be sure that every bit of functionality related to the published APIs is covered at a minimum. Given that the APIs should remain consistent over a period of time while other internal functionality might change, we have separated the test coverage above into two blocks, one for APIs and the other for anything else covering non-API core functionality. In this specific case, auto-disabling the component in the absence of a URL is not part of the published API, but is an important behavior that users would expect to remain consistent.



## The Dropdown Component

Here is where the fun begins. Our menu item component has a single, simple purpose and has the same complexity as the smart button component example in the previous chapter. This sort of component directive is easy to code using the AngularJS docs and various AngularJS books and blogs as guides.

### Code Challenges and Choices

Our dropdown component, on the other hand, is a bit more complex and presents some interesting coding challenges and choices:

- Is there existing code to build from given that this is a common component?
- How does one dropdown know to close when another is opened given that any component should not have knowledge outside of itself?
- How do we add the flexibility to accept either JSON data or markup as menu item contents?
- How do we give this component the ability to exist in containers other than a global navigation bar?

### Development Strategy

Given that the dropdown component construction is going to require complexity far beyond coding a simple, stand-alone AngularJS directive we will use the following strategy as our approach:

1. Locate any existing open source code we can use or extend in order to
  - understand how others have approached this task
  - not reinvent the entire wheel from scratch
2. Assemble a template from Bootstrap HTML fragments for their version of a dropdown component in order to be Bootstrap style and class compatible
3. Identify complex tasks and functionality that require more coding tools than the AngularJS directive definition object has to offer.
4. Code supporting services and directives for the above first
5. Complete the code for our dropdown component directive using the supporting tools we created

## Reusing Existing Open Source Code

These days if we have a web programming challenge, the chances are that someone has already faced the same challenge and posted questions in places like StackOverflow or the Google AngularJS

group, or has solved it and bragged about their clever solution in a blog somewhere. One thing I see all the time in young, brilliant, yet inexperienced web developers is the tendency to waste time creating their own solution to a problem that has already been solved many time over.

We only hear about the popular JavaScript toolkits and frameworks like AngularJS, jQuery, Dojo, Ember.js, and Backbone.js, but there are over a thousand JavaScript frameworks and toolkits out there in the wild that someone wasted time creating for some project or organization that approximates the existing functionality of the top frameworks at best. In fact, every large enterprise software company that I have consulted work for has created their own JavaScript framework at some time in the past. It is extremely rare that a JavaScript framework created within an organization is so novel and exceptional that it catches fire in the open source community- examples being AngularJS (Google), Backbone.js (Document Cloud), Bootstrap (Twitter), and YUI (Yahoo).



**BEST PRACTICE** don't reinvent the wheel!

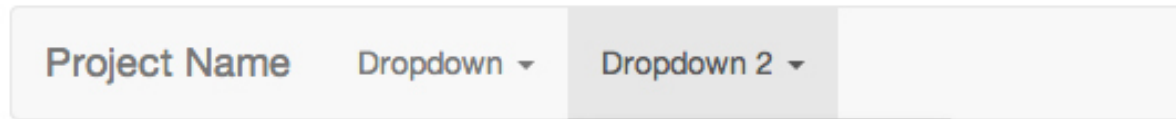
Our search of the web has revealed that dropdown components have been implemented many times, and two of the best candidates for sourcing some usable code or patterns are Bootstrap for the HTML, class and styling structure, and the Angular-UI project which has a sub project, Angular-Bootstrap, that provides AngularJS native implementations for most Bootstrap components.

## Where to Start?

Navigating to the JavaScript section of the Bootstrap documentation we see that Bootstrap defines a nice dropdown component:

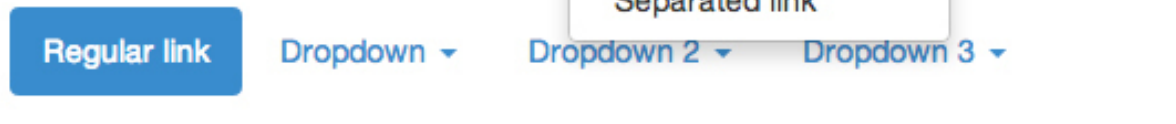
## Within a navbar

### EXAMPLE



## Within pills

### EXAMPLE



## Usage

Screen grab of a Bootstrap dropdown component from <http://getbootstrap.com/javascript/#dropdowns>

The dropdown in this screen grab looks pretty close to what we will need given some custom styling. Let's take a look at the HTML behind it:

```

▶ <li class="dropdown">...</li>
▼ <li class="dropdown open">
  ▼ <a href="#" id="drop2" role="button" class="dropdown-toggle" data-toggle="dropdown">
    "Dropdown 2 "
    <b class="caret"></b>
  </a>
  ▼ <ul class="dropdown-menu" role="menu" aria-labelledby="drop2">
    ▼ <li role="presentation">
      <a role="menuitem" tabindex="-1" href="http://twitter.com/fat">Action</a>
    </li>
    ▼ <li role="presentation">
      <a role="menuitem" tabindex="-1" href="http://twitter.com/fat">Another action</a>
    </li>
    ▼ <li role="presentation">
      <a role="menuitem" tabindex="-1" href="http://twitter.com/fat">Something else here</a>
    </li>
    <li role="presentation" class="divider"></li>
    ▼ <li role="presentation">
      <a role="menuitem" tabindex="-1" href="http://twitter.com/fat">Separated link</a>
    </li>
  </ul>
</li>
::after
</ul>

```

Screen grab of the HTML for Bootstrap dropdown component using Chrome dev tools

Looking at the HTML structure in Chrome Dev Tools, it looks like a very good starting point for

our Directive template, so we will use the right-click “copy as HTML” to grab the `<li>` fragment and paste it into our IDE of choice.

#### 6.4 Dropdown Directive Template

---

```
// html5 markup that replaces custom <uic-dropdown-menu> component element
var dropdownTpl =
  // the "namespaced" .uic-dropdown class can be used to provide
  // some encapsulation for the component css
  '<li class="uic-dropdown">'
  // directive to toggle display of dropdown
+ '  <a dropdown-toggle ng-bind-html="dropdownTitle">'
+ '<b class="caret"></b></a>'
  // this handles menu items supplied via JSON
+ '  <ul class="dropdown-menu" ng-if="jsonData">'
  // set .disabled class if no url provided
+ '    <li ng-repeat="item in menuItems" ng-class="disabable"
      ng-init="disabable=(item.url)?null:\'disabled\'">'
+ '      <a ng-href="{ { item.url } }" ng-bind="item.text"
      ng-click="selected($event, this)"></a>'
+ '    </li>'
+ '  </ul>'
  // this handles menu items supplied via markup
+ '  <ul class="dropdown-menu" ng-if="!jsonData" ng-transclude></ul>'
+ '</li>';
```

---

Listing 6.4 contains what will be our finished dropdown directive template. Given that the root element is a list item, `<li>`, it will need to be contained within an unordered list element, `<ul>`, for styling consistent with Bootstrap.css as a base that we override. Major omissions from the copied HTML fragment are the accessibility attributes, `role`, `aria-*`, and `tabindex`. For a production quality UI component library, these should *always* be present. Screen readers for the visually impaired have a tough time if they are not present, and if the work you are doing is for government clients or contractors, section 508 compliance may be a requirement. These attributes have been omitted from our example templates solely for the purpose of focusing on AngularJS functionality.

All of our AngularJS directives appear in bold, and these actually end up comprising the bulk of the template. So you can tell there is a lot going on here. For starters, the template has been manually in-lined as JavaScript. The example source file includes this along with the other dropdown directive and service blocks. In a true development and build environment we would prefer to maintain all templates of more than one line in separate AngularJS template files for ease of development, and in-line them as an intermediate build step. We will actually explore this and other build tasks in later chapters.

## CSS Namespacing

Scanning through the template sequentially, the first item of note is the class name used on the root element, `<li class="uic-dropdown">`. We are using our library prefix, `uic-`, in order to provide a minor level of encapsulation for component specific styling. The emphasis is on *minor* because a single unique class name likely is not enough extra specificity to prevent unintended bleed through of global styles alone. It's better than nothing, but less than using an id attribute to namespace. Singleton UI components, like the global navigation header we will discuss next, have the luxury of using id attributes for style name spacing since there will not be more than one in the page. Dropdowns, on the other hand, are almost guaranteed to be used more than once, restricting any name spacing to class names. In either case, a consistent and unique prefix should be used.

The next line of the template, `<a dropdown-toggle ng-bind="dropdownTitle">`, uses `ngBind` for the label text to display. We choose to do this, rather than using curly braces, should something interfere with fast processing of our directive code causing the dreaded FOUT (flash of unstyled text) usually associated with loading web fonts. In AngularJS, the FOUT includes the unevaluated `$scope` variable and braces.

## Borrowed State Toggling Functionality

`dropdownToggle` is a directive whose source comes primarily from the Angular-Bootstrap directive which itself is an Angularized version of the `dropdown.js` (jQuery based) functionality from Bootstrap.

### 6.5 DropdownToggle Directive

---

```
// Credit to the Angular-UI team:
// Angular ui.bootstrap.dropdownToggle
// https://github.com/angular-ui/bootstrap
// helper directive for setting active/passive state on the
// necessary elements
.directive('dropdownToggle', function() {
  return {
    // keep to attributes since this is not a UI component
    restrict: 'A',
    // list of UI components to work for
    require: '?^uicDropdownMenu',
    link: function(scope, element, attrs, dropdownCtrl) {
      // render inert if no dropdown controller is injected
      if ( !dropdownCtrl ) {
        return;
      }
      // set the toggle element in the dropdown component
      dropdownCtrl.toggleElement = element;
    }
  };
});
```

```

    // click event listener for this directive
    var toggleDropdown = function(event) {
        // prevent the browser default behavior for anchor elements
        event.preventDefault();
        event.stopPropagation();
        // check that we are not disabled before toggling visibility
        if ( !element.hasClass('disabled') && !attrs.disabled ) {
            // call toggle() on the correct component scope
            scope.$apply(function() {
                dropdownCtrl.toggle();
            });
        }
    };
    // add click evt binding
    element.bind('click', toggleDropdown);
    // clean up click event binding
    scope.$on('$destroy', function() {
        element.unbind('click', toggleDropdown);
    });
}
};
});

```

---

This is an example of not reinventing the wheel. For good Karma, any code borrowed should always include proper credit. Minor changes are in bold.

This directive is meant to work as part of the dropdown UI component directive to listen for click events on the dropdown header and tell it to change state of the `$scope.isOpen` boolean to the opposite in response. One thing we have not had a chance to cover in practice yet is the `require` option of the directive definition object to expose functionality. `require: '?uicDropdownMenu'` is making the `uicDropdownMenu` directive an optional dependency since our dropdown will be useful in many other containers besides the global navigation header. We are also restricting use of this directive to *attribute only* in order to keep the template class name as uncluttered as possible. Other than that, the directive has the same functionality as the Angular-Bootstrap version.

This directive is a great example of a task that could be accomplished in many different ways using AngularJS. For example, we could have rolled similar functionality into the component directive using `ngClick` on the `<a>` element to trigger a toggle, but since a directive already exists we save time and debugging effort by using it. Often times when working with AngularJS you will need to weigh the pros and cons of two or more implementation options, and the skill to do this can only be gained through experience. It cannot be effectively taught in a book, class or video.

Moving on to the element, `<ul>`, in the dropdown template we see that there are two of these, and

they use `ngIf` directives.

```
<ul class="dropdown-menu" ng-if="jsonData">
...
<ul class="dropdown-menu" ng-if="!jsonData" ng-transclude>
```

`ngIf` is different from `ngShow` or `ngHide` in that the latter are merely the “Angular way” of setting or removing `display:none` on the element. The elements are compiled, linked, and load any additional resources even if not part of DOM flow. This can be quite undesirable in certain situations. For example you have a form that shows and validates certain fields for this purpose, but shows and validates a slightly different set of fields for that purpose. Using `ngShow` and `ngHide` will not prevent the non-displayed fields from still being validated. `ngIf`, on the other hand, will only render an element in the DOM if it evaluates to true.

In our case, as you might be able to guess from the name of the argument, `jsonData`, one `<ul>` element is meant to contain menu items that originated from JSON data while the other `<ul>` element’s purpose is to contain menu item component directives transcluded from HTML markup, thus the `ngTransclude` directive on that one. You’ll notice that the contents of the first `<ul>` nearly match the markup in the `MenuItem` component template.

## Dropdown Menu SlideToggle Directive

Another implementation choice we have made with these elements is to attach a directive as a class name, as opposed to the more typical use of element of attribute.

### 6.6 DropdownMenu Directive

---

```
// the angular version of $('<div>.dropdown-menu').slideToggle(200)
.directive('dropdownMenu', function(){
  return {
    // match just classnames to stay consistent with other implementations
    restrict: 'C',
    link: function(scope, element, attr) {
      // set watch on new/old values of isOpen boolean
      // for component instance
      scope.$watch('isOpen', function( isOpen, wasOpen ){
        // if we detect that there has been a change for THIS instance
        if(isOpen !== wasOpen){
          // stop any existing animation and start
          // the opposite animation
          element.stop().slideToggle(200);
        }
      });
    }
  };
});
```

```

    }
  };
})

```

The reason chosen for using the class name is for *consistency* with other dropdown component implementations including Bootstrap's and Angular-Bootstrap's that act on the class name, dropdown-menu, via `jQuery.slideToggle()` or `ngAnimate` CSS animations. I have to admit that CSS animations are not one of my strengths, and after spending about an hour trying to get a slide-up and slide-down effect that looked as good as `slideToggle()`, I gave up in favor of creating a directive that simply wraps `jQuery.slideToggle()` in the Angular fashion, which looks good in both the desktop and mobile dimensions. It likely has something to do with the fact that we are not able to work with fixed heights on these elements.

So the code in listing 6.6 is a great example of encapsulating an imperative jQuery action into some declarative AngularJS. The directive simply sets a watch on the element's `$scope.isOpen` boolean value and if a change is detected, toggles the slide effect.

Getting back to the topic of using the class name for consistency, the purpose for that is so other developers who may fork this code, and who already have familiarity with Bootstrap will immediately know what is going on. In other words, it enhances maintainability. If we created a custom attribute name it would likely be more confusing since we must use the class name, `.dropdown-menu`, for matching CSS rules anyhow.



**BEST PRACTICE** if extending existing functionality, try to keep the naming consistent for ease of maintenance

Moving on to the last set of elements in our template for the `uicDropdownMenu` component directive, we see a few more core AngularJS directives being employed.

```

<li ng-repeat="item in menuItems" ng-class="disablabl"
    ng-init="disablabl=(item.url)?null:'disabled'">
  <a ng-href="{ item.url }" ng-bind="item.text"
    ng-click="selected($event, this)"></a>
</li>

```

This block of template is for the `<ul>` element that handles menu item data supplied via JSON, thus the use of `ngRepeat` to iterate over the array of supplied menu objects and create an `<li>` element for each.

We also initialize via `ngInit` a scope variable, `disablabl`, to insert the class name, “disabled”, should the menu item object not include a URL string. In other words, we gray the text color and make the element non-clickable. The AngularJS expression argument for `ngInit` is probably right on the edge of



being too long, ugly and confusing for effective inclusion in the template. Any more complexity in this expression, and we would certainly want to locate it in the directive's controller constructor instead. This is almost an example of where too much "declarativeness" can become counter productive, since as always, every line of code we write should be written with the *expectation that someone else will have to maintain or extend this code in the future*.

We are already familiar with the use of `ngHref`, `ngBind`, and `ngClick="select()"` on the `<a>` element. These are used in exactly the same way as our `MenuItem` component directive.

## Dropdown Service

So by now we have addressed and solved most of the requirements and issues for our dropdown menu UI component listed earlier, but we are saving the best for last. We still have not addressed how multiple dropdown menus can exist on the same page, either inside the global navigation bar container component, or in another container. How will one dropdown close upon another opening, whether inside or outside of a container component? We cannot use a container like our navigation bar to manage all of the dropdowns that may exist on the page since any container component should only have dominion over its children, and cannot know about the DOM world beyond it's element borders. Likewise, we want to avoid polluting the `$rootScope` object with a major operation like this since it becomes very easy to step on someone else's code or have ours stepped on.

As much as we would like to keep our UI component logic and presentation as "encapsulated" within the component directive as possible, this is a problem that clearly defies encapsulation of any sort. There is one tool left AngularJS provides that we can use to address this issue, and address it in a way that is injectable, testable, and allows our component code to retain its componentized integrity. If you are an AngularJS application developer, then you create these all the time, and you probably already guessed that it is the good old *AngularJS service*. Services are built specifically to exist as singletons at the root application level and handle specific functions or tasks that may be required by many UI components on a page. We use services when we need to interact with the non-AngularJS JavaScript world or we do not want to replicate the same logic every time a controller is instantiated. All of the core building blocks of an AngularJS application are contained in services including the logic for compiling templates, instantiating controllers, creating filters, making AJAX requests, console logging, and so on.

### 6.7 Dropdown Service

---

```
// because we have a transclusion option for the dropdowns we cannot
// reliably track open menu status at the component scope level
// so we prefer to dedicate a service to this task rather than pollute
// the $rootScope
.service('uicDropdownService', ['$document', function($document){
    // currently displayed dropdown
    var openScope = null;
    // array of added dropdown scopes
```

```

var dropdowns = [];
// event handler for click evt
function closeDropdown( evt ) {
    if (evt && evt.isDefaultPrevented()) {
        return;
    }
    openScope.$apply(function() {
        openScope.isOpen = false;
    });
};
// event handler for escape key
function escapeKeyBind( evt ) {
    if ( evt.which === 27 ) {
        openScope.focusToggleElement();
        closeDropdown();
    }
};
// exposed service functions
return {
    // called by linking fn of dropdown directive
    register: function(scope){
        dropdowns.push(scope);
    },
    // remove/unregister a dropdown scope
    remove: function(scope){
        for(var x = 0; x < dropdowns.length; x++){
            if(dropdowns[x] === scope){
                dropdowns.splice(x, 1);
                break;
            }
        }
    },
    // access dropdown array
    getDropdowns: function(){
        return dropdowns;
    },
    // access a single dropdown scope by $id
    getById: function(id){
        var x;
        for(x = 0; x < dropdowns.length; x++){
            if(id === dropdowns[x].$id) return dropdowns[x];
        }
    }
};

```

```

        return false;
    },
    // open a particular dropdown and set close evt bindings
    open: function( dropdownScope ) {
        if ( !openScope ) {
            $document.bind('click', closeDropdown);
            $document.bind('keydown', escapeKeyBind);
        }
        if ( openScope && openScope !== dropdownScope ) {
            openScope.isOpen = false;
        }
        openScope = dropdownScope;
    },
    // close a particular dropdown and set close evt bindings
    close: function( dropdownScope ) {
        if ( openScope === dropdownScope ) {
            openScope = null;
            // cleanup to prevent memory leaks
            $document.unbind('click', closeDropdown);
            $document.unbind('keydown', escapeKeyBind);
        }
    }
};
})();

```

For our purposes, using a service to hold references to all instantiated dropdown on the page and track their open or closed state makes perfect sense, and most of the needed functionality already exists in the Angular-Bootstrap dropdown service. All we have done above is borrow that functionality rather than *reinventing the wheel* once again, and added a few more useful functions of our own. All of the good stuff in listing 6.7 is in bold.

Scanning through the code for our dropdown service we see that we have some private variables, `openScope` which holds a reference to the one dropdown scope on the page that is expanded if any, and `dropdowns` which is an array of all dropdown component instances that have registered themselves with this service. We also have some private functions, `closeDropdown()` which toggles the `$scope.isOpen` boolean to false on the expanded dropdown, and `escapeKeyBind()` which handles key press events for the escape key only to close any expanded dropdown. The object returned by the service includes the public functions `register(scope)`, `getDropdowns()`, `getById(id)`, `open(scope)`, and `close(scope)` through which the dropdown components may interact with the service.

The dropdown service also acts as a buffer for binding and removing events to the DOM document since this is where a click outside of any dropdown must be captured in order for any expanded dropdown to be closed. The `$document.unbind()` calls are especially important for preventing runaway memory leakage as the user interacts with the page.

## Dropdown Menu Component Directive

By now we have run through the code and reasoning behind it for all the supporting directives and service for our dropdown menu UI component directive. We have finally arrived at the directive component itself.

### 6.8 Dropdown Menu Component Directive

---

```
// As AngularJS and the example UI components built upon it are continuously
// evolving, please see the GitHub repo for the most up to date code:
// https://github.com/dgs700/angularjs-web-component-development
// Primary dropdown component directive
// this is also technically a container component
.directive('uicDropdownMenu', [
  'uicDropdownService', function(uicDropdownService){
    return {
      // covered earlier
      template: dropdownTpl,
      // component directives should be elements only
      restrict: 'E',
      // replace custom tags with standard html5 markup
      replace: true,
      // allow page designer to include menu item elements
      transclude: true,
      // isolate scope
      scope: {
        url: '@'
      },
      controller: function($scope, $element, $attrs){
        // $scope.disableable = '';
        // persistent instance reference
        var that = this,
            // class that would set display: block
            // if CSS animations are to be used for slide effect
            closeClass = 'close',
            openClass = 'open';

        // supply the view-model with info from json if available
        // this only handles data from scopes generated by ng-repeat
        angular.forEach($scope.$parent.menu,
          function(menuItems, dropdownTitle){
            if(angular.isArray(menuItems)){
              // uses ng-bind-html for template insertion
              $scope.dropdownTitle = dropdownTitle
                + '<b class="caret"></b>';
            }
          }
        );
      }
    };
  })
];
```

```

        $scope.menuItems = menuItems;

        // add a unique ID matching title string for future reference
        $scope.uicId = dropdownTitle;
    }
});
// supply string value for dropdown title via attribute API
if($attrs.text){
    $scope.uicId = $attrs.text;
    $scope.dropdownTitle = $scope.uicId + '<b class="caret"></b>';
}
// indicate if this component was created via data or markup
// and hide the empty <ul> if needed
if($scope.menuItems){
    $scope.jsonData = true;
}
// add angular element reference to controller instance
// for later class toggling if desired
this.init = function( element ) {
    that.$element = element;
};
// toggle the dropdown $scope.isOpen boolean
this.toggle = function( open ) {
    $scope.isOpen = arguments.length ? !!open : !$scope.isOpen;
    return $scope.isOpen;
};
// set browser focus on active dropdown
$scope.focusToggleElement = function() {
    if ( that.toggleElement ) {
        that.toggleElement[0].focus();
    }
};
// event handler for menu item clicks
$scope.selected = function($event, scope){
    $scope.$emit('menu-item-selected', scope);
    $event.preventDefault();
    $event.stopPropagation();
    // optionally perform some action before navigation
}
// all dropdowns need to watch the value of this expr
// and set evt bindings and classes accordingly
$scope.$watch('isOpen', function( isOpen, wasOpen ) {

```

```

        if ( isOpen ) {
            $scope.focusToggleElement();
            // tell our service we've been opened
            uicDropdownService.open($scope);
            // fire off an "opened" event (event API) for any listeners
            //out there
            $scope.$emit('dropdown-opened');
        } else {
            // tell our service we've been closed
            uicDropdownService.close($scope);
            // fire a closed event (event API)
            $scope.$emit('dropdown-closed');
        }
    });
    // listen for client side route changes
    $scope.$on('$locationChangeSuccess', function() {
        $scope.isOpen = false;
    });
    // listen for menu item selected events
    $scope.$on('menu-item-selected', function(evt, element) {
        // do something when a child menu item is selected
    });
},
link: function(scope, iElement, iAttrs, dropdownCtrl){
    dropdownCtrl.init( iElement );
    // add an element ref to scope for future manipulation
    scope.iElement = iElement;
    // add to tracked array of dropdown scopes
    uicDropdownService.register(scope);
}
};
}))

```

All of the key code is in bold. The `$scope.$watch( currentValue, previousValue )` command is a great illustration of using manual AngularJS data-binding to save writing a ton of code. If a change is detected between the pre-digest and post-digest values of `$scope.isOpen` as a result of a click, the dropdown service is told to update the scope that is referenced by its `openScope` variable. There is no need to manually create variables to hold old values and new values, or functions specifically designed to compare the values. Handling such a task imperatively with jQuery would take, at best, dozens of lines of code.

Additionally, the triggering of `$scope.$watch()` fires the events, “dropdown-opened” and “dropdown-closed” to satisfy the event API’s that we publish for our dropdown component. Custom action

events serve to make the overall implementation of the component more robust and useful to any container or page it resides in. Listeners for these events can perform related actions like expanding or contracting a section of the page or deactivating other components. If you scan all the code for our global navigation container, you'll see that none of it listens for these events, but there's a high probability that consumers of our component library will expect some sort of action callback events to be available as part of the API given that most quality widget libraries include them. This directive does set listeners (soft dependencies) for actions from it's children, as well as, route changes from `$routeProvider` for premium functioning.



**BEST PRACTICE** Use custom action events!

The controller code performs some tests to determine whether to set the key scope variables, `dropdownTitle` and `menuItems`, from JSON data or user supplied attribute API values. For component instances that do not instantiate `MenuItems` (JSON supplied data), we simulate some of the `MenuItem` component API functionality with `$scope.selected()`. There is also a controller instance function, `this.toggle()`, that gets called from the `dropDownToggle` helper directive which toggles the boolean value for `$scope.isOpen`. Finally, as part of the linking function, the component registers itself with the `dropdownService`.

## Dropdown API Documentation

When it comes time to add the `uicDropdownMenu` container component to our published component pallet we might add some API documentation like the following:

---

COMPONENT DIRECTIVE: `uicDropdownMenu`, `<uic-dropdown-menu>`

USAGE AS ELEMENT:

```
<uic-dropdown-menu
  text="[string]" display title of dropdown
>
  Transclusion HTML, can include <uic-menu-item>
</uic-dropdown-menu >
```

ARGUMENTS:

Param	Type	Details
text	string	display title, i.e. "About Us"

## EVENTS:

Name	Type	Trigger	Args
'dropdown-opened'	\$emit()	click, blur	Event Object
'dropdown-closed'	\$emit()	click, blur	Event Object
'menu-item-selected'	\$on()	N/A	Event Object, TargetScope
'menu-item-selected'	\$emit()	click	Event Object

## Dropdown Menu Unit Test Coverage

At this point, our Dropdown UI container component plus dependencies is almost complete, with the exception of test coverage. So let's make sure all of the dropdown UI component's API methods, properties, events, and services have a corresponding unit test.

### 6.9 Dropdown Menu Directive and Service Unit Tests

---

```
// Unit test coverage for the Dropdown UI component
describe('My Dropdown component directive', function () {
  var $compile, $rootScope, $scope, $element, element, $event,
      $_uicDropdownService;
  // manually initialize our component library module
  beforeEach(module('uiComponents.dropdown'));
  // make the necessary angular utility methods available
  // to our tests
  beforeEach(inject(function (_$compile_, _$rootScope_) {
    $compile = _$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 menu item component in their page that covers all of
  // the API options
  var tpl =
    '<uic-dropdown-menu text="Custom Dropdown">' +
    '  <uic-menu-item text="Second Item" url="http://google.com/">' +
    '    </uic-menu-item>' +
    '  <uic-menu-item text="No Url" url=""></uic-menu-item>' +
    '</uic-dropdown-menu>';
```



```

// template whose contents will be generated
// with JSON data
var tplJson = '<uic-dropdown-menu></uic-dropdown-menu>';
// some fake JSON
var menu = { "Company": [
  {
    "text": "Contact Us",
    "url": "/company/contact"
  }, {
    "text": "Jobs",
    "url": "/company/jobs"
  }, {
    "text": "Privacy Statement",
    "url": "/company/privacy"
  }, {
    "text": "Terms of Use",
    "url": "/company/terms"
  }
]
};
// 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('Dropdown Component markup API coverage', function(){
  var scope;
  beforeEach(function(){
    compileDirective();
    // get access to the actual controller instance
    scope = $element.data('$scope').$$childHead;
    // create a fake click event
    $event = $.Event( "click" );
  });
});

```

```

    it('should use the attr value (API) of "text" as for the label',
      function(){
        expect(element).toContain("Custom Dropdown");
      });
    it('should transclude the 2 menu item components (API)', function(){
      expect($element.find('li').length).toBe(2);
    });
    it('should toggle open state when clicked', function(){
      spyOn(scope, '$emit');
      scope.isOpen = false;
      // click on the dropdown to OPEN
      $element.find('a').trigger('click');
      expect(scope.isOpen).toBe(true);
      expect(scope.$emit).toHaveBeenCalled('dropdown-opened');
      // click on the dropdown to CLOSE
      $element.find('a').trigger('click');
      expect(scope.isOpen).toBe(false);
      expect(scope.$emit).toHaveBeenCalled('dropdown-closed');
    });
  });
  // test JSON constructed dropdowns
  describe('Dropdown Component JSON API coverage', function(){
    var scope;
    beforeEach(function(){
      $scope.$parent.menu = menu;
      compileDirective(tplJson);
      scope = $element.data('$scope').$$childHead;
    });
    it('should get its title text from the menu JSON obj key', function(){
      expect(scope.dropdownTitle).toEqual("Company");
    });
    it('should know that it is using JSON data for rendering', function(){
      expect(scope.jsonData).toBe(true);
    });
    it('should render the correct number of menu items (4)', function(){
      expect($element.find('li').length).toBe(4);
    });
    // coverage for other service methods not covered above
    describe('Dropdown Service methods', function(){
      // create another scope so there is more than one
      var anotherScope;
      // get an explicit reference to the dropdown service

```

```

beforeEach(inject(function (uicDropdownService) {
    $_uicDropdownService = uicDropdownService;
    anotherScope = $element.data('$scope').$$childHead;
    $_uicDropdownService.register(anotherScope);
}));
it('should now have 2 registered dropdowns', function(){
    // covers .register() and .getDropdowns() methods
    expect($_uicDropdownService.getDropdowns().length).toBe(2);
});
it('should retrieve a dropdown by id', function(){
    var testScope = $_uicDropdownService.getById(anotherScope.$id);
    expect(testScope).toBe(anotherScope);
});
});
});
});

```

---

Our DropdownSpec.js above is a bit more extensive than the menu item tests since this component is responsible for a lot more functionality. Most of the initial setup is the same, and the code lines of note are, as usual, in bold.

## Custom HTML Tag Coverage

For the first set of tests we are creating one fake template that simulates the custom HTML markup a page developer would use if building a dropdown menu manually using our container component. For the second set of tests we have the same custom tag, `<uic-dropdown-menu>`, empty of custom `<uic-menu-item>` elements, and a fake JSON object that the component will use to populate itself.

Our first priority is to make sure the APIs we plan to publish have *thorough* coverage. So we include tests for the custom attribute, “text”, and the menu item contents (transclusion) API. Along with the compiled and linked directive template, we have also created a fake “click” event that we use to simulate toggling the dropdown menu’s open and closed states.

You’ll notice the test for toggling open and closed states is fairly long which is necessary to cover these actions from all angles including the events, “dropdown-open” and “dropdown-closed”, that should be emitted and the scope boolean property, `isOpen`, that should switch from false to true to false. We are also implicitly covering the exposed service methods, `open` and `close`, which actually toggles the `isOpen` variable for the dropdown scope.

## JSON Data Coverage

Our second “describe” block contains the tests meant to cover the dropdown component in situations where it would automatically generate its contents of `<uic-menu-item>` components from supplied

JSON data. So we test that the title string is correctly read, that `scope.jsonData` gets set to true, and that four `<li>` elements have been created that match the four menu item objects from the JSON object.

## Nested `describe()` Blocks

We also take advantage of the fact that we already created a fake directive instance to cover some of the other exposed dropdown service methods including `.getDropdowns()`, `.register(dropdown)`, and `.getById(scope)`. We do this with a nested `describe()` block with an additional scope. We only use this scope to test the dropdowns array length since there really is only one scope to test at the dropdown menu level. In the next section when we create test coverage at the global navigation bar level we can complete the dropdown service test coverage since we will have multiple different dropdown scopes to work with.

## Global Navigation Bar Container

We finally reach the top level of our example set of components and containers with our global navigation bar UI container component (NavBar). Our NavBar component builds on the concepts we covered in constructing the dropdown menu component. It's actually somewhat simpler since this component is essentially a singleton in that there is only meant to be one instance of this in the page, so we do not have to devise a way to track and coordinate several of these for proper functioning and display.

An implementation that provides APIs that cover our business requirements from the beginning of this chapter should be fairly straightforward. However we do have a challenge concerning the requirement that the NavBar be able to contain child dropdowns and menu items that are generated from *both* page developer markup *and* JSON data objects that will require us to deviate a little from the stock AngularJS directive APIs. Additionally, given that the NavBar is likely to be a component in a single page application, populating it with dropdowns only on instantiation will not be enough. As application routing takes place (during runtime), a versatile NavBar should be able to add or delete menu dropdowns upon request from the router or application.

## Business Requirements

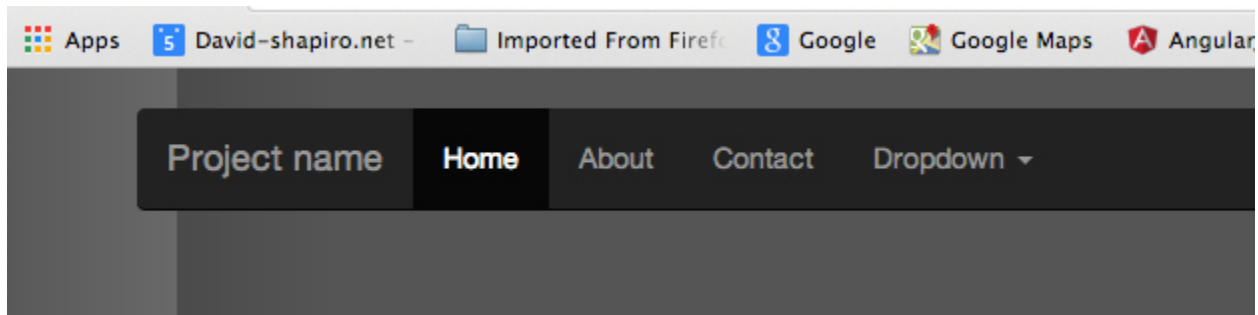
Recall that our global NavBar component must have the following attributes and capabilities:

- Viewport responsive styles for desktop and mobile display
- API options for “minimal” and “maximal” content display states with the ability to switch during run-time to support single page apps
- API options for static (top of document) or sticky (top of viewport) positioning
- API method for including an optional CSS sub-theme class
- Easy content creation for non-technical page authors via custom HTML tags and attributes

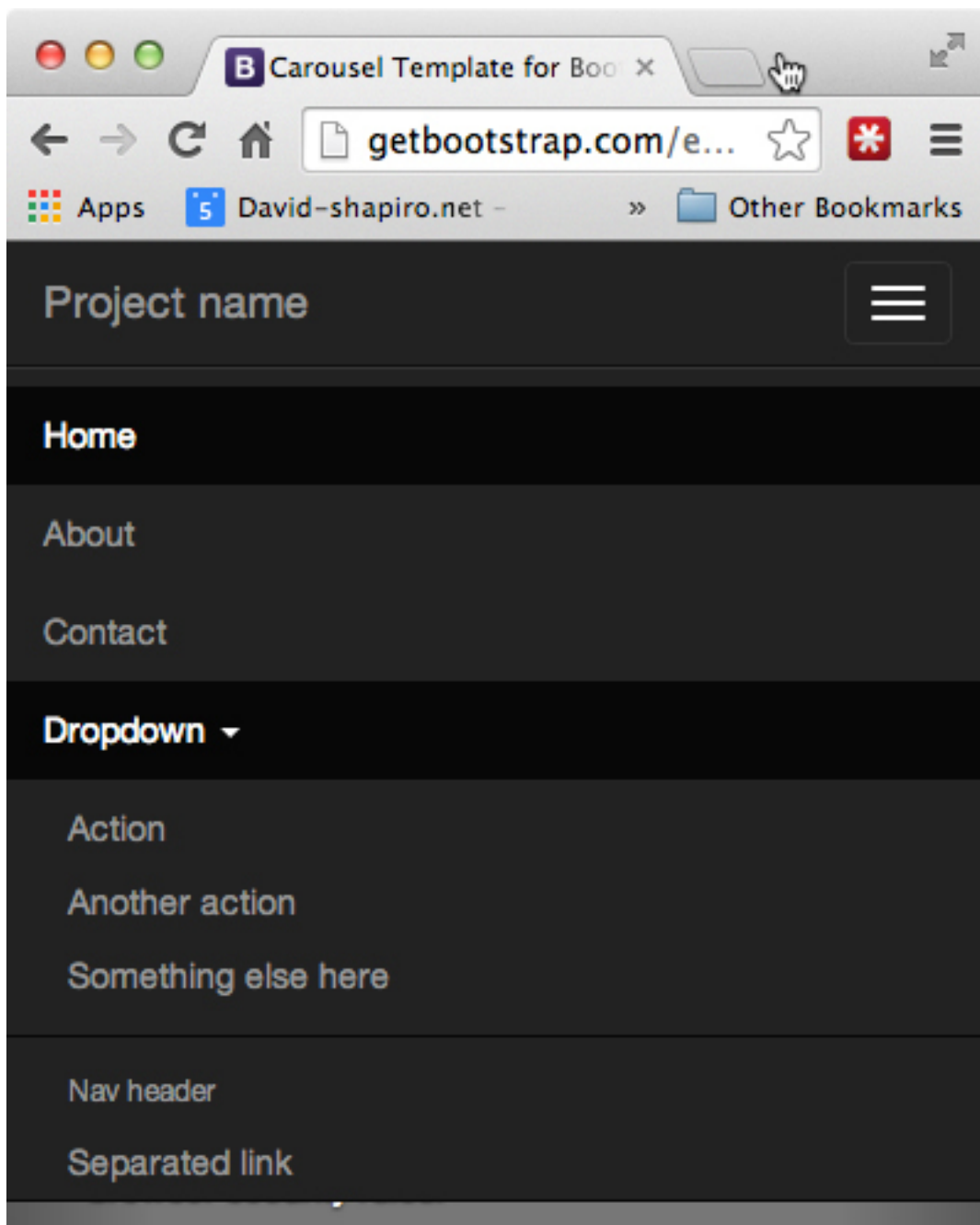
- Programmatic content creation via JSON data objects
- Ability to include both static and JSON dropdowns in the same instance
- Ability to add or remove child dropdowns during runtime via an event API

## uicNavbar Template

We will follow the same time-saving strategy in generating our container template as we did with the dropdown template by seeing what Bootstrap has to offer as a starting point. Browsing around Bootstrap's GetStarted page, we see that there are some starter templates with navigation headers that look close to what we need. In fact, that one for the example carousel template looks like an ideal candidate. It has both individual menu items and a dropdown.



<http://getbootstrap.com/examples/carousel/> example navigation header expanded



<http://getbootstrap.com/examples/carousel/> example navigation header mobile

So let's open up our browser dev tools and take a look at the HTML structure:

```

::before
▼ <div class="navbar navbar-inverse navbar-static-top" role="navigation">
  ::before
  ▼ <div class="container">
    ::before
    ▼ <div class="navbar-header">
      ::before
      ▼ <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=
        ".navbar-collapse">
        <span class="sr-only">Toggle navigation</span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
      </button>
      <a class="navbar-brand" href="#">Project name</a>
    </div>
    ▼ <div class="navbar-collapse collapse">
      ::before
      ▼ <ul class="nav navbar-nav">
        ::before
        ▼ <li class="active">
          <a href="#">Home</a>
        </li>
        ▼ <li>
          <a href="#about">About</a>
        </li>
        ▼ <li>
          <a href="#contact">Contact</a>
        </li>
        ▼ <li class="dropdown">
          ▶ <a href="#" class="dropdown-toggle" data-toggle="dropdown">...</a>
          ▶ <ul class="dropdown-menu">...</ul>
        </li>
      </ul>
    </div>
  </div>
  ::after
  ...

```

HTML fragment from <http://getbootstrap.com/examples/carousel/> navigation header

The advantage of doing this when working with Bootstrap derived CSS is that all the necessary class names are all already present including those necessary for responsive design. So we will start with this HTML fragment, replace the menu items and dropdowns with our custom dropdown element, and add a few more AngularJS directives for proper functioning and display. The end result will be our component template below:

### 6.10 Global Navigation Bar Container Template

---

```
// html5 markup that replaces custom <uic-nav-bar> component element
var navbarTpl =
  '<nav id="uic-navbar" class="navbar navbar-inverse" ng-class="[position,theme]">'
+ '  <div class="container-fluid">'
+ '    <div class="navbar-header">'
+ '      <button class="navbar-toggle" type="button" '
+ '        'ng-click="isCollapsed = !isCollapsed">'
+ '        <span class="sr-only">Toggle navigation</span>'
+ '        <span class="icon-bar"></span>'
+ '        <span class="icon-bar"></span>'
+ '        <span class="icon-bar"></span>'
+ '      </button>'
+ '      <a class="navbar-brand" ng-href="{ { homeUrl } }">Brand Logo</a>'
+ '    </div>'
+ '    <div class="collapse navbar-collapse" collapse="isCollapsed">'
// this renders if menu json data is available
+ '      <ul class="nav navbar-nav" ng-hide="minimalHeader">'
+ '        <uic-dropdown-menu ng-repeat="menu in menus"></uic-dropdown-menu>'
+ '      </ul>'
// this renders if the designer includes markup for dropdowns
+ '      <ul class="nav navbar-nav" ng-hide="minimalHeader" uic-include></ul>'
+ '    </div>'
+ '  </div>'
+ '</nav>';
```

---

You'll notice that most of the non-AngularJS HTML is the same. Running through our additions, the first item of note is the id name we are using, `<nav id="uic-navbar">`. Given that our NavBar is a *singleton* component, we have the luxury of being able to use a unique id name with our namespace for stronger CSS encapsulation. Any special branding style rules can now be prefixed with `#uic-navbar`.

Next we have `ng-class="[position,theme]"` which uses the array option for `ngClass`. The position and theme scope variables will be our CSS hooks for adding class names that would position the NavBar as either static (top of document) or fixed (top of viewport), and adding an optional sub-theme class for things like other brand colors depending on location within the entire site.

`ng-click="isCollapsed = !isCollapsed">` adds the collapse and expand click or touch functionality for the NavBar in its mobile state. This works with the div element that has the custom attribute `collapse="isCollapsed"`. Next we have an anchor, `<a>` placeholder element with a dynamic home URL scope variable that would normally be used to wrap a branding logo.

Finally we arrive at the meat and potatoes of our new container component, two `<ul>` elements that will be filled with dropdown or menu item components via the two different API methods. The



first that includes the custom element for our dropdown component, `<uic-dropdown-menu>`, along with an `ngRepeat` that iterates over a `menus` object, is what will ultimately be filled with dropdowns and menu items derived from JSON data. The second `<ul>` element has a special attribute directive, `uic-include`, needed to populate the container with the markup option for including dropdowns and menu items in the `NavBar` component to overcome an edge case situation for a partial transclusion that AngularJS core does not provide. We will explain further when we create the directive code for this below, and it will function as a special substitute for what we would normally want to do with `ngTransclude`. Both `<ul>` elements also have `ng-hide="minimalHeader"`, which fulfills our API requirement of allowing the page or application to optionally hide the `NavBar` contents in special situations.

## Supporting Services and Directives

Similar to our dropdown component, our `NavBar` component has some complexities that are best organized into their own functional components. The first concern is how best to manage situations where the dropdowns and menu items are representations of supplied JSON data objects. The norm in the AngularJS world is to wrap the various methods of data transmission, organization, and retrieval into “services” which is what we will do.

Our second concern is much the same in that we need to handle `NavBar` populations with dropdowns and menu items from page developer supplied markup, rather than via JSON data. The major difference is that the information comes from HTML custom element markup, and conceptually is more closely related with the `V` of `MV*` whatever. In the AngularJS world, the accepted container for this functionality is a directive. In 98% of most AngularJS situations we would solve this using `ngTransclude`, since it was designed just for that. But we have a bit of a wrinkle with `ngTransclude`’s default behavior of maintaining any transcluded DOM bindings to its original scope context. This won’t work for us because the isolate scope of our `NavBar` container component must be the common ancestor of all scopes bound to its child DOM elements including all dropdowns and menu items regardless of origin. This is necessary in order to maintain consistent, decoupled communication via `$emit()`, and `$broadcast()` to and from all dropdown components. What happens with a plain vanilla `ngTransclude` is the scope of the transcluded dropdown ends up being a sibling scope with that of the `NavBar`, and the `NavBar` is unable to use events to communicate like it can with the JSON generated dropdowns since those are child scopes.

Here is the code for our supporting service and directive:

### 6.11 Global Navigation Component Supporting Directive and Service

---

```
angular.module('uiComponents.navbar', ['uiComponents.dropdown'])
  // utility functions for nav bar population
  .service('uicNavBarService', [
    '$window', function($window){
      // functionality can expanded to include menu data via REST
      // check if a menus json object is available
      this.getMenus = function(){
        if($window.UIC && $window.UIC.header){
          return $window.UIC.header;
        }else{
          return false;
        }
      };
    }
  ])

  // utility directive that replaces ngTransclude to bind the content
  // to a child scope of the directive rather than a sibling scope
  // which allows the container component complete control of its
  // contents
  .directive('uicInclude', function(){
    return {
      restrict: 'A',
      link: function(scope, iElement, iAttrs, ctrl, $transclude) {
        $transclude(scope, function(clone) {
          iElement.append(clone);
        });
      }
    };
  });
}
```

---

The 'uicNavBarService' service is nothing special. For our purposes, it just looks for any menu JSON data that may have been bootstrapped with the page load. Conceptually it would be the logical container for additional methods of data retrieval and storage. For example, we could add functionality to retrieve data from a REST service, and we could add functionality to write out a JSON representation object of the current menu structure to local or session storage, as well as read it back in in order to maintain state should the user navigate elsewhere for a moment.

To address our transclusion scope issue, the solution ends up being quite simple. When a directive definition object (as we'll see below) has transclude set to true, the associated ngTransclude directive in the directive template is placed at the template attach point for the original (pre-compiled) markup, and a \$transclude function is injected into the linking function of the ngTransclude core

directive. A complete clone of this markup, original scope bindings and all, is appended to that element.

If we take a look at the source code for `ngTransclude`, it's really just a wrapper for `jQuery.append()`:

```
restrict: 'A',
link: function(scope, iElement, iAttrs, ctrl, $transclude) {
    $transclude(function(clone) {
        iElement.append(clone);
    });
}
```

There's a bit of an obscure AngularJS API option for the injected `$transclude` function that allows us to override the default scope with another if we include it as the first argument to the `$transclude` function. In our case, we want to override the default transcluded scope with our directive scope, so that resulting isolate scope of the compiled dropdown directive will be a child rather than a sibling scope of the `NavBar`. A good analogy would be what happens when a JavaScript closure is created. An extra context scope gets inserted into the context scope chain of the accessing object.

Luckily for us, the replacement scope we need is exactly the one that is injected into the directive linking function so all we have to do is include that scope reference as the optional first argument to the `$transclude` function as we have done in our `uicInclude` directive- basically the addition of a single word. If this seems exceptionally confusing, which it should, I suggest loading the example code available on the GitHub repo for this book into a Chrome browser with the Angular Batarang extension installed and enabled. Look at the scope hierarchy for the components while swapping `uicInclude` with `ngTransclude` in the `NavBar` template.

## NavBar Container Component Code

Now that we've coded all of the supporting functionality for the `NavBar` component, we can create the code for the component itself. As an extra bonus, since we are including significant DOM rewriting capability during runtime as part of the required functionality, we get to make use of the rarely used `$compile` service in our directive.

### 6.12 Global Navigation UI Container Component Directive

---

```
// As AngularJS and the example UI components built upon it are continuously
// evolving, please see the GitHub repo for the most up to date code:
// https://github.com/dgs700/angularjs-web-component-development
// Navigation Bar Container Component
.directive('uicNavBar', [
    'uicDropdownService',
    'uicNavBarService',
    '$location',
```

```

'$compile',
function( uicDropdownService, uicNavBarService, $location, $compile){
  return {
    template: navbarTpl,
    restrict: 'E',
    // allow page designer to include dropdown elements
    transclude: true,
    replace: true,
    // isolate scope
    scope: {
      // attribute API for hiding dropdowns
      minimalHeader: '@minimal',
      homeUrl: '@'
    },
    controller: [
      '$scope',
      '$element',
      '$attrs', function($scope, $element, $attrs){
        // make sure $element is updated to the compiled/linked version
        var that = this;
        this.init = function( element ) {
          that.$element = element;
        };
        // add a dropdown to the nav bar during runtime
        // i.e. upon hash navigation
        this.addDropdown = function(menuObj){
          // create an isolate scope instance
          var newScope = $scope.$root.$new();
          // attach the json obj data at the same location
          // as the dropdown controller would
          newScope.$parent.menu = menuObj;
          // manually compile and link a new dropdown component
          var $el =
            $compile('<uic-dropdown-menu></uic-dropdown-menu>')(newScope);
          // attach the new dropdown to the end of the first child <ul>
          // todo - add more control over DOM attach points
          $element.find('ul').last().append( $el );
        };
        // remove a dropdown from the nav bar during runtime
        // i.e. upon hash navigation
        this.removeDropdown = function(dropdownId){
          // get a reference to the target dropdown

```

```

    var menuArray = $scope.registeredMenus.filter(function (el){
        return el.uicId === dropdownId;
    });
    var dropdown = menuArray[0];
    // remove and destroy it and all children
    uicDropdownService.remove(dropdown);
    dropdown.iElement.remove();
    dropdown.$destroy();
};

// check for single or array of dropdowns to add
// available on scope for additional invocation flexibility
$scope.addOrRemove = function(dropdowns, action){
    action = action + 'Dropdown';
    if(angular.isArray(dropdowns)){
        angular.forEach(dropdowns, function(dropdown){
            that[action](dropdown);
        });
    }else{
        that[action](dropdowns);
    }
};

// at the mobile width breakpoint
// the Nav Bar items are not initially visible
$scope.isCollapsed = true;
// menu json data if available
$scope.menus = uicNavBarService.getMenus();
// keep track of added dropdowns
// for container level manipulation if needed
$scope.registeredMenus = [];
// listen for minimize event
$scope.$on('header-minimize', function(evt){
    $scope.minimalHeader = true;
});
// listen for maximize event
$scope.$on('header-maximize', function(evt){
    $scope.minimalHeader = false;
});
// handle request to add dropdown(s)
// obj = menu JSON obj or array of objs
$scope.$on('add-nav-dropdowns', function(evt, obj){
    $scope.addOrRemove(obj, 'add');
});

```

```

    // handle request to remove dropdown(s)
    // ids = string or array of strings matching dd titles
    $scope.$on('remove-nav-dropdowns', function(evt, ids){
        $scope.addOrRemove(ids, 'remove');
    });

    // listen for dropdown open event
    $scope.$on('dropdown-opened', function(evt){
        // perform an action when a child dropdown is opened
        $log.log('dropdown-opened', evt.targetScope);
    });

    // listen for dropdown close event
    $scope.$on('dropdown-closed', function(evt){
        // perform an action when a child dropdown is closed
        $log.log('dropdown-closed', evt.targetScope);
    });

    // listen for menu item event
    $scope.$on('menu-item-selected', function(evt, scope){
        // grab the url string from the menu item scope
        var url;
        try{
            url = scope.url || scope.item.url;
            // handle navigation programatically
            $location.path(url);
        }catch(err){
            // $console.log('no url')
        }
    });
}],
link: function(scope, iElement, iAttrs, navCtrl, $transclude){
    // know who the tenants are
    // note that this link function executes *after*
    // the link functions of any inner components
    // at this point we could extend our NavBar component
    // functionality to rebuild menus based on new json or
    // disable individual menu items based on $location
    scope.registeredMenus = uicDropdownService.getDropdowns();
    // Attr API option for sticky vs fixed
    scope.position = (iAttrs.sticky == 'true')
        ? 'navbar-fixed-top'
        : 'navbar-static-top';
    // get theme css class from attr API if set
    scope.theme = (iAttrs.theme) ? iAttrs.theme : null;;
}

```

```
        // send compiled/linked element back to ctrl instance
        navCtrl.init( iElement );
    }
};
}});
```

---

Our NavBar container component directive definition has quite a lot of logic and functionality necessary to cover all the APIs in our list of requirements, and the functionality included doesn't come close to all of the possibilities for this kind of component in terms of styling, display, placement, transformations and contents. The real goal here is to represent a broad display of how a container component directive can accomplish tasks in different ways that can be generalized to any specialized container component.

## Component Lifecycle APIs

Given all the different methods of including user accessible APIs including attributes, data, and events including destruction, the categorization of these in an understandable way can be challenging compared to REST or other server side APIs. AngularJS core directive APIs are mostly just attribute arguments. AngularJS service APIs are categorized as arguments, returns, methods, and events. This works well during app or component instantiation, but that doesn't differentiate the lifecycle of the component all very well. So another useful way of categorizing client-side code APIs can be via page, application, and component lifecycle events including instantiation or configuration, runtime, and destruction. With AngularJS, the end of instantiation and beginning of runtime would be the point at which the entire directive-controller hierarchy has been compiled and linked with the DOM and code can be executed in an AngularJS `.run()` block. At this point it is also useful to remind that the innermost AngularJS directives complete their post-linking phase before the outer directives similar to the DOM event capture and bubble model.

## Instantiation APIs

We cover four of the API requirements via custom HTML attribute including setting an initial minimal header display (menus hidden) via `minimal="true"` and adding the home URL via `home-url="[URL]"` in the isolate scope option of the directive definition. The styling theme and sticky classes are evaluated in the linking function since setting the correct value requires some if-else logic that the scope object is not set up to handle.

We also grab any bootstrapped menu JSON data using the NavBar service and populate the `$scope.menus` object which `ngRepeat` iterates over to instantiate dropdowns. So the API for the user is to populate `window.UIC.header` within script tags with the proper JSON on initial page load. Conversely, we grab any inner HTML supplied by the user which is presumed to be either dropdown or menu item custom elements and quasi-transclude them as the NavBar contents. So looking at this from a lifecycle perspective, our component configuration APIs include four custom attributes, a global data object, and custom inner HTML.

Other items to note that take place during NavBar instantiation include passing the instantiated jQuery element reference back to the NavBar controller from the linking function, and doing the same for all of the instantiated inner dropdowns to the NavBar scope instance. This is done to provide the controller and scope instance with the necessary information for some of the more powerful runtime event APIs

## Runtime APIs

If all we wanted to do was configure and instantiate a static navigation header that had certain styling and contents upon page load we could easily get by with what Bootstrap has to offer. The real power of a framework like AngularJS comes into play after the page has loaded and rendered. AngularJS gives us many useful tools for bringing life to a page component during the time that the user is interacting with it. In any single page application that avoids full page reloads it is wasteful to have to completely remove and recreate page components. In the case of client-side page navigation, it is generally more efficient to modify the navigation component with only what needs to be added or removed, shown or hidden as the visitor moves around the application. So, naturally this is what the bulk of our source code lines in our NavBar directive definition are concerned with.

To handle a good example set of what one might want a navigation container component to do, we have included several `$scope.$on()` event listeners with the event names and arguments as the APIs that other components or the containing application can fire. The first two, “header-maximize” and “header-minimize” will do just what they say in order to show or hide the component contents. We have also added a couple empty event handlers for “dropdown-opened” and “dropdown-closed”. These would presumably be filled in with certain NavBar reactions to certain navigation areas chosen- the point being to again show how the container and contained components can communicate their state in a decoupled fashion.

We have also included a “menu-item-selected” listener that has an over simplified handler for changing the DOM location path variable. In a production version of this this type of component it would likely communicate with a client side router service to handle the real navigation since making direct changes at the page or application level should be avoided by inner UI components.

Saving the best for last, we have two event listeners that listen for requests to add or remove inner dropdown menus on the fly. Active management of a container component’s contents is the essence of why we bother to componentize and a DOM container element like a `<nav>`. An application level controller should know about it’s current location within the application navigation possibilities, and should be able to order the navigation UI component what menus and links to display based on the current app location. An example would be to add and remove dropdowns that an application thinks would be most relevant to a certain user on a certain part of the app. So our NavBar component provides “add-nav-dropdowns” and “remove-nav-dropdowns” listener APIs.

These event listeners call a multipurpose `$scope.addOrRemove()` function that examines the arguments for dropdown ids or objects and an action which in term forwards to appropriate private controller function for the actual handling. The purpose of the this function is to provide a layer of flexibility for the caller’s arguments while keeping things DRY.



If the code for the `addDropdown()` function looks similar to how we manually instantiate directives for unit tests, its because it is. In order to add dropdowns after the fact, we must create a new, isolate scope, thus the call to `$scope.$root.$new()` rather than just `$scope.$new()`. The latter would create a full child scope with access to values on the current scope- very bad. Next we attach the menu item data and invoke the `$compile()` service using the custom element markup for the dropdown components and link it to our newly created isolate scope all in one shot using the lovely functional JavaScript syntax `val = fn()`. The return value is the `jQuery` or `Angular.element()` wrapped reference for the DOM fragment representing the new dropdown which is then appended to the last position of the last `<ul>` element in the current `NavBar` template. Recall that during `NavBar` instantiation we specifically passed the “linked” element reference back to the `NavBar` controller from the linking function allowing us access to the `NavBar` element instance from the controller.

The `removeDropdown()` function illustrates different but equally important AngularJS concepts. This function takes the unique access id (title string) assigned to a dropdown, returns its scope reference in the array of registered dropdowns, and calls `uicDropdownService.remove()` to unregister the dropdown. Additionally the `jQuery` command that detaches a fragment from the DOM and destroys its bindings, `element.remove()`, is called, and `scope.destroy()` is called to remove that scope plus all menu item child scopes and bindings. The main takeaway of this example is proper *manual* cleanup after runtime changes. Failure to remove bindings and other references to orphaned objects prevents JavaScript from allowing these objects to be garbage collected contributing to memory leakage. This is easy to overlook given that most day to day AngularJS operations include automatic binding removal. *JavaScript is not Java.*

## NavBar API Documentation

When it comes time to add the `uicNavBar` container component to our published component pallet we might add some API documentation like the following:

---

COMPONENT DIRECTIVE: `uicNavBar`, `<uic-nav-bar>`

USAGE AS ELEMENT:

```

<uic-nav-bar
  minimal="[true|false]" show or hide menu and dropdown contents
  home-url="[URL string]" set URL of anchor tag that wraps the logo
  sticky="[true|false]" position the NavBar as fixed or static
  theme="[class name string]" optional theme CSS class name
>
  Transclusion HTML, can include <uic-dropdown-menu>, <uic-menu-item>
</uic-nav-bar>

```

#### ARGUMENTS:

Param	Type	Details
minimal	boolean	hide or show contents
homeUrl	string	URL that wraps the logo image
sticky	boolean	position at top of viewport or page
theme	string	additional theme class name
transclusion content	HTML fragment	any HTML including menus, dropdowns

#### EVENTS:

Name	Type	Trigger	Args
'header-minimize'	\$on()	N/A	Event Object
'header-maximize'	\$on()	N/A	Event Object
'add-nav-dropdowns'	\$on()	N/A	Event Object, JSON Data Object
'remove-nav-dropdowns'	\$on()	N/A	Event Object, ID string
'dropdown-opened'	\$on()	N/A	Event Object
'dropdown-closed'	\$on()	N/A	Event Object
'menu-item-selected'	\$on()	N/A	Event Object, TargetScope

## NavBar Unit Test Coverage

Our NavBar container component example is the most sophisticated component directive in terms of being able to manage and react to changes of its contents. The NavBar is currently managing

two levels of menu component hierarchy including runtime dropdown insertions and removals. Naturally, the unit test coverage must be more involved as well, in order to prevent defect regressions as future additional functionality is added, and we have only scratched the surface as far as the possibilities go.

### 6.13 Global Navigation Container Unit Test Coverage

---

```
// As AngularJS and the example UI components built upon it are continuously
// evolving, please see the GitHub repo for the most up to date code:
// https://github.com/dgs700/angularjs-web-component-development
// Unit test coverage for the NavBar UI container
describe('My NavBar component directive', function () {
    var $compile, $rootScope, $scope, $element,
        element, $event, $_uicNavBarService, $log;

    // create some HTML to simulate how a developer might include
    // our menu item component in their page that covers all of
    // the API options
    var tpl =
        '<uic-nav-bar' +
        '  minimal="false"' +
        '  home-url="http://www.david-shapiro.net"' +
        '  sticky="true"' +
        '  theme="default">' +
        '    <uic-dropdown-menu' +
        '      text="Another DD"' +
        '    >' +
        '      <uic-menu-item' +
        '        text="First Item"' +
        '        url="http://david--shapiro.blogspot.com/"' +
        '      >' +
        '    </uic-dropdown-menu>' +
        '    <uic-menu-item' +
        '      text="Link Only"' +
        '      url="http://david--shapiro.blogspot.com/"' +
        '    ></uic-menu-item>' +
        '</uic-nav-bar>';
    // this tpl has some attr API values set to non-defaults
    var tpl2 =
        '<uic-nav-bar' +
        '  minimal="true"' +
        '  home-url="http://www.david-shapiro.net"' +
        '  sticky="false"' +
```

```

    ' theme="light-blue">' +
    ' <uic-dropdown-menu' +
    '   text="Another DD"' +
    ' >' +
    '   <uic-menu-item' +
    '     text="First Item"' +
    '     url="http://david--shapiro.blogspot.com/"' +
    '   >' +
    ' </uic-dropdown-menu>' +
    ' <uic-menu-item' +
    '   text="Link Only"' +
    '   url="http://david--shapiro.blogspot.com/"' +
    ' ></uic-menu-item>' +
    '</uic-nav-bar>';

// tpl with no HTML contents for testing JSON population
var tplJson =
  '<uic-nav-bar' +
  '  minimal="false"' +
  '  home-url="http://www.david-shapiro.net"' +
  '  sticky="true">' +
  '</uic-nav-bar>';

// an array of dropdowns
var header = [
  {"Products": [
    {
      "text": "Scrum Manager",
      "url": "/products/scrum-manager"
    }, {
      "text": "AppBeat",
      "url": "/products/app-beat"
    }, {
      "text": "Solidify on Request",
      "url": "/products/sor"
    }, {
      "text": "IQ Everywhere",
      "url": "/products/iq-anywhere"
    }, {
      "text": "Service Everywhere",
      "url": "/products/service-everywhere"
    }
  ]},
],

```

```

    {"Company": [
      {
        "text": "Contact Us",
        "url": "/company/contact"
      }, {
        "text": "Jobs",
        "url": "/company/jobs"
      }, {
        "text": "Privacy Statement",
        "url": "/company/privacy"
      }, {
        "text": "Terms of Use",
        "url": "/company/terms"
      }
    ]
  }
];
// a single dropdown object
var dropdown = {"About Us": [
  {
    "text": "Contact Us",
    "url": "/company/contact"
  }, {
    "text": "Jobs",
    "url": "/company/jobs"
  }, {
    "text": "Privacy Statement",
    "url": "/company/privacy"
  }, {
    "text": "Terms of Use",
    "url": "/company/terms"
  }
]
};

// manually initialize our component library module
beforeEach(module('uiComponents.navbar'));
// make the necessary angular utility methods available
// to our tests
beforeEach(inject(function (_$compile_, _rootScope_, _$log_) {
  $compile = _$compile_;
  $rootScope = _rootScope_;
  $log = _$log_;
  // note that this is actually the PARENT scope of the directive

```

```

    $scope = $rootScope.$new();
  }));
  // 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();
    });
  }

describe('NavBar component configuration APIs', function () {
  describe('configuration defaults', function () {
    beforeEach(function () {
      compileDirective();
    });
    it('should show all contents if API attribute "minimal" is not true',
      function () {
        // .ng-hide will be set on hidden menu groups
        expect($element.find('ul.ng-hide').length).toBe(0);
      });
    it('should set the brand logo URL to the API attribute "home-url"',
      function () {
        expect($element.find('a.navbar-brand').attr('href'))
          .toContain('www.david-shapiro.net');
      });
    it('should fix position the nav bar if API attr "sticky" is true',
      function () {
        // can only determine that the "navbar-fixed-top" class is set
        expect($element.hasClass('navbar-fixed-top')).toBe(true);
      });
    it('should contain dropdowns and menu components as inner html',
      function () {
        //there are 2 menu items and 1 dropdown in this test template
        expect($element.find('li.uic-menu-item').length).toBe(2);
        expect($element.find('li.uic-dropdown').length).toBe(1);
      });
  });
});

```

```

    });

describe('configuration API alternatives', function () {
    beforeEach(function () {
        // now we are using the second test template
        compileDirective(tpl2);
    });
    it('should hide all contents if API attribute "minimal" is true',
        function () {
            // the 2 <ul> elements should now have .ng-hide set
            expect($element.find('ul.ng-hide').length).toBe(2);
        });

    it('should static position the navbar if API attr "sticky" is falsy',
        function () {
            // can only determine that the "navbar-static-top" class is set
            expect($element.hasClass('navbar-static-top')).toBe(true);
        });
    it('should add a theme class if API attr "theme" equals "classname"',
        function () {
            // the "light-blue" attr val should be added as a class
            // on the root element
            expect($element.hasClass('light-blue')).toBe(true);
        });
});

describe('configuration API via JSON', function () {
    beforeEach(inject(function (uicNavBarService) {
        $_uicNavBarService = uicNavBarService;
        // manually add menu data to the nav bar service
        $_uicNavBarService.addMenus(header);
        compileDirective(tplJson);
    }));
    it('should contain dropdowns and menu components provided as JSON',
        function () {
            // there are 9 total menu items in the test data
            expect($element.find('li.uic-dropdown > ul > li')
                .length).toBe(9);
        });
});
});

```

```

describe('Runtime event APIs', function () {
  var scope;
  beforeEach(inject(function (uicNavBarService) {
    $_uicNavBarService = uicNavBarService;
    // manually add menu data to the nav bar service
    $_uicNavBarService.addMenus(header);
    compileDirective(tplJson);
    // get access to the actual controller instance
    scope = $element.isolateScope();
    // create a fake click event
    $event = $.Event( "click" );
  }));
  it('should hide contents on "header-minimize"', function () {
    // default state is to show all contents
    expect($element.find('ul.ng-hide').length).toBe(0);
    $rootScope.$broadcast('header-minimize');
    scope.$digest();
    // both template <ul>s should now have .ng-hide set
    expect($element.find('ul.ng-hide').length).toBe(2);
  });
  it('should show contents on "header-maximize"', function () {
    // first we need to explicitly hide contents since that
    // is not the default
    $rootScope.$broadcast('header-minimize');
    scope.$digest();
    expect($element.find('ul.ng-hide').length).toBe(2);
    // now broadcast the API event
    $rootScope.$broadcast('header-maximize');
    scope.$digest();
    // .ng-hide should now be removed
    expect($element.find('ul.ng-hide').length).toBe(0);
  });
  it('should add a dropdown on "add-nav-dropdowns"', function () {
    // upon initialization there should be 2 dropdowns with
    // 9 menu items total
    expect($element.find('li.uic-dropdown').length).toBe(2);
    expect($element.find('li.uic-dropdown > ul > li').length).toBe(9);
    // broadcast the API event with data
    $rootScope.$broadcast('add-nav-dropdowns', dropdown);
    scope.$digest();
    // now there should be 3 dropdowns and 13 menu items
    expect($element.find('li.uic-dropdown').length).toBe(3);
  });
});

```



```

    expect($element.find('li.uic-dropdown > ul > li').length).toBe(13);
  });
  it('should remove a dropdown on "remove-nav-dropdowns"', function () {
    // upon initialization there should be 2 dropdowns with
    // 9 menu items total
    expect($element.find('li.uic-dropdown').length).toBe(2);
    expect($element.find('li.uic-dropdown > ul > li').length).toBe(9);
    // broadcast the API event with data
    $rootScope.$broadcast('remove-nav-dropdowns', 'Products');
    scope.$digest();
    // now there should be 1 dropdowns with 4 menu items
    expect($element.find('li.uic-dropdown').length).toBe(1);
    expect($element.find('li.uic-dropdown > ul > li').length).toBe(4);
  });
  it('should log dropdown-opened on "dropdown-opened"', function () {
    spyOn($log, 'log');
    // grab a reference to a dropdown and its scope
    var elem = $element.find('li.uic-dropdown > a');
    var targetScope = elem.isolateScope();
    // simulate opening it
    elem.trigger('click');
    // make sure the event handler gets the correct scope reference
    expect($log.log)
      .toHaveBeenCalledWith('dropdown-opened', targetScope);
  });
  it('should log dropdown-closed on "dropdown-closed"', function () {
    spyOn($log, 'log');
    // grab a reference to a different dropdown and its scope
    var elem = $element.find('li.uic-dropdown > a').last();
    var targetScope = elem.isolateScope();
    // open the dropdown
    elem.trigger('click');
    // then close it again
    elem.trigger('click');
    // make sure the event handler gets the correct scope reference
    expect($log.log)
      .toHaveBeenCalledWith('dropdown-closed', targetScope);
  });
  it('should log the URL on "menu-item-selected"', function () {
    spyOn($log, 'log');
    // get a menu item reference plus scope and url if any
    var menuItem = $element.find('li.uic-dropdown > ul > li');

```

```

    var itemScope = menuItem.scope();
    var url = itemScope.item.url;
    // manually $emit a menu-item-selected event
    itemScope.selected($event, itemScope);
    // only testing menu items w/ urls since those without should
    // be selectable anyhow
    if(url) expect($log.log).toHaveBeenCalledWith(url);
  });
});
});

```

---

## NavBar Unit Test Notes

As you can see, full API coverage requires a significant number of tests. Just as some of the event APIs are currently just stubs, so are the corresponding tests that look for a logged message. The point is that it is a best practice to develop the unit tests along side any code that implements a published API. As real production responses are added to the event callbacks, so should the test change accordingly.



**BEST PRACTICE** Develop API test coverage during development, not after.

Also note that unit testing cannot be used to effectively determine if contents are actually hidden or not. All that can actually be determined is whether or not the CSS class, `.ng-hide`, has been add to or removed from the element in question. The same goes for other CSS styling classes. There is still the indeterminate dependency that the correct CSS rule is applied by the browser rendering engine. The point here is that the inclusion or exclusion of the class is the real concern at this level of testing. Correct visual display falls into the domain of integration testing.

When testing a hierarchical set of directives that have isolate scopes, some of which are manually created in the `beforeEach()` block, and others created automatically by a child directive, finding the correct scope to manipulate can be a challenge. In addition, being able to properly inspect the tested scopes can be difficult given that most of the default console output is truncated strings. A helpful hint for inspection is to run the tests via a configuration that uses Chrome and to click the big “Debug” button. This opens a second browser widow whose developer tools console will display the normal object hierarchy for any scope that is logged to console. Just be sure to close that window before the next test run or Karma will hang.

If the inspected scope does not have the expected properties, it is likely not the scope the compiled template was linked with. The next step is to grab the jQuery reference for the directive element and call `$element.isolateScope()` which will likely have the expected properties. Do not forget to call `$scope.$digest()` on the returned scope just to be sure all the bindings are up to date.

Another item to note is that for API events and calls that set a component state back to its default, the component must be set to some non-default state first which is why the events API tests require many more lines of code. For example, since the default for the NavBar is to display its contents, they must be explicitly hidden as part of the test in order for the test to be valid.

Finally, in wrapping up the comments on this set of API unit coverage tests, you may notice a lot of jQuery looking code snippets, which is fine. Actual AngularJS implementations discourage the use of jQuery coding style because it reduces the declarative nature of the source code, and adds unnecessary DOM coupling and event binding boilerplate. However, it is important to remember the purpose of the code that constructs the unit tests. The purpose is to build and configure the test scenarios for the components by hand (imperatively). This is unavoidable, and in your source code, AngularJS is doing this behind the scenes anyhow. The point here is not to obsess about writing your test code “the AngularJS way”. Use whatever means to create valid and reliable tests and move on. You may also notice that not every controller, link or scope function block has an associated unit test. There are many of the opinion that they should, which holds much more weight when covering integrated application source code. However, with discreet portable, reusable and publishable UI components, the primary concern is that API functioning and consistency is preserved while the internal implementations may change quite a bit, and this is where the focus and effort of unit test coverage should be.

## **Additional NavBar Development**

Dropdowns and menu items only scratch the surface of the potential content UI components the NavBar might manage. In a full production version we could and should add a search component with type ahead and dropdown suggestions, and a sign-in / sign-up widget. Depending on the needs of a particular application we would likely need to add several more event and event handler APIs. The possibilities are numerous as long as they don’t venture into functionality that should be contained or managed by applications or other, more appropriate UI components.

## **Summary**

The goals of chapter 6 were to follow a methodical strategy in building out a set of hierarchical container UI component directives in a manner that resembles the processes used in real world situations. We started with a hypothetical set of business requirements and designs, then followed time-saving strategies to quickly get to implementation start points by using existing open-source code and not reinventing the wheel.

Our implementation of the requirements included a working hierarchy of container and contained UI component directives including a menu item component with the versatility to exist within a dropdown component, a navigation bar component, or elsewhere since there are no outside dependencies, the dropdown component with the same versatility, and a navigation bar component with sophisticated content management capability. We did not stop with the working code,

but included a full set of publishable API documentation and API unit test coverage that will significantly increase the maintainability of our code.

Given that one of the primary themes of this book is to understand web UI component architecture and implementation as applied in real-world scenarios, there are still steps that must happen in order to move our working component code off of our development laptops and to a place, packaged and ready, for our customers to download and start using.

In Chapter 7 we will set up an example automated build and delivery system for our components using some of the more popular open source tools among web UI developers including Grunt/Gulp, Travis CI (continuous integration), and Bower package management. We will also explore techniques for integrating our new AngularJS components into existing web applications built on older frameworks with the focus on replacing bloated and duplicate code from the inside out.

# Chapter 7 - Build Automation & Continuous Integration for Component Libs

So you've succeeded in creating this fantastic set of UI components that encapsulate JavaScript, CSS, and HTML using AngularJS directives. Now what? They look and work great on our local development machine, but that is not the solution to our original business requirement. Our component library needs to be in current use by our target audience of web developers and designers, and there is a big gap between "looking good" on our development laptop and looking good in a production website or web application.

## Delivering Components To Your Customers

We need to consider issues of performance, quality, delivery, and maintenance. Performance includes the obvious steps of minifying and concatenating our source files in the way that provides the best load and evaluation performance as possible for the end user. Quality includes the obvious steps of unit, integration, and end-to-end testing as part of any release to ensure that the APIs remain consistent. Delivery includes all the steps the customer must perform in order to find, access, download, and integrate our library components into their application including package management. Finally, we need to consider how to manage our ever-growing source code as we add new components to the library. Attempts to perform these steps by hand, for anything more than a single component, will quickly become a developer's nightmare.

In the bad old days of software development, the typical (waterfall) release cycle took anywhere from 6 to 18 months from requirements to delivery. These periods would include, in sequence, UML modeling of the requirements, complete source code development, quality assurance processes, compiling, building, packaging and shipping. If requirements changed mid-cycle, often times this meant resetting the development cycle back to square one and starting over.

With the rise of the commercial Internet, delivery methods and timeframes began to change. Customers started downloading packages, updates, and documentation rather than waiting for them to arrive in the mail. So called "agile" methodologies attempted to shorten the development-delivery cycle from 3 weeks to 3 months. The era of formal scrum and extreme programming methods arose to whip lazy engineers into shape. As delivery cycles shortened, reasonable expectations were quickly replaced by the need for instant gratification.

Major software libraries and frameworks that now power the Internet have, at the absolute worst, a nightly build (24 hour delivery cycle). The norm is continuous integration and delivery. This typically

means that as a developer commits a chunk of source code to the repository, an automated build and delivery process is triggered automatically. If that source code is flawed, but somehow manages to pass the full test suite, it still can wind up in the hands of the end user who performs another round of QA by default. Thus, most frameworks and libraries have older “stable branches” and newer “unstable branches”. The former is considered generally safe for production dependency, whereas, the latter is considered “use at your own risk” code.

In this chapter, we will look at a subset of the many tools that are used to automate the build and delivery process for JavaScript based libraries, and provide an example of one working continuous delivery system for our existing set of UI components. We will actually be automating two different build processes. One of them will be the development build process that takes place on our local development laptops. The other one will be the build process that takes place in the “cloud” when any development team member commits code into the main repository branch.

## Source Code Maintenance?

Source code maintenance refers to all of the technics and best practices that enable the product or package to be easily added to, or upgraded by, developers. In previous chapters, we’ve already covered many of these technics for the JavaScript source code ad nauseam including dependency injection, encapsulation, declarative programming, liberal source documentation, DRY, and comprehensive unit test coverage. These technics can be categorized as relating to the individual developer.

Many enterprise organizations skip these some or all of these practices in the interest of delivering a product a few days or weeks earlier. This is the recipe for an eventual massive lump of jQuery spaghetti that periodically must be dumped in the trash and development restarted from scratch. In my many years in Silicon Valley, I have yet to see any major technology organization conduct their development otherwise. The sources of this are typically non-technical marketing bozos in executive management who make hubris driven technical decisions for the engineering organization.

Almost as destructive to good source code maintenance are the outdated ivory tower architects and engineering managers of “proprietary software” who still think JavaScript is not a real programming language. They tend to build the engineering environments around increasingly arcane and inappropriate tools like Java, Oracle, and CVS/SVN. In their quest to abstract away JavaScript in a manner that emulates the traditional object-oriented environments, they gravitate toward tools like GWT and ultimately the same massive, unmaintainable JavaScript hairball.

## Git and GitHub

Before I digress too far, lets back up and talk about source code maintenance in multi-developer environments. Until now, the maintenance technics discussed have been restricted to those under the control of a single developer. There is no debate that concurrent versioning management is a necessity. The versioning tools chosen need to facilitate, not hinder, rapid, iterative development by a team of developers. The tool of choice in the AngularJS world is Git. The Git server of choice

is GitHub. The source code for this book is located in a public repository on GitHub. An in depth discussion about Git and how to best use it is beyond the scope of this book, and moving forward, a normal user level knowledge of Git will be assumed.

The open-source ecosystem on GitHub has benefits beyond facilitating geographically distributed development teams. Public repositories on GitHub can take advantage of things such as package and plugin registration systems including Bower, NPM, Grunt, and many more. If you produce a package, framework, or library that is useful to others in front-end development, as we are trying to do in this book, with a simple JSON config file, the library can be registered on Bower for others to easily locate and install including all dependencies. Projects hosted on GitHub can also take advantage of continuous integration services such as Travis-CI, auto dependency version checking and more.

Git is rapidly replacing Subversion and other proprietary repository management systems because it emphasizes frequent creation and destruction of feature branches, commit tagging, forking, and pull requests. SVN emphasizes “main branch” development and is better suited of longer cycle releases. Git is better suited for rapid, distributed development while preventing junior developers from committing destructive code to the main branch. In many high profile open source projects anyone can check out or fork the repository, fix a bug or make an improvement, and issue a “pull request”. If the committed code is deemed of value by the core team of committers, the code is then “pulled” into the main repository. If a developer makes enough positive contributions to the project, they can often earn their way onto the core team.

## Tooling

In front-end development, new tools arise every month. We will focus on those that are most popular and useful and include discussion on those that are gaining in popularity and likely to replace certain tools already entrenched. The list includes:

- Server-side package management - Node and NPM
- Task runners for automation - Grunt and Gulp plus common tasks for:
- JavaScript code linting
- HTML to JavaScript file includes
- CSS builds from LESS or SASS source
- JavaScript and CSS concatenation
- JavaScript and CSS minification
- Test runners
- File watchers
- Client-side package management - Bower
- Continuous Integration in the Cloud - Travis-CI

## UI Library Focused

Our goal will be to discuss how these tools can facilitate build and delivery of a component library similar to jQuery UI or Bootstrap. The vast majority of scaffolds, and project starters for AngularJS and other JavaScript framework projects are geared heavily towards single page applications. However, available scaffolds and starters for component libraries are almost non-existent.

One of the unique needs in this area is the ability for the user to download a build that includes just a subset of the library or even an individual component rather than the whole thing. It is rare when a web application or website will need to use every component of a library. Having just the necessary subset with required dependencies can significantly improve loading performance.

Our goal will not be to present comprehensive coverage of the above tools, preferring instead to cover the general concepts given that the tools themselves come and go, and there are many ways to implement them for any given situation. Any provided CLI commands assume Mac or Linux as the OS platform. The project used for inspiration in this chapter is Angular-UI Bootstrap <https://github.com/angular-ui/bootstrap><sup>23</sup>.

## Project Directory Structure

Before we start our tour through the laundry list of the popular open source tools for maintaining a UI component library, we need to take a quick detour and consider a directory structure that facilitates optimum organization of the source code.

In our previous examples, as we began initial UI component development, we used a very flat structure that grouped files by either type or purpose. If you recall the Karma configuration file from Chapters 5 and 6, we were including all of the JavaScript files in one directory and all of the unit test files in another. Grouping files by type and purpose is best suited for a single page application directory structure. If you look at the directory structure for the Angular-seed application, you will notice this type of organization:

<https://github.com/angular/angular-seed#directory-layout><sup>24</sup>

---

<sup>23</sup><https://github.com/angular-ui/bootstrap>

<sup>24</sup><https://github.com/angular/angular-seed#directory-layout>



# Directory Layout

```
app/                --> all of the files to be used in production
  css/              --> css files
    app.css         --> default stylesheet
  img/              --> image files
  index.html        --> app layout file (the main html template)
  index-async.html  --> just like index.html, but loads js files asynchronously
  js/               --> javascript files
    app.js          --> application
    controllers.js  --> application controllers
    directives.js   --> application directives
    filters.js      --> custom angular filters
    services.js     --> custom angular services
  partials/         --> angular view partials (partial html)
    partial1.html
    partial2.html

test/               --> test config and source files
  protractor-conf.js --> config file for running e2e tests with protractor
  e2e/              --> end-to-end specs
    scenarios.js
  karma.conf.js     --> config file for running unit tests with karma
  unit/             --> unit level specs/tests
    controllersSpec.js --> specs for controllers
    directivesSpec.js  --> specs for directives
    filtersSpec.js     --> specs for filters
    servicesSpec.js    --> specs for services
```

Screen grab of the AngularJS' team's recommended directory structure for an AngularJS app.

Up until now, we've only developed JavaScript source, HTML templates, CSS, and unit tests for a total of four UI components. However, in a realistic situation, this sort of library could have a couple dozen or more components. This sort of structure can become difficult to work with in situations where we need to hunt for and open all of the files for a single component in an IDE. In fact, just knowing which files belong to a particular component can cause a headache.

An alternative approach for organizing a library of components would be to group all of the files related to a particular component by directory named for the component. For example, we could create a directory like the following for the source files of each component:

**SourcefileorganizationintheNavbardirectoryforallfilesassociatedwiththeNavbarcomponent**

---

```
ProjectRoot/  
  Navbar/  
    Navbar.js  
    Navbar.tpl.html  
    Navbar.less  
    test/  
      NavbarSpec.js  
    docs/  
      Navbar.demo.html  
      Navbar.demo.js
```

---

This way it becomes obvious to anyone new to the project which files are associated with which component. Additionally, when it comes time to automate the build workflow in a way that we can optionally build just one or two components instead of the entire library, we can better ensure that only the source files under the “Navbar” directory are used when “Navbar” is an argument to the build command.

When you consider that a highly encapsulated UI component that includes its own JavaScript behavior, CSS styling, and HTML templating, is a mini application conceptually, maintaining source files in a directory per component seems logical. This is not to say that every AngularJS component library project should necessarily be structured this way. Chosen structure should reflect a project's needs, goals, and development roadmaps. Some UI component libraries will be coupled with a large application, and others may be mixed in with non-UI components. There are a couple suggestions for help in matching a directory structure with requirements that can be offered. First, consider how quickly an engineer new to the project can figure out what can be found where, and second, check out all the great project starter templates contributed by the AngularJS community at Yeoman:

<http://yeoman.io/><sup>25</sup>

For the remainder of this chapter we will be organizing the source code by directory for each component which will be reflected in the build workflow configuration files.

---

<sup>25</sup><http://yeoman.io/>

## Node.js and NPM

Node.js is a server side container and interpreter for running JavaScript analogous to how the browser is the container on the client-side. In fact, Node.js is based on the same JavaScript engine as Chrome and is therefore actually coded primarily in C++ rather than JavaScript as many people assume. What makes Node.js different from most server-side language interpreters or runtime environments is it is designed to execute multiple commands asynchronously (non-blocking) via a single OS thread rather than synchronously (blocking). In this way, idle time for any thread of execution is minimized and computer CPU is used in a much more efficient manner. A Node.js webserver will handle multiple requests per process; whereas, other web servers will typically dedicate an entire process to handling one request.

Also, contrary to popular assumption is that Node.js was not created by front-end developers who wanted to use JavaScript to program on servers, but rather was picked as the console interface language for Node.js because it was the language in widest use that offered asynchronous and functional programming syntax. If such syntax were offered by Ruby, PHP, Python, PERL, or Java, then one of those languages might have been used instead.

Another advantage of Node.js for front-end developers is that it (along with NoSQL databases) allows one to perform full-stack development in one language. This is significant since it is highly likely that at some point (Web 4.0?), the network and all of the AJAX or REST boilerplate will be abstracted away via platforms like Meteor.js in much the same manner that AngularJS abstracts away the need for any manual data and event binding today. If you are relatively new to web development and plan to be in this business for the next several years, then it is highly advisable to spend professional development time learning Node.js rather than Ruby, PHP or Python even though more mature server-side MVC frameworks exist for them.

The default libraries included with Node.js allow easy access and manipulation of all I/O subsystems on the OS for \*nix based machines. This includes file and network I/O such as setting up, managing, and tearing down HTTP web servers, or reading and writing files to disc.

That said, make sure Node.js, the most recent major release number (with stable minor version number), is installed globally on your development machine <http://nodejs.org/><sup>26</sup>. OS X should already have it, and most Linux's should have easy access via a package manager. My condolences to those behind corporate firewalls with draconian port policies. It may be a bit more difficult obtaining the latest necessary packages.

Mac users without Node.js should install the latest Xcode from the app store, and then install a Ruby package manager called Homebrew. From there, you run:

```
~$ sudo brew install git
```

```
~$ sudo brew install node
```

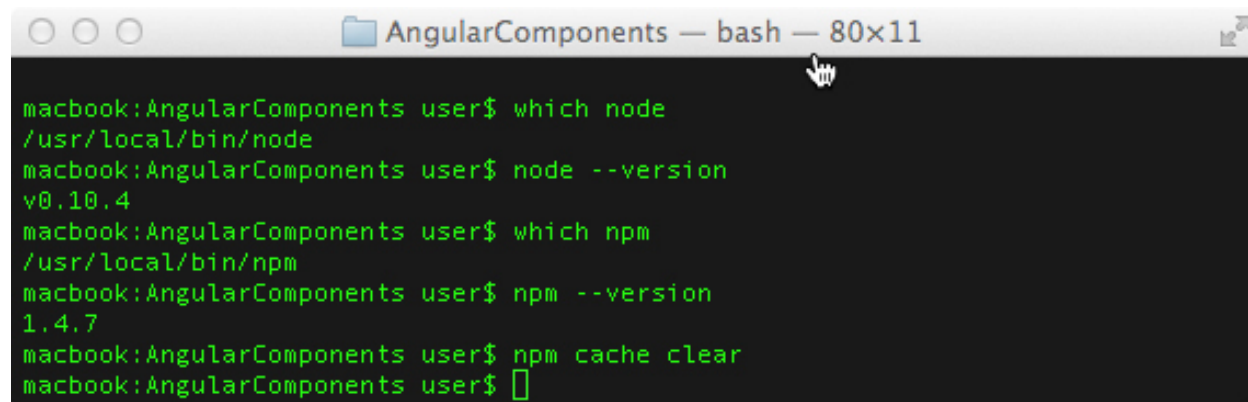
Finally make sure NPM (Node Package Manager) is updated to the latest.

---

<sup>26</sup><http://nodejs.org/>

```
~$ sudo npm update -g
```

You should be able to run `user$ which npm` and see it in your path.

A terminal window titled "AngularComponents — bash — 80x11" showing a series of commands and their outputs. The commands are: `which node`, `node --version`, `which npm`, `npm --version`, and `npm cache clear`. The outputs are: `/usr/local/bin/node`, `v0.10.4`, `/usr/local/bin/npm`, `1.4.7`, and a blank line respectively. The prompt is `macbook:AngularComponents user$`.

```
macbook:AngularComponents user$ which node
/usr/local/bin/node
macbook:AngularComponents user$ node --version
v0.10.4
macbook:AngularComponents user$ which npm
/usr/local/bin/npm
macbook:AngularComponents user$ npm --version
1.4.7
macbook:AngularComponents user$ npm cache clear
macbook:AngularComponents user$
```

Commands to check for Node and NPM on Mac and Linux plus cache clearing

Having the up-to-date version of NPM and clearing NPM's cache will improve the chances of successful downloads of the necessary NPM modules including Grunt, Karma, Bower, and Gulp.

Within any given project directory that utilizes NPM modules, there should be a **package.json** file in the root that holds references to all of the projects NPM dependency modules at or above a particular version. There will also be a sub directory called `node_modules/` where all the local project dependencies are stored. This directory name should be included in all configuration files where directories and files are excluded from any processing such as `.gitignore`. We do not want to package or upload the megabytes of code that can end up in this directory, nor do we want to traverse it with any task commands that search for JavaScript files with globs such as `/*.js` as this can take some time and eat up a lot of CPU needlessly.

By listing all the NPM module dependencies in `package.json`, the project becomes portable by allowing the destination user to simply run `~$ npm install` in the project root directory in order to get all of the required dependencies. All necessary documentation plus available modules can be found at the NPM website:

<https://www.npmjs.org/><sup>27</sup>

We will defer listing our `package.json` file example until towards the end of this chapter, as its contents will make a lot more sense after we have covered all the third party packages that it references.

## Task Runners and Workflow Automation

Build automation has been a vital part of software development for decades. Make was the first such tool used in Unix / C environments to simplify stringing together several shell commands required

---

<sup>27</sup><https://www.npmjs.org/>

to create tmp directories, find dependencies, run compilation and linking, copy to a distribution directory, and clean temporary files and directories created during the process.

Ant was the next tool that arose along with Java in the 90s. Rather than programmatic statements, Ant offered XML style configuration statements meant to simplify creating the build file. A few years after Ant came Maven, which offered the ability to download build files and dependencies from the Internet during the build process. Maven was also XML based and Java focused. Maven inspired several more build automation runners specific to languages such as Rake for Ruby and Phing for PHP.

In the JavaScript world, Grunt is the task automation tool that is currently dominant. There are thousands of available tasks, packaged as NPM modules on the web, which can be configured and invoked in a project's Gruntfile.js. There is another task runner called Gulp that is aiming to knock Grunt off of its throne. It will likely succeed for some good reasons.

Because at the time of this writing, Gulp and the available Gulp tasks are not yet numerous and mature enough to satisfy our build automation needs at a production level of quality, we will explore build automation for our UI component libraries using both tools.

## Grunt - The JavaScript Task Runner

Grunt is acquired as an NPM module. It should be installed globally in your development environment:

```
~$ sudo npm install -g grunt-cli
```

It should be available in your command path after installation. If it is not, it may be necessary to locate the installed executable and symlink it to a name in your command path. Documentation can be found on the Grunt project website:

<http://gruntjs.com/><sup>28</sup>

Registered Grunt tasks can be searched at:

<http://gruntjs.com/plugins><sup>29</sup>

When the grunt command is run in the root directory of a project, it looks for a Node.js file called Gruntfile.js which contains its instructions. There are some templates available on the Grunt project website that can be used with the command grunt-init to generate a Gruntfile based on answers to various questions, however, it is probably faster, easier and more informative to locate GitHub repos of similar projects and inspect the contents of their Gruntfiles.

Gruntfiles are Node.js JavaScript files. In them, there are typically three primary commands or sections:

---

<sup>28</sup><http://gruntjs.com/>

<sup>29</sup><http://gruntjs.com/plugins>

```
grunt.loadNpmTasks( 'task-name' );
grunt.initConfig({configurationObject});
grunt.registerTask( 'task-name', [ 'stepOne', 'stepTwo', 'stepThree', ... ] );
```

Third party tasks are loaded. A (sometimes very large) JSON object is used to reference and configure the loaded tasks, and then one or more task names are defined with an array of steps that map to the definitions in the configuration object. These task names can then be used in a file watcher or on the command line:

```
project_dir$ grunt task-name
```

### 7.0 Example Gruntfile.js with a Single Task

---

```
module.exports = function (grunt) {
  // load 3rd party tasks
  grunt.loadNpmTasks('grunt-contrib-uglify');
  // configure the tasks
  grunt.initConfig({
    // external library versions
    ngversion: '1.2.16',
    bsversion: '3.1.1',
    // make the NPM configs available as vars
    pkg: grunt.file.readJSON('package.json'),
    // provide options and subtasks for uglify
    uglify: {
      options: {
        banner: '/*! <%= pkg.name %> */\n',
        mangle: false,
        sourceMap: true
      },
      build: {
        src: 'js/MenuItem.js',
        dest: 'build/MenuItem.min.js'
      },
      test: {
        files: [{
          expand: true,
          cwd: 'build/src',
          src: '*.js',
          dest: 'build/test',
          ext: '.min.js'
        }]
      }
    }
  })
}
```

```
});  
// define our task names with steps here  
grunt.registerTask('default', ['uglify:test']);  
};
```

---

The code listing above contains an example of a Gruntfile.js with a single task “default” which will run on the command line simply with the grunt command and no extra arguments. It runs a configured subtask of “uglify” which is a commonly used JavaScript minifying tool.

In practice, a Gruntfile with a single task would be pointless since the whole point of build automation is to perform several tasks. The point of the above listing is to illustrate the three main sections of the Gruntfile and what typically goes in them. The first step in debugging a Gruntfile that fails is to check each of these three sections and make sure all the names jibe.

### 7.1 Example Gruntfile.js with a Multiple Tasks

---

```
module.exports = function (grunt) {  
  // load 3rd party tasks  
  grunt.loadNpmTasks('grunt-contrib-watch');  
  grunt.loadNpmTasks('grunt-contrib-concat');  
  grunt.loadNpmTasks('grunt-contrib-copy');  
  grunt.loadNpmTasks('grunt-contrib-jshint');  
  grunt.loadNpmTasks('grunt-contrib-uglify');  
  grunt.loadNpmTasks('grunt-html2js');  
  grunt.loadNpmTasks('grunt-import');  
  grunt.loadNpmTasks('grunt-karma');  
  grunt.loadNpmTasks('grunt-conventional-changelog');  
  grunt.loadNpmTasks('grunt-ngdocs');  
  grunt.loadNpmTasks('grunt-contrib-less');  
  
  // configure the tasks  
  grunt.initConfig({  
    // external library versions  
    ngversion: '1.2.16',  
    bsversion: '3.1.1',  
    // make the NPM configs available as vars  
    pkg: grunt.file.readJSON('package.json'),  
    // provide options and subtasks for uglify  
    uglify: {  
      options: {  
        banner: '/*! <%= pkg.name %> */\n',  
        mangle: false,  
        sourceMap: true  
      }  
    }  
  });  
};
```



```

    },
    build: {
      src: 'js/MenuItem.js',
      dest: 'build/MenuItem.min.js'
    },
    test: {
      files: [{
        expand: true,
        cwd: 'build/src',
        src: '*.js',
        dest: 'build/test',
        ext: '.min.js'
      }]
    }
  },
  html2js: {
    options: {},
    dist: {
      options: {
        // no bundle module for all the html2js templates
        module: null,
        base: '.'
      },
      files: [{
        expand: true,
        src: ['template/*.html'],
        ext: '.html.js'
      }]
    },
    main: {
      options: {
        quoteChar: '\'',
        module: null
      },
      files: {
        'build/tmp/Dropdown.tpl.js': ['html/Dropdown.tpl.html'],
        'build/tmp/MenuItem.tpl.js': ['html/MenuItem.tpl.html'],
        'build/tmp/Navbar.tpl.js': ['html/Navbar.tpl.html'],
        'build/tmp/SmartButton.tpl.js': ['html/SmartButton.tpl.html']
      }
    }
  }
}

```



```
    },

    watch: {
      files: [
        'js/*.js'
      ],
      tasks: ['uglify']
    },

    import: {
      options: {},
      dist: {
        expand: true,
        cwd: 'js/',
        src: '*.js',
        dest: 'build/src/',
        ext: '.js'
      }
    },

    less: {
      dev: {
        files: {
          'css/ui-components.css': 'less/ui-components.less'
        }
      },
      production: {
        options: {
          cleancss: true
        },
        files: {
          'css/ui-components.css': 'less/ui-components.less'
        }
      }
    }
  }
});

grunt.registerTask('css', ['less:production']);
grunt.registerTask('default', ['html2js']);
grunt.registerTask('test', ['html2js:main', 'import:dist', 'uglify:test']);
};
```

---

The code listing above is closer to a real world Gruntfile with multiple 3rd party task imports, a much longer configuration object, and multiple registered task names with steps at the end. The task names in bold should be recognizable as typical steps performed somewhere in the build workflow including testing, concatenating, minifying, linting, etc.

The last line in the listing shows three task steps in sequence. The first two steps each write out files to disc before beginning the next step. For small projects, the time this takes will not be noticeable. However, for large libraries or applications with hundreds to thousands of source files, the disc I/O time can take minutes, which becomes impractical to run whenever a single source file is saved.

This is one of the major complaints of those who have created Gulp as an alternative. Rather than outputting directly to disc, the default output of any Gulp task is a data stream that can be “piped” directly to the next task instead. The output data from a task or series of tasks will only be written to disc when explicitly commanded. In addition, tasks can be executed concurrently or in parallel if one is not dependent on the previous. Complete, in memory processing for very large projects can result in time savings of several orders of magnitude. We will further discuss the differences, pros, and cons of both Grunt and Gulp in later sections.

## Useful Tasks for a Component Lib Workflow

The build tasks we may want to automate as part of our workflow typically fall into a few different categories.

### Tasks in Sequence

We likely want to perform some preprocessing of our source files to inline the templates and generate associated CSS. Then we would want to check the quality of these files, and generate any associated documentation. Finally, we want to create performance-optimized versions of these files suitable for distribution to our customers. Optionally, we might want to create a second layer of automation by setting up file watchers to perform certain tasks when the files are saved. For example, we would want any CSS output immediately when we save a .less file, and we would want to run unit tests and source linting before deciding to make a Git commit of our recent batch of changes.

### Tasks Before and After Committing

Looking at this from another angle (pun intended), when we add or modify a component, we want to run some tasks (unit tests and linting), then determine whether or not to commit our code to the main repository based on the outcome. If our code quality is up to par, then we want to proceed with the tasks in the workflow that move the component code toward production including minification, documentation, integration testing and so on.

## Preprocessing Source Files with LESS, html2js and JavaScript Imports

Excluding unit tests and configuration files, most UI developers find it easiest to edit HTML, CSS and JavaScript as separate files. This includes all of the preprocessor languages such as CoffeeScript, TypeScript, and Dart for JavaScript, Jade, HAML, and Slim for HTML, and LESS or SASS for CSS.

Source code preprocessors, in some cases, solve some serious limitations inherent in the language they compile to. Others are nothing more than syntactic sugar for developers who feel comfortable working with syntax similar to their server-side language. For our purposes, we will remain preprocessor agnostic with the exception of CSS.

CSS preprocessor languages add a tremendous amount of value beyond what CSS has to offer. CSS has no notion of variables, functions, or nested rule sets. Because of this CSS for large projects can end up with a lot of duplicated rules and ultimately poor network loading performance and very difficult maintenance. The manner in which they can increase styling consistency and reduce code duplication is beyond the scope of our topic. However, one nice thing about LESS and SASS is that plain old CSS syntax is perfectly valid, so there is no reason for us not to include one as part of our example build workflow which will include CSS source code in LESS files.

If you navigate to plugins page of the Grunt project website and search for “less” a list of many modules appear. The one at the top “grunt-contrib-less” is what we want, as it has the code that compiles LESS to CSS. From the root directory of our project, we download this module with the following command:

```
npm install grunt-contrib-less --save-dev
```

and then add the following line to our Gruntfile.js:

```
grunt.loadNpmTasks('grunt-contrib-less');
```

The NPM command installs the plugin local to the project under the node\_modules/ directory. The --save-dev option will add a dependency reference to the project’s package.json file. In this way, the project can be portable without having to include anything under the node\_modules/ directory in the tar ball or Git glob, which can be thousands of files. All one needs at the destination is the package.json file in the project root, and then they can run npm install to get all the dependency modules downloaded to the node\_modules/ directory.

Next, we would add something like the following to our Gruntfile.js in order to provide the LESS and Grunt with a useful configuration:

## 7.2 Adding a LESS task to a Gruntfile.js

---

```
grunt.initConfig({
  less: {
    dev: {
      files: [
        {
          expand: true, // Enable dynamic expansion.
          cwd: 'src/', // Src matches are relative to this path.
          src: ['/*.less'], // Actual pattern(s) to match.
          dest: 'build/src/', // Destination path prefix.
          ext: '.css' // Dest filepaths will have this extension.
        }
      ],
    },
    production: { ... }
  }
});
```

---

The files object contains information that will search for all .less files under the src/ and any sub directories as source to compile, and the output will be placed in build/src/. The dev:{} object defines one subtask for the “less” task that we can call from the command line as:

```
~$ grunt less:dev
```

It is typical to have more than one subtask configuration for different purposes such as development vs. production or distribution. The dev task outputs individual .css files for development purposes, whereas, a production subtask might use a different source .less file that has @includes for several LESS modules to build the entire project CSS.

Besides the actual LESS compiling, the above configuration example can be generalized to most other Grunt task configurations for file input and output.

As mentioned previously, it is preferable to maintain separate source files for HTML templates and JavaScript for ease of development, spotting bugs, and cooperation from our IDE’s. However, because tight integration and a high level of encapsulation is desired for our components, we want to inline the HTML template as JavaScript into the same file as the primary directive definition object for our custom HTML element that encloses the component in the DOM.

This is a bit different from the way templates are handled in libraries like (Angular) UI Bootstrap, and AngularStrap. Those libraries are intended to be forked, extended and modified to the user’s liking, so the HTML templates are compiled and (optionally) included as JavaScript in separate AngularJS modules along side the component modules that contain the JavaScript for the directives and services. The need to swap templates is a common occurrence. Because our strategy is focused on creating UI components meant to enforce styling for consumption by junior developers and

designers in large organizations, we need to treat JavaScript and the associated HTML as one. If browser technology allowed the same for CSS, we would include that as well.

What this means is that we will need a task(s) that will stringify the HTML in the template, add it as the value to a var definition so it becomes legal JavaScript, and then insert it as a “JavaScript include” into the file with the directive definition. JavaScript, unlike most other languages, still does not have the notion of “includes”. The hope is that it will be an added feature for ES7, but for today, we need some outside help.

The following series of screen grabs illustrate how the source files that we would edit in our IDE need to be transformed into a consolidated file that we would test in our browser and with unit coverage.

```
1 <li class="uic-dropdown">
2   <a dropdown-toggle
3     ng-bind-html="dropdownTitle"><b class="caret"></b></a>
4   <ul class="dropdown-menu"
5     ng-if="jsonData">
6     <li ng-repeat="item in menuItems"
7       ng-class="disablabale"
8       ng-init="disablabale=(item.url)?null:disabled">
9       <a ng-href="{{ item.url }}"
10        ng-bind="item.text"
11        ng-click="selected($event, this)"></a>
12     </li>
13   </ul>
14   <ul class="dropdown-menu"
15     ng-if="!jsonData"
16     ng-transclude></ul>
17 </li>
```

Dropdown.html source file template that is edited as HTML

```

1  tpl = '<li class="uic-dropdown">' +
2      '    <a dropdown-toggle' +
3      '      ng-bind-html="dropdownTitle"><b class="caret"></b></a>' +
4      '    <ul class="dropdown-menu"' +
5      '      ng-if="jsonData">' +
6      '        <li ng-repeat="item in menuItems"' +
7      '          ng-class="disabable"' +
8      '          ng-init="disabable=(item.url)?null: \'disabled\'">' +
9      '            <a ng-href="{ item.url }"' +
10         '              ng-bind="item.text"' +
11         '              ng-click="selected($event, this)"></a>' +
12         '          </li>' +
13         '        </ul>' +
14         '        <ul class="dropdown-menu"' +
15         '          ng-if="!jsonData"' +
16         '          ng-transclude></ul>' +
17         '</li>';

```

Generated Dropdown.tpl.js after conversion to valid JavaScript by a Grunt task

```

3  (function(){
4      'use strict';
5
6      var tpl = '';
7      //@import "../build/src/Dropdown/Dropdown.tpl.js";
8
9

```

The editable source file, Dropdown.js with a special “//@import” comment

```

3  (function(){
4      'use strict';
5
6      var tpl = '';
7      tpl = '<li class="uic-dropdown">' +
8          '<a dropdown-toggle' +
9          '    ng-bind-html="dropdownTitle"><b class="caret"></b></a>' +
10         '<ul class="dropdown-menu"' +
11         '    ng-if="jsonData">' +
12         '    <li ng-repeat="item in menuItems"' +
13         '        ng-class="disablabl"' +
14         '        ng-init="disablabl=(item.url)?null: \'disabled\'">' +
15         '            <a ng-href="{ { item.url } }"' +
16         '                ng-bind="item.text"' +
17         '                ng-click="selected($event, this)"></a>' +
18         '        </li>' +
19         '    </ul>' +
20         '    <ul class="dropdown-menu"' +
21         '        ng-if="!jsonData"' +
22         '        ng-transclude></ul>' +
23         '</li>';
24

```

Generated Dropdown.js after the Grunt task replaces the “import” statement with the template JavaScript

By maintaining editable versions of the html template and JavaScript without the in-lined template, and as both valid HTML and JavaScript, we are free to develop these files without any nasty clutter or screams from our IDE about invalid code syntax.

However, this can create the undesirable situation where we are not able to perform the fast loop of “edit-and-reload”, the instant gratification that front-end developers love. We get around this obstacle by setting up a “file watcher” as another Grunt task that will monitor file changes in the source directories, and run the conversion tasks automatically. We can even go so far as to enable a “live-reload” as a Grunt task so we do not need to constantly hit the browser “reload” button in order to see the fruits of our work upon each edit.

First let’s cover getting the source file processing and merging configured in Grunt. At the time of this writing, no Grunt plugins did exactly what we needed as shown above. However, some came very close. “grunt-html2js” will stringify an HTML file into an AngularJS module meant to be loaded into \$templateCache at runtime, and “grunt-import” will insert the contents of one JavaScript file into another, replacing an “@include filename.js” statement. The issue with the latter is that having “@include filename.js” sitting in a JavaScript source file is not valid JavaScript and results in serious complaints from most IDEs. We would need to tweak the task so that the import statement can sit behind a JavaScript line comment as `//@import filename.js` which would be replaced with the

template JavaScript.

Tweaking these plugins to do what we need involves a couple alterations to the Node.js source code for the tasks located under:

```
project_root/node_modules/[module_name]/tasks/
```

It is best not to alter these tasks in place unless you fork the module code, make the changes, and register as new node modules, otherwise Node.js dependency management for the project will break. A short term solution would be to copy the module directory to a location outside of node\_modules/, make the edits, and import them into the grunt configuration using `grunt.loadTasks('task_path_name')`. Since Node.js development is beyond the scope of this book, please see the book's GitHub repo for the actual code alterations. We will add a bit of description to these tasks and reference them as `html2jsVar` and `importJs`.

Let's look at the Grunt configuration that watches all of our source file types where they live, and runs the development tasks upon any change:

### 7.3 Grunt Source Code Development Tasks

---

```
// prepare encapsulated source files
grunt.loadNpmTasks('grunt-contrib-less');
grunt.loadTasks('lib/project/grunt-html2js-var/tasks');
grunt.loadTasks('lib/project/grunt-import-js/tasks');
// configure the tasks
grunt.initConfig({
  ...
  html2jsVar: {
    main: {
      options: {
        quoteChar: '\'',
        module: null
      },
    },
    files: [{
      expand: true, // Enable dynamic expansion.
      cwd: 'src/', // Src matches are relative to this path.
      src: ['/*.*html'], // Actual pattern(s) to match.
      dest: 'build/src/', // Destination path prefix.
      ext: '.tpl.js' // Dest filepaths will have this extension.
    }]
  },
  importJs: {
    dev: {
      expand: true,
      cwd: 'src/',
```



```

        src: ['/*.js', '!/test/*.js'],
        dest: 'build/src/',
        ext: '.js'
    }
},
less: {
    dev: {
        files: [
            {
                expand: true, // Enable dynamic expansion.
                cwd: 'src/', // Src matches are relative to this path.
                src: ['/*.less'], // Actual pattern(s) to match.
                dest: 'build/src/', // Destination path prefix.
                ext: '.css' // Dest filepaths will have this extension.
            }
        ],
    }
},
watch: {
    source: {
        files: ['src/**/*.js', 'src/**/*.html', '!src/test/**/*.js'],
        tasks: ['dev'],
        options: {
            livereload: true
        }
    },
},
});

grunt.registerTask('dev', ['html2jsVar:main', 'importJs:dev', 'less:dev']);

```

---

The code listing above contains the contents of our Gruntfile.js (in bold) that we have added. We can now run:

```
~$ grunt watch:source
```

...from the command line. As long as we point our test HTML page to the correct JavaScript and CSS files under the project\_root/build/src/ directory, we can edit our uncluttered source files, and immediately reload the merged result to visually test in a browser. Running the above command even starts a LiveReload server. If we enable the test browser to connect to the live-reload port, we don't even need to hit "refresh" to see the result of our source edit.

Notice in two of the "files" arrays we include '!/test/\*.js'. This addition prevents any watches or action on the unit test JavaScript files. As the UI component library source code grows, we need to

limit the source files operated on in order to maintain the ability to perform instant reloads. The processing time in milliseconds is sent to stdout, so we can keep tabs on how long the cycle takes as components are added. If the library size becomes large enough, it may be necessary to further restrict the file sets in the Grunt config, or to switch to a streaming system like Gulp. Also, note that if for some reason we do not wish to start a “watch”, we can call `grunt dev` at the command line to run source merging once after an edit.

## Source Code Quality Check with jsHint and Karma

Now that we have the portion of our build workflow configured for ongoing source file editing, it is time to think about the next step in the build workflow process. Assume we have modified or added a feature or component, which has involved edits to several source files. Everything looks good in our test browser, and we are thinking about committing the changes to the main repository branch. What can we do to automate some quality checks into our pre-commit workflow?

The first and most obvious task is running the unit tests that we have been writing as we have been developing the component. You have been doing TDD haven’t you? Conveniently, there is a Karma plugin for Grunt that can be installed. It does exactly the same thing as using Karma directly from the command line as we have discussed in previous chapters.

In previous chapters we had Karma configured to run using Chrome, and to watch the source files directly in order to re-run upon any change. When running Karma from Grunt, it is usually preferable to use a headless browser (PhantomJS) and let Grunt handle any file watches. The Karma Grunt task allows you to use different Karma configuration files, or to use one file and override certain options. Maintaining multiple Karma config files would likely become more painful than just overriding a few options, which is how we will handle these to changes in our Gruntfile.js. One rather irritating discovery about running Karma from Grunt is that all of the Karma plugins referenced must also be installed locally to the project, rather than globally which is common for developers working on many projects\*.

Another common quality check is code “linting”. The most popular linter for JavaScript at the time of this writing is jsHint. jsHint is somewhat of an acrimonious fork of jsLint, as many in the JavaScript community felt the latter was becoming more opinionated than practical\*.

jsHint can be configured to check for all kinds of syntax and style issues with JavaScript source code. There are two main areas of focus when code linting. The first is concerned with error prevention. JavaScript was originally design to be very forgiving of minor mistakes in order that non-engineers could tinker with it while creating web pages. Unfortunately, a uniform feeling of syntax forgiveness is not shared among all browsers. Trailing commas do not cause errors in Firefox or Chrome, whereas they do in Internet Explorer. JavaScript also has syntax rules that do not cause errors in any browser, but have a high probability of leading to errors in the future. Optional semi-colons and braces are the two worst offenders. jsHint can be configured to watch and report many code issues that degrade quality and maintenance like these.

The other area where a tool like jsHint is handy is in multi-developer environments. A major best practice for multi-developer teams is a consistent coding style. A good lead developer will enforce

a code style guide for the team, and adhering to a coding style for almost all open source project contributions is a requirement. Without a consistent coding style, engineers will always do their own thing, and it will become increasingly difficult for one engineer to understand and maintain another engineer's source code. jsHint can be of assistance in this area, at least as far as code syntax is concerned.

Creating a custom configuration for a project's jsHint involves adding a file to the root directory of a project with various "switches" turned on. Below is the .jshintrc file in use for this project at the time of this writing.

#### 7.4 jsHint Configuration JSON - .jshintrc

---

```
{
  "curly": true,
  "immed": true,
  "newcap": true,
  "noarg": true,
  "sub": true,
  "boss": true,
  "eqnull": true,
  "quotmark": "single",
  "trailing": true,
  "strict": true,
  "eqeqeq": true,
  "camelcase": true,
  "globals": {
    "angular": true,
    "jquery": true,
    "phantom": true,
    "node": true,
    "browser": true
  }
}
```

---

Some of switches above may be obvious. Since the topic for this book is not general JavaScript syntax or maintenance, please visit the jsHint project webpage for comprehensive documentation.

<http://www.jshint.com/docs/><sup>30</sup>

Configuration switches can also be specified directly in your Gruntfile.js in the options:{} object. Additionally, you may want to create jsHint subtasks that run it after merging or concatenating JavaScript source files, and there are optional output report formatters that you may prefer. By

---

<sup>30</sup><http://www.jshint.com/docs/>

default, the jsHint grunt task will fail and abort any grunt process on errors unless you set the “force” option to “true”.

An up and coming alternative to jsHint at the time of this writing is ESLint, created by Nicholas Zakas in mid-2013. ESLint includes all of the switches and rules of jsHint, plus a lot more rules organized by purpose. What sets ESLint apart from jsHint (or jsLint) is that ESLint is extensible. jsHint comes with a preset list of linting switches. Additional options or switches must be built into the source code. ESLint, on the other hand, allows you the ability to add additional rules via configuration JSON. More info can be gleaned at: <http://eslint.org/><sup>31</sup>

Putting all of the above together in our Gruntfile.js, we get:

### 7.5 Grunt Pre-Commit Code Quality Tasks

---

```
// check the js source code quality
grunt.loadNpmTasks('grunt-contrib-jshint');
grunt.loadNpmTasks('grunt-karma');
// configure the tasks
grunt.initConfig({
  ...
  karma: {
    options: {
      configFile: 'test/karma.conf.js',
      autoWatch: false,
      browsers: ['PhantomJS']
    },
    unit: {
      singleRun: true,
      reporters: 'dots'
    }
  },
  jshint: {
    options: {
      force: true // do not halt on errors
    },
    all: ['Gruntfile.js', 'src/**/*.js']
  }
});

grunt.registerTask('preCommit', ['jshint:all', 'karma:unit']);
```

---

The additions to our Grunt configuration file above will help to ensure code quality and style conformance prior to making a commit to a team repository, however it is not a substitute for

---

<sup>31</sup><http://eslint.org/>

developer testing in all the major browsers or peer code reviews. It's the minimum that a developer can do to avoid team embarrassment when their commit breaks other parts of the code base.

Before moving on to other essential build workflow automation tasks, it's worth mentioning that until now our tasks have included all source files as input. As component libraries grow and tasks begin to eat up more time and CPU, Grunt gives us the ability to pass in arguments in addition to tasks to run on the command line. If it makes sense, we can pass in the names of individual components for processing, rather than everything at once. By using Grunt variables, templates, and a bit more coding, we can run tasks on just the desired modules and their direct dependencies. This is part of the reason for restructuring the source code directory organization as we did. By calling Grunt with something like the following on the command line we can pass in a component "key=value" (prefixed with a double dash) that can be used within the Gruntfile.js to only select and operate on files and directories of that name:

```
~$ grunt preCommit --component=Dropdown
```

We can then access this in the Gruntfile like this:

```
var component = grunt.config('component');
```

If, on the other hand, will do not need the parameter to be *globally* accessible in our Gruntfile.js, and we just want to pass parameters or arguments into a particular task, we do so with colon notation on the command line:

```
~$ grunt build:navbar
```

```
~$ grunt build:dropdown:menuItem
```

Assuming the colon separated strings are not subtasks of build, they can be accessed *locally* inside the build task function as this.args.

In the next section, we will create an example section of our Gruntfile.js that uses task arguments to create custom builds that include just a subset of the available UI components.

## Performance Optimization with Concat and Uglify

Until now we have explored task automation related to source code development. Let's assume we've developed a new UI component, and we are ready to either commit or merge it into our main Git branch. As is standard these days in web tier development, we should run a build on our local development laptop upon each commit to make sure the build succeeds before pushing our commit to the origin repository server.

If you have a good devOps engineer on the team, she has created for you a development VM (virtual machine) with an environment that is as identical to the production server as possible, using tools like Vagrant. Your build should perform the same steps as closely as possible to those on the team integration server. There will still be differences, such as running time consuming multi-browser e2e, and regression tests.

Regardless, the build should produce production ready files of our component library including necessary dependencies. “Production ready” is defined, at a bare minimum, to be minified JavaScript that is concatenated into as few files as possible to minimize download file size and latency. It can be defined, more robustly, to include all API documentation and demos, version information, change logs, additional unminified files with source maps, etc.

One special requirement, as mentioned earlier, is the ability to create a build that includes just the modules or components that a particular customer may need, so they do not have to depend on file sizes any larger than absolutely necessary. Most major JavaScript component libraries and toolkits now have some form of custom build option. You can create a custom jQuery build by checking out different Git branches of their repository and running a local build. Bootstrap and Angular-UI Bootstrap allow you to create a custom build by selecting or deselecting individual modules via their web interface. Only the collection of selected components and necessary dependencies will be included in the files you receive.

The following additions to our Gruntfile.js include tasks and functions for JavaScript minification, file concatenation, and build options for the entire library or just a sub-set of UI components.

#### 7.6 Minification, Concatenation, and Build Tasks Added to Gruntfile.js

---

```
// configure the tasks
grunt.initConfig({
  // external library versions
  ngversion: '1.2.16',
  components: [],
  //to be filled in by build task
  libs: [],
  //to be filled in by build task
  dist: 'dist',
  filename: 'ui-components',
  filenamecustom: '<%= filename %>-custom',
  // make the NPM configs available as vars
  pkg: grunt.file.readJSON('package.json'),
  srcDir: 'src/',
  buildSrcDir: 'build/src/',
  // keep a module:filename lookup table since there isn't
  // a regular naming convention to work with
  // all small non-component and 3rd party libs
  // MUST be included here for build tasks
  libMap: {
    "ui.bootstrap.custom": "ui-bootstrap-collapse.js",
    "ngSanitize": "angular-sanitize.min.js"
  },
  meta: {
    modules: 'angular.module("uiComponents", [<%= srcModules %>]);',
```

```

    all: '<%= meta.srcModules %>',
    banner: ['/*',
      ' * <%= pkg.name %>',
      ' * <%= pkg.repository.url %>\n',
      ' * Version:
        <%= pkg.version %> - <%= grunt.template.today("yyyy-mm-dd") %>',
      ' * License: <%= pkg.license %>',
      ' */\n'].join('\n')
  },
  uglify: {
    options: {
      banner: '<%= meta.banner %>',
      mangle: false,
      sourceMap: true
    },
    dist: {
      src: ['<%= concat.comps.dest %>'],
      dest: '<%= dist %>/<%= filename %>-<%= pkg.version %>.min.js'
    }
  },
  concat: {
    // it is assumed that libs are already minified
    libs: {
      src: [],
      dest: '<%= dist %>/libs.js'
    },
    // concatenate just the modules, the output will
    // have libs prepended after running distFull
    comps: {
      options: {
        banner: '<%= meta.banner %><%= meta.modules %>\n'
      },
      //src filled in by build task
      src: [],
      dest: '<%= dist %>/<%= filename %>-<%= pkg.version %>.js'
    },
    // create minified file with everything
    distMin: {
      options: {},
      //src filled in by build task
      src: ['<%= concat.libs.dest %>', '<%= uglify.dist.dest %>'],
      dest: '<%= dist %>/<%= filename %>-<%= pkg.version %>.min.js'
    }
  }
}

```

```

    },
    // create unminified file with everything
    distFull: {
        options: {},
        //src filled in by build task
        src: ['<%= concat.libs.dest %>', '<%= concat.comps.dest %>'],
        dest: '<%= dist %>/<%= filename %>-<%= pkg.version %>.js'
    }
}
});

// Credit portions of the following code to UI-Bootstrap team
// functions supporting build-all and build custom tasks
var foundComponents = {};
var _ = grunt.util._;
// capitalize utility
function ucwords (text) {
    return text.replace(/^[a-z])|\s+([a-z])/g, function ($1) {
        return $1.toUpperCase();
    });
}
// uncapitalize utility
function lcwords (text) {
    return text.replace(/^[A-Z])|\s+([A-Z])/g, function ($1) {
        return $1.toLowerCase();
    });
}
// enclose string in quotes
// for creating "angular.module(..." statements
function enquote(str) {
    return "'" + str + "'";
}

function findModule(name) {
    // by convention, the "name" of the module for files, dirs and
    // other reference is Capitalized
    // the nme when used in AngularJS code is not
    name = ucwords(name);
    // we only need to process each component once
    if (foundComponents[name]) { return; }
    foundComponents[name] = true;
    // add space to display name

```



```

function breakup(text, separator) {
    return text.replace(/[A-Z]/g, function (match) {
        return separator + match;
    });
}

// gather all the necessary component meta info
// todo - include doc and unit test info
var component = {
    name: name,
    moduleName: enquote('uiComponents.' + lcwords(name)),
    displayName: breakup(name, ' '),
    srcDir: 'src/' + name + '/',
    buildSrcDir: 'build/src/' + name + '/',
    buildSrcFile: 'build/src/' + name + '/' + name + '.js',
    dependencies: dependenciesForModule(name),
    docs: {} // get and do stuff w/ assoc docs
};

// recursively locate all component dependencies
component.dependencies.forEach(findModule);
// add this component to the official grunt config
grunt.config('components', grunt.config('components')
    .concat(component));
}

// for tracking misc non-component and 3rd party dependencies
// does not include main libs i.e. Angular Core, jQuery, etc
var dependencyLibs = [];
function dependenciesForModule(name) {
    var srcDir = grunt.config('buildSrcDir');
    var path = srcDir + name + '/';
    var deps = [];
    // read in component src file contents
    var source = grunt.file.read(path + name + '.js');
    // parse deps from "angular.module(x,[deps])" in src
    var getDeps = function(contents) {
        // Strategy: find where module is declared,
        // and from there get everything i
        // nside the [] and split them by comma
        var moduleDeclIndex = contents
            .indexOf('angular.module(');
        var depArrayStart = contents
            .indexOf('[', moduleDeclIndex);
    };
}

```

```

    var depArrayEnd = contents
      .indexOf(']', depArrayStart);
    var dependencies = contents
      .substring(depArrayStart + 1, depArrayEnd);
    dependencies.split(',').forEach(function(dep) {
      // locate our components that happen to be deps
      // for tracking by grunt.config
      if (dep.indexOf('uiComponents.') > -1) {
        var depName = dep.trim()
          .replace('uiComponents.', '')
          .replace(/["']/g, '');
        if (deps.indexOf(depName) < 0) {
          deps.push(ucwords(depName));
          // recurse through deps of deps
          deps = deps
            .concat(dependenciesForModule(depName));
        }
        // attach other deps to a non-grunt var
      } else {
        var libName = dep.trim().replace(/["']/g, '');
        if (libName && !_.contains(dependencyLibs, libName)) {
          dependencyLibs.push(libName);
        }
      }
    });
  });
  getDeps(source);
  return deps;
}

grunt.registerTask('build', 'Create component build files', function() {
  // map of all non-component deps
  var libMap = grunt.config('libMap');
  // array of the above to include in build
  var libFiles = grunt.config('libs');
  var fileName = '';
  var buildSrcFiles = [];
  var addLibs = function(lib){
    fileName = 'lib/' + libMap[lib];
    libFiles.push(fileName);
  };
  //If arguments define what modules to build,

```

```

// build those. Else, everything
if (this.args.length) {
    this.args.forEach(findModule);
    _.forEach(dependencyLibs, addLibs);
    grunt.config('filename', grunt.config('filenamecustom'));
// else build everything
} else {
    // include all non-component deps in build
    var libFileNames = _.keys(grunt.config('libMap'));
    _.forEach(libFileNames, addLibs);
}
grunt.config('libs', libFiles);

var components = grunt.config('components');
// prepare source modules for custom build
if(components.length){
    grunt.config('srcModules', _.pluck(components, 'moduleName'));
    buildSrcFiles = _.pluck(components, 'buildSrcFile');
// all source files for full library
}else{
    buildSrcFiles = grunt.file.expand([
        'build/src/*.js',
        '!build/src/*.tpl.js',
        '!build/src/test/*.js'
    ]);
// prepare module names for "angular.module('',[])" in build file
var mods = [];
_.forEach(buildSrcFiles, function(src){
    var filename = src.replace(/^[^\w\/]+/, '')
        .replace(/\.js/, '');
    filename = enquote('uiComponents.' + lcwords(filename));
    mods.push(filename);
})
grunt.config('srcModules', mods);
}

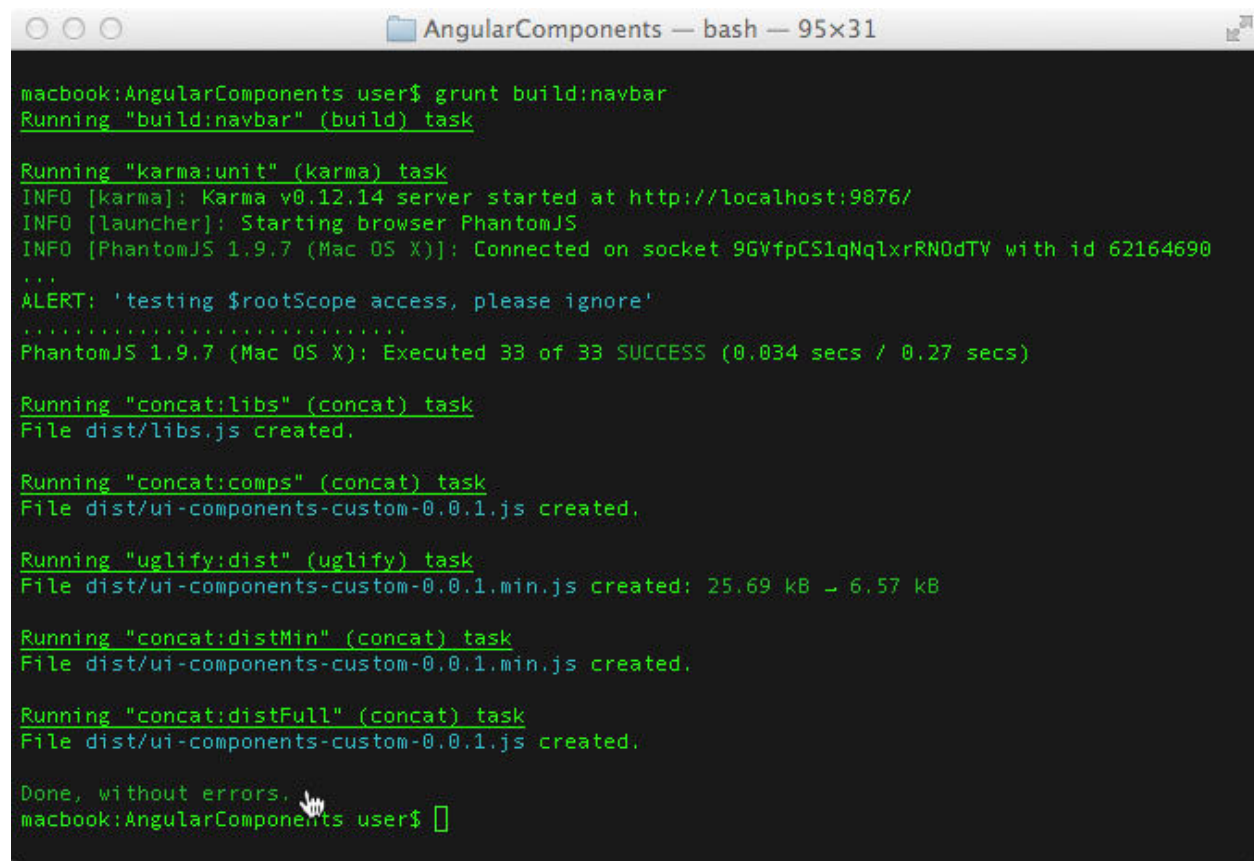
// add src files to concat sub-tasks
grunt.config('concat.comps.src', grunt.config('concat.comps.src')
    .concat(buildSrcFiles));
grunt.config('concat.libs.src', grunt.config('concat.libs.src')
    .concat(libFiles));

```

```
// time to put it all together
grunt.task.run([
  'karma:unit',
  'concat:libs',
  'concat:comps',
  'uglify',
  'concat:distMin',
  'concat:distFull'
]);
});
```

---

If we run a custom build task, and all goes well, we should see something like the following:



```
macbook:AngularComponents user$ grunt build:navbar
Running "build:navbar" (build) task

Running "karma:unit" (karma) task
INFO [karma]: Karma v0.12.14 server started at http://localhost:9876/
INFO [launcher]: Starting browser PhantomJS
INFO [PhantomJS 1.9.7 (Mac OS X)]: Connected on socket 9GVfpCS1qNqlxrRN0dTV with id 62164690
...
ALERT: 'testing $rootScope access, please ignore'
...
PhantomJS 1.9.7 (Mac OS X): Executed 33 of 33 SUCCESS (0.034 secs / 0.27 secs)

Running "concat:libs" (concat) task
File dist/libs.js created.

Running "concat:comps" (concat) task
File dist/ui-components-custom-0.0.1.js created.

Running "uglify:dist" (uglify) task
File dist/ui-components-custom-0.0.1.min.js created: 25.69 kB → 6.57 kB

Running "concat:distMin" (concat) task
File dist/ui-components-custom-0.0.1.min.js created.

Running "concat:distFull" (concat) task
File dist/ui-components-custom-0.0.1.js created.

Done, without errors.
macbook:AngularComponents user$
```

Console output for grunt build:navbar command

The additions to the Gruntfile.js in bold illustrate the complexity of adding custom build options for a subset of UI component to the command line as component name arguments. Much of this code deals with finding the UI components that match the names, parsing components to get the dependency references from the code, and then finding the dependency files, and doing the same recursively until all dependencies are resolved. It's basically the same algorithm that AngularJS

performs in the browser upon load to resolve all of the referenced modules.

## Gruntfile.js Odds, Ends, and Issues

The code above makes significant use of Grunt templates such as:

```
'<%= dist %>/<%= filename %>-<%= pkg.version %>.js'
```

to dynamically fill in the list of input files from the previous task's list of output files. If you are familiar with ERB, Underscore, or Lodash templates, the delimiters are the same. This save us from having to write a lot of code to manage variable input and output file lists.

Even with this code savings, our Gruntfile.js has still grown to hundreds of lines of JavaScript, making it hard for other engineers to grok quickly. There are still many tasks and sub-tasks yet to configure in a “real world” situation that could add a few hundred more lines of code. Fortunately, there is a Grunt plugin called `load-grunt-config` which allows you to break up large Gruntfile.js' by factoring out tasks into their own files. The NPM page for this module shows how to use it.

<https://www.npmjs.org/package/load-grunt-config><sup>32</sup>

For the remainder of this chapter, we will use a single, monolithic Gruntfile.js for concept illustrative purposes, however, if adapting this Gruntfile.js for use with an actual UI component library, factoring out tasks into their own files should be on the *todo* list.

## Custom Build GUIs

Now that we have custom build capability, it's fairly trivial for us to add a web GUI interface so our customers can just check the boxes for the set of UI components needed, and then zip and serve the built files as the response. For a working example, you can check out the AngularUI Bootstrap page (upon which much of the example build task code is based) and select the “Create a Build” option to get an idea. The result will be full and minified module and template files (4 files).

<http://angular-ui.github.io/bootstrap/><sup>33</sup>

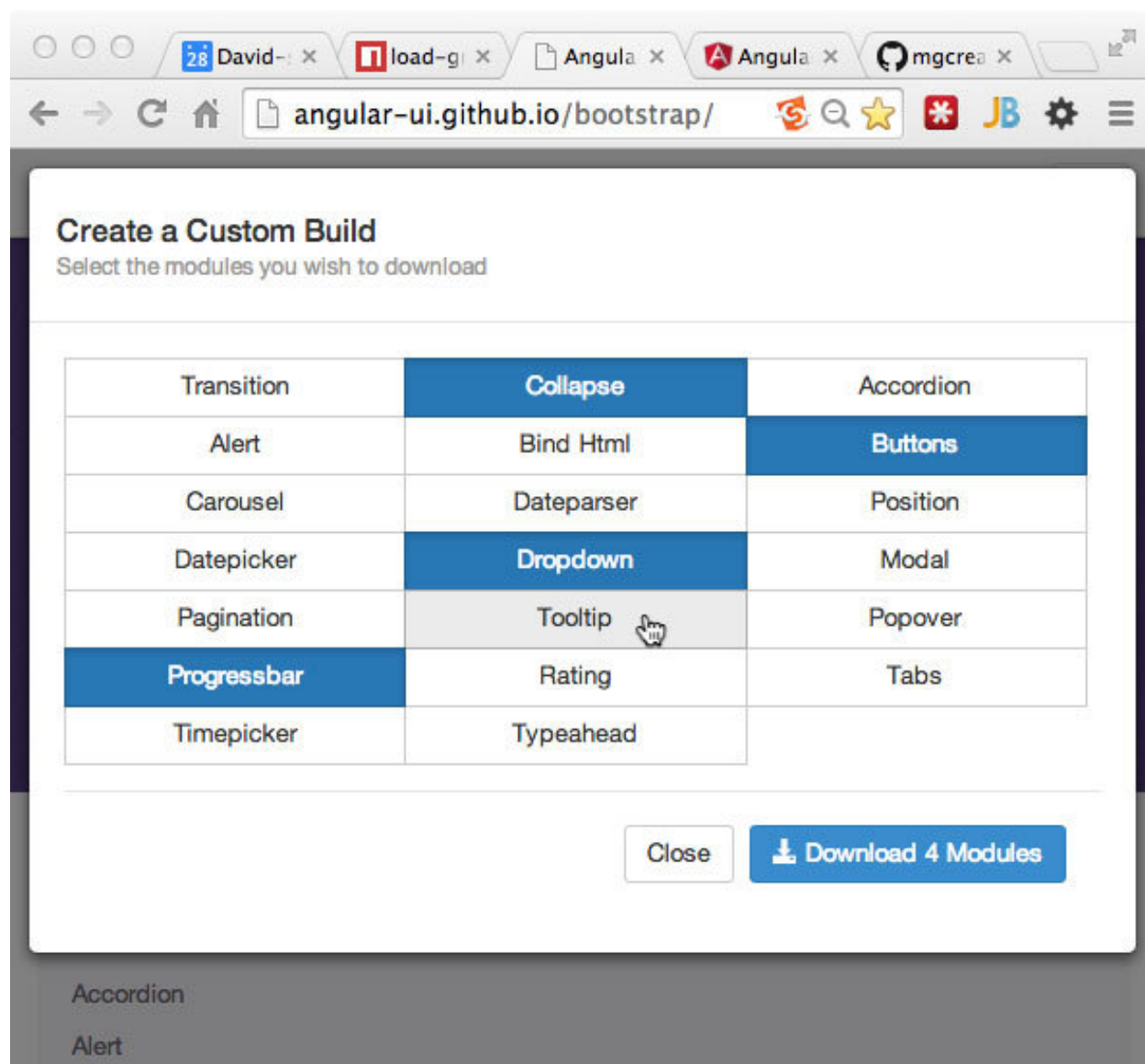
---

<sup>32</sup><https://www.npmjs.org/package/load-grunt-config>

<sup>33</sup><http://angular-ui.github.io/bootstrap/>



Screen grabs of the AngularUI team's custom build GUI



Screen grabs of the AngularUI team's custom build GUI

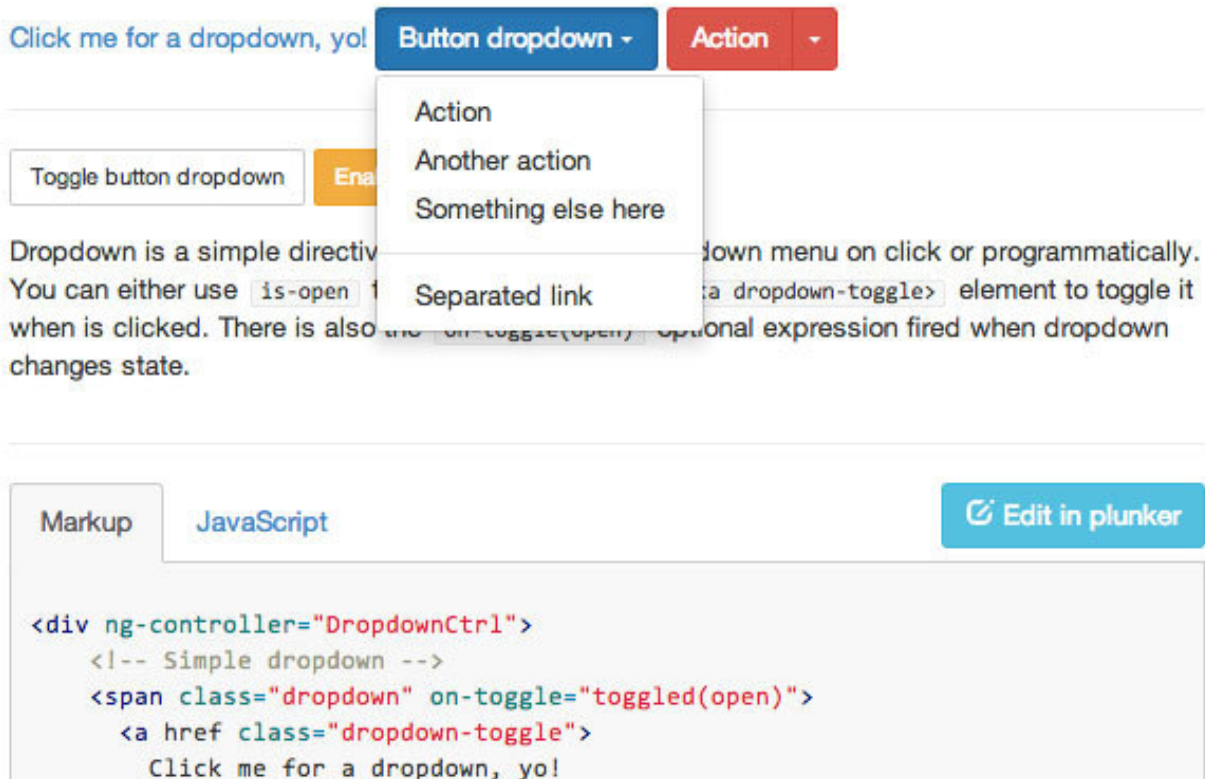
Creating a web interface similar to the example from UI-Bootstrap is the way to go, given that one of the major themes of this book is creating UI component libraries that are as simple as possible for web page designers to integrate and use, making them easy to propagate throughout an enterprise sized organization. We expect that our customers are not all senior level web developers, so we want to avoid having to checkout Git branches and running builds, or having to enter command line arguments (with correct spelling and case) to get a custom build.

Also, as part of the website for our UI component library that customers will interact with, we must include, not just API documentation, but also HTML and JavaScript examples of each component. The examples should visually illustrate the various states and API options of each component in action. The HTML should illustrate where to place the custom elements, and what to place inside



them. Again, using the same AngularUI-Bootstrap page, something like the following, in addition to the API documentation example from the previous chapter, should work:

## Dropdown (ui.bootstrap.dropdown)



Example section from the AngularUI Bootstrap team's site shown their dropdown usage

At the beginning of this chapter, we included a source documentation folder as part of the suggested UI component library source directory structure where the files that illustrate example usage should live. A dropdown.demo.html file and a dropdown.demo.js file would map to the content for the two tabs in the screen grab above.

## Additional Grunt Tasks

At the time of this writing there are exactly 2,928 registered Grunt plugins of which we have worked with 8 including the two that we forked for our own customizations. There are some useful plugins for build automation tasks we have not yet covered including generating documentation, adding change-log entries, version “bumping”, cleaning build artifacts, creating zip files for down load, and more.



Unfortunately we can only dedicate part of a chapter on task automation with Grunt, enough to give a high level overview, to cover task very unique to our needs, and to provide code examples to help you get a start with task and build automation for your own UI component library.

There are a handful of books dedicated entirely to Grunt. However, I believe the best source of information about what you can automate with existing task is to just Google what you need, check out the Grunt plugin page <http://gruntjs.com/plugins><sup>34</sup>, and check out Gruntfile.js' from public repositories of project similar to your own. If time and space permitted, we would also cover the following list of plugins, and hopefully some or all of them will see use in the example GitHub repository for this book by the time it is published (in addition to our own tasks being registered as official Grunt plugins):

- grunt-contrib-clean - delete unneeded build generated files
- grunt-conventional-changelog - maintain a change log from version to version
- grunt-contrib-copy - for miscellaneous file copying to other locations
- grunt-contrib-cssmin - minify the CSS output from LESS source
- grunt-bump - uses Sem(antic)Ver(sioning) to bump up release numbers
- grunt-contrib-livereload - automatically reload test page on file save
- grunt-jsdoc - generate parts of the API docs from jsDoc comments in the source code (possibly extend to add tags that handle non-standard APIs)

## Gulp.js - The Streaming Build System

If you scan the build tasks in our Gruntfile.js, you may notice that in the process of merging templates with code, concatenating it, minifying it, and merging with the concatenated library file, Grunt is reading and writing to disc four times! If our library grows to hundreds of source files, builds can start taking quite a while.

Since about mid-2013, Gulp.js has risen up as an alternative to Grunt, in part, to bypass all of the unnecessary disc I/O. It takes advantage of data streams and “pipes” to stream the output from one task directly to the next task as an input data stream rather than stream the data to the file system in between. I have to admit that when I began reading about Gulp.js, I started having horrid flashbacks to my Unix sysadmin and Java programming days. The upside to those flashbacks was that I already understood pipes from Unix and Streams from Java. Java has extensive libraries for handling data streams of all types. I was surprised to find out that granular manipulation of data streams were not a part of Node.js until version 0.10+.

The following code listing contains a sample gulpfile.js forked from a GitHub Gist by Mark Goodyear:

<https://gist.github.com/markgoodyear/8497946><sup>35</sup>

---

<sup>34</sup><http://gruntjs.com/plugins>

<sup>35</sup><https://gist.github.com/markgoodyear/8497946>

### 7.7 Example gulpfile.js File

---

*// Load plugins*

```
var gulp = require('gulp'),
    sass = require('gulp-ruby-sass'),
    autoprefixer = require('gulp-autoprefixer'),
    minifycss = require('gulp-minify-css'),
    jshint = require('gulp-jshint'),
    uglify = require('gulp-uglify'),
    imagemin = require('gulp-imagemin'),
    rename = require('gulp-rename'),
    clean = require('gulp-clean'),
    concat = require('gulp-concat'),
    notify = require('gulp-notify'),
    cache = require('gulp-cache'),
    livereload = require('gulp-livereload'),
    lr = require('tiny-lr'),
    server = lr();
```

*// Styles*

```
gulp.task('styles', function() {
  return gulp.src('src/styles/main.scss')
    .pipe(sass({ style: 'expanded', }))
    .pipe(autoprefixer('last 2 version', 'safari 5', 'ios 6', 'android 4'))
    .pipe(gulp.dest('dist/styles'))
    .pipe(rename({ suffix: '.min' }))
    .pipe(minifycss())
    .pipe(livereload(server))
    .pipe(gulp.dest('dist/styles'))
    .pipe(notify({ message: 'Styles task complete' }));
});
```

*// Scripts*

```
gulp.task('scripts', function() {
  return gulp.src('src/scripts/*.js')
    .pipe(jshint('.jshintrc'))
    .pipe(jshint.reporter('default'))
    .pipe(concat('main.js'))
    .pipe(gulp.dest('dist/scripts'))
    .pipe(rename({ suffix: '.min' }))
    .pipe(uglify())
    .pipe(livereload(server))
    .pipe(gulp.dest('dist/scripts'))
});
```

```
.pipe(notify({ message: 'Scripts task complete' }));
});

// Images
gulp.task('images', function() {
  return gulp.src('src/images/**')
    .pipe(cache(imagemin({
      optimizationLevel: 3,
      progressive: true,
      interlaced: true })))
    .pipe(livereload(server))
    .pipe(gulp.dest('dist/images'))
    .pipe(notify({
      message: 'Images task complete'
    })));
});

// Clean
gulp.task('clean', function() {
  return gulp.src([
    'dist/styles',
    'dist/scripts',
    'dist/images'
  ], {read: false})
    .pipe(clean());
});

// Default task
gulp.task('default', ['clean'], function() {
  gulp.run('styles', 'scripts', 'images');
});

// Watch
gulp.task('watch', function() {
  // Listen on port 35729
  server.listen(35729, function (err) {
    if (err) {
      return console.log(err)
    }
  });
  // Watch .scss files
  gulp.watch('src/styles/**/*.scss', function(event) {
```

```
    console.log('File ' +
        event.path +
        ' was ' +
        event.type +
        ', running tasks...');
    gulp.run('styles');
});
// Watch .js files
gulp.watch('src/scripts/*.js', function(event) {
    console.log('File ' + event.path + ' was ' +
        event.type +
        ', running tasks...');
    gulp.run('scripts');
});
// Watch image files
gulp.watch('src/images/*', function(event) {
    console.log('File ' + event.path + ' was ' +
        event.type + ', running tasks...');
    gulp.run('images');
});
});
});
```

---

The code listing above has nothing to do with our sample UI component library. It is included to provide a contrast of the syntax between Grunt and Gulp.js. Mark Goodyear also has an excellent Gulp.js introductory blog post at:

<http://markgoodyear.com/2014/01/getting-started-with-gulp/><sup>36</sup>

## Gulp.js Pros

Gulp.js' major selling point is plugins that can input and output data streams which can be chained with pipes, offering much faster build automation for industrial sized projects. Additionally, if the function functionality for a given task already exists as a Node.js library or package, it can be "required()" and used directly, no need to wrap as a grunt-task.

Gulp.js' other selling points are a matter of opinion based on what type of programming and languages you are most comfortable with. Grunt is object "configuration" based, whereas, Gulp.js utilizes attributes of functional and imperative programming to define tasks and workflows. Developers who spend most of the day using Node.js are generally more comfortable with the latter. Those who have used systems such as Ant, Maven, Rake, YML or any other system after the days of Make (which was basically shell programming) are probably more comfortable with the former.

---

<sup>36</sup><http://markgoodyear.com/2014/01/getting-started-with-gulp/>

Another major difference between Grunt and Gulp.js is that by default, dependencies for a Grunt task are executed in sequence by default. Dependencies for a Gulp.js task are executed concurrently, or in parallel, by default. Executing non-interdependent tasks concurrently can shave time off of builds, but requires special handling in Gulp.js with deferreds or callbacks (called “hints”).

Finally, if you cannot find a Gulp.js plugin that does what your Grunt plugin does, there is a Gulp.js plugin that wraps Grunt plugins called `gulp-grunt`.

## Gulp.js Cons

The basic Gulp.js tasks as of this writing seem to be immature. I was unable to get “`gulp-include`” to work with piped output from a `concat` task as it did not know how to resolve the include file’s path. I could only get it to work with the files read from and written to disc which defeats the purpose of Gulp.js. Overall, the core Grunt tasks are much more tried and true.

The documentation for Gulp.js is extremely minimal, I suppose because the assumption is that you should have sufficient knowledge of I/O programming with Node.js. The majority of front-end developers, myself included, are not “intimate” with Node’s inner I/O workings. As much as I wanted to debug the issue with the `gulp-include` plugin, I didn’t know where to start other than taking a deep dive into Node.js’ I/O library docs. As mentioned earlier, at this time, Gulp seems to have a much higher level of accessibility to the Node.js vs. non-Node.js communities.

The Gulp team touts that plugins for Gulp.js are held to “stringent” standards, and it looks like several plugins that have been written for Gulp on their search page are crossed out with red labels indicating that they are duplicates of another plugin or don’t comply with some rule that is required by the Gulp team. If you click the red label, it takes you to a GitHub open-an-issue page where you need to explain why this plugin should be removed from the “blacklist”.

Given that anyone who takes the time to contribute to open-source is doing so on their spare time, and for free, this strikes me as both insulting and elitist. I personally would not bother to spend the time and effort on contributing to this project given the core team’s attitude. I’m including this commentary; not as a rant, but to illustrate that this attitude towards contributions would likely alienate a lot of would-be, plugin contributors.

I personally would like to see stream and pipe capability added to Grunt. Performance for large builds is the one real downside to Grunt from a pragmatic point of view. The other downsides claimed by the Gulp community essentially boil down to opinion.

## Continuous Integration and Delivery

So far we have covered all the necessary tool categories needed from the time we edit a source file to the end result of a built and minified file of our UI component library on our local development environment. Unless we are also the only customer for our library, there is still more to be done in order to get our code into the hands of our customers.

Continuous integration and delivery is considered the current best practice in web application development. Continuous integration (CI) grew out of XP (Extreme Programming) of the late 90s. Today, most organizations practice CI without XP. The idea is to kick off an automated test and build upon a source code commit to the main branch on a central integration server at a minimum. While code in a developer's local environment may test and build without error, unless the environment is identical to build, staging, and production servers, as well as all major browsers, errors may arise in builds performed after the code leaves the developer's laptop. The central build upon commit helps identify any breakage caused by the commit so everyone can see it. More often than not, developers who are either lazy or arrogant feel they can get away with the local quality checks prior to a commit. CI helps catch the breakage and who caused it, so they can be publically shamed and hopefully learn to respect best practices more. Besides the public humiliation for the developer who broke the codebase, CI help prevent nasty regression bugs from making their way into production. These are the defects that can be very difficult to trace back to a source given that in web development multiple languages are employed and must work together to produce what is delivered to the customer.

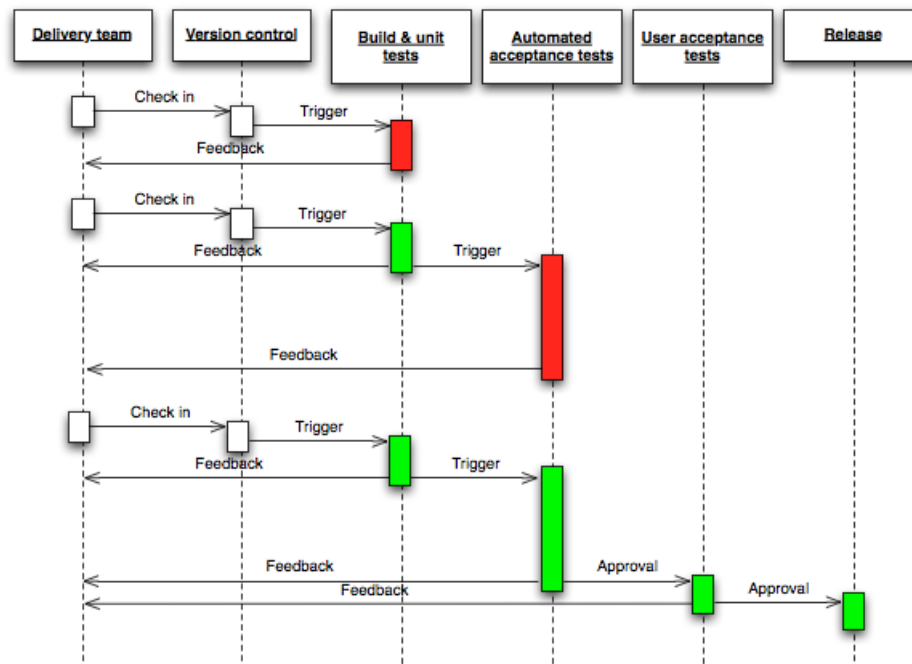
CI usually goes beyond just a central test-then-build process. Once the build is produced, an additional round of integration and end-to-end tests on multiple servers, and in all supported browsers happens. Following that is typically some sort of a deployment to a staging environment, or even to production as a "snapshot" release. Some of the more popular integration applications include Jenkins, Continuum, Gump, TeamCity, and the one we will be using, Travis-CI.

For a more in depth introduction to CI check out the Wikipedia page:

[http://en.wikipedia.org/wiki/Continuous\\_integration](http://en.wikipedia.org/wiki/Continuous_integration)<sup>37</sup>

---

<sup>37</sup>[http://en.wikipedia.org/wiki/Continuous\\_integration](http://en.wikipedia.org/wiki/Continuous_integration)



A flow representation of continuous delivery

## Travis-CI

Travis-CI is one of the many cloud based CI systems that have sprung up in recent years. It provides runtime environments for multiple versions of most popular languages including Node.js/JavaScript. Travis-CI provides free service for public GitHub repositories, and you log in using your GitHub account, which is the first step to setting up a watch for the repository that needs continuous integration. Once logged in, you can choose which repos for Travis-CI to watch. The next step is creating and validating a Travis-CI configuration file (.travis.yml) in the root directory of the repo to watch.

Travis-CI configuration for client-side code bases is simple. It can get more involved when you add servers and databases. You can validate your configuration with a linting tool available from them. Most of the work that actually takes place on the Travis-CI server is what was defined in your Grunt/Gulp file and package.json. The next step to initiate continuous integration and deployment for your project is to make a commit. You can

### 7.8 Travis-CI Configuration (.travis.yml) for Our Project

---

```
language: node_js
node_js:
  - "0.10"

before_script:
  - export DISPLAY=:99.0
  - sh -e /etc/init.d/xvfb start
  - npm install --quiet -g grunt-cli karma
  - npm install

script: grunt
```

---

Given the contents of our .travis.yml file above we can start to piece together the CI workflow steps.

1. Make a code commit
2. Travis-CI instantiates a transient VM the exists only for the length of the workflow
3. A Node.js version 0.10 runtime is instantiated as per the language requirement
4. Some pre-integration set up takes place
  - a fake display is added to the VM for message I/O (- export DISPLAY=:99.0; - sh -e /etc/init.d/xvfb start)
  - Grunt and Karma are installed globally (- npm install --quiet -g grunt-cli karma)
  - all development dependencies listed in package.json are installed via NPM (- npm install)
5. The default Grunt task from our Gruntfile.js runs (script: grunt)

We are only using two of the available break points in Travis-CI's lifecycle. The other break points where commands can be executed include: before\_install, install, before\_script, script, after\_script, after\_success, or after\_failure, after\_deploy.

If any Grunt tasks fail in this environment, the build will be marked as failed. If you have the Travis-CI badge in your Readme.md file, it will display a green, build-passing image or a red, build-failing image. Common causes of failures include unlisted dependencies in package.json and inability to find files or directories during tests or build due to relative path problems. Builds can take several minutes to complete depending on the current Travis-CI workload, how many dependencies need to download, and amount of file I/O.

Travis-CI and some of the other cloud based CI systems that integrate with GitHub, are great continuous integration solutions for developers who'd rather spend the bulk of their time on coding rather than development operations. On the other hand, if you are on a larger team with a dedicated devOps engineer and servers behind a firewall, then dedicated, configurable CI systems like Jenkins or TeamCity offer a lot more flexibility.



You might be asking where is the deployment or delivery? Travis-CI will optionally transfer the designated distribution files to various cloud hosts such as Heroku. For our immediate purposes we are only using Travis-CI for continuous integration or build verification so we know that nothing should be broken if others clone the main repository and run a build.

There are multiple ways our library could be deployed:

1. Via direct download of a zipfile from a server
2. Via public CDN (content delivery network)
3. As a check out or clone directly from GitHub
4. Via a package repository such as Bower

Given that we are offering both the complete library, and custom subsets of components we will need to use two of the methods above. For custom builds we would need to create a GUI front-end to the Grunt CLI command for custom builds and use a web hosting account that clones the main Git Repository behind the scenes. For complete versions of the library, we can take advantage of a package management repository of which Bower is the most appropriate.

## Bower Package Registry

<http://bower.io/><sup>38</sup>

Bower (from Twitter) is a central web registry for packages, libraries, and apps geared toward client-side code including primarily JavaScript, CSS, and HTML. It is similar to Node Package Manager (NPM) or Ruby Gems. If we register a version of our UI component library as a package with Bower, anyone who has Bower installed can download our package using into their development directory using:

```
~$ bower install angular-components
```

This provides an easy route for our customers to get a hold of our library quickly.

Registering our component library with Bower is almost as simple. The first step involves generating a bower.json file, which has a similar structure to our package.json file. You can start the process by running:

```
~$ bower init
```

in the project root directory. It will take you through a short series of questions and present default answers that you can override or not. It will output the bower.json file, which you will likely want to amend in your editor.

---

<sup>38</sup><http://bower.io/>

### 7.9 Bower Configuration (bower.json) for Our Project

---

```
{
  "name": "angular-components",
  "version": "0.0.1",
  "homepage": "https://github.com/dgs700/angularjs-web-component-development",
  "authors": {
    "name": "David Shapiro",
    "email": "dave@david-shapiro.net",
    "url": "https://github.com/dgs700"
  },
  "description": "AngularJS UI Components",
  "main": "dist/ui-components-0.0.1.min.js",
  "keywords": [
    "Angular",
    "Components"
  ],
  "license": "MIT",
  "ignore": [
    "/*",
    "img",
    "less",
    "src",
    "node_modules",
    "test"
  ],
  "dependencies": {
    "angular": "~1.2.16",
    "bootstrap": "~3.1.1",
    "ngSanitize": "~0.0.2",
    "jquery": "1.10.2"
  },
}
```

---

This is the file that gives Bower the necessary information about the version of our component library we want to register. You can get a better idea of what should be included in your bower.json file by looking at the same file for the thousands of other registered packages. What is most important is having a unique, descriptive name, making sure the correct dependencies (not development dependencies) are included, and a “version” conforming to SemVer (semantic versioning) is listed. We will discuss SemVer in a bit, but for now the repository we will register must be Git tagged with the version (v0.0.1 for ours). You’ll also likely want to make sure any files and directories that should not be downloaded are in the “ignore” list.

Another important configuration switch to consider is “private:[true|false]”. If you want friends or colleagues to be able to download your package, but do not want it searchable on Bower, then you can set “private” to “true”. The package still needs to map to a public GitHub repository. If the package has proprietary content and is located on a private Git server, perhaps behind a corporate firewall, its possible to create your own Bower repository and run it on a server behind the firewall. Although, I have not yet tried it myself, you should be able to clone or fork the Bower GitHub repository and build your own. Often in larger, enterprise organizations with arcane and obtuse IT departments, this is the only option.

Once we have the bower.json file committed and the repository tagged, we can run:

```
~$ bower register <name> <GitHub endpoint> ~$ bower register angular-components  
https://github.com/dgs700/angularjs-web-component-development.git
```

...to register the package which is now installable via the “install” command above.

There are a few more items of importance concerning Bower. The names used are on a first-come first-served basis, and there is no way to un-register a package from Bower other than creating a Git issue and hoping for a response. In addition, any downloads from Bower are meant to be used as is. You should never run any kind of build or even edit anything that Bower installs in the bower\_components/ directory. That’s what cloning or forking repositories is for. What this means when it comes to component libraries is that you may need to register several permutations of the library with Bower to make everything available to customers. For example, the AngularJS team maintains both an “angular-latest” and an “angular” package on Bower along with packages for several of the non-core components like ngRoute.

Also, just as with all the other revolving trends and fads in the world front-end development, Bower is popular today, but may become obsolete tomorrow. My personal prediction is that at some point NPM will gradually replace Bower as the registry of choice as the distinction between client-side and server-side JavaScript continues to blur. The real knowledge takeaway here is that a package repository, whether Bower or something else, is an excellent way to provide continuous delivery to your customers.

## Versioning

Our discussion of building, automating, integrating, and delivering would not be complete until we talk about how versioning fits into all of this. Component libraries have some “special needs” when it comes to versioning. A proper system of versioning is essential for dependency management, both the dependencies your library depends on, as well as who depends on your library. However, it does not end at the library level. What about the components themselves, given that they are really miniature applications?

Suppose that over the course of a few development months we add small features and fix bugs for a few components while the rest are largely untouched. How do we manage version tracking between components when some components in the library depend on other components? Do we bump up the version of the entire library, or just certain components?

As far as I can tell, there is no definitive answer or established best practice, and there probably needs to be. As of today, most JavaScript component libraries maintain a single version number for the entire library as a whole using a specification known as “Semantic Versioning” or “SemVer” for short. For a “semantic” overview on SemVer see:

<http://semver.org/><sup>39</sup>

In a nutshell, semantic versioning uses three dot-separated numbers that can be represented by X.Y.Z. Certain types of metadata can be appended such as X.Y.Z-rc.1. X (major release) is incremented when API changes are not fully compatible with previous versions. Y (minor release) is incremented upon feature additions that are backwards compatible, and Z (patch) is incremented upon backwards-compatible bug fixes. New packages whose APIs are expected to be in a high state of flux during initial development should remain at 0.Y.Z until the API stabilizes. Upon version 1.0.0, the public API is considered set, and any changes require incrementing to 2.0.0.

Let’s suppose we start the version at 0.1.0 for the library and for each component. Logically then, if we add a backwards compatible feature to our Dropdown component, then we should increment the versioning for the component and the library to 0.2.0. There is no reason why individual versioning for the other components should not remain at 0.1.0. By definition, the components are all still compatible with each and can have dependency on each other. The same logic can be applied to patch versions as well.

When it comes to non-backwards compatible changes for any of the components, then the major version number should increment along with the major number for the components as well. This is expected since an incompatible API change for one component necessitates updating any other dependent components to be compatible once again. Major releases, therefore, require a lot of planning and coordination beforehand.

While there is no known best practice for versioning individual components in relation to the versioning for the library as a whole (the conclusion drawn after running every Google search I could think of), there are some situations to consider when determining the best strategy. The first is whether or not there are components that will often be used without the rest of the library. A grid component is a common example of this. The second is the nature of the development team for the library. If certain developers work on some components but not others, yet a significant number of components have dependencies on other components, individual tracking may make sense. If individual versioning is employed, the library and all the components should be versioned at the same major number and the minor and patch numbers for any component should never exceed those of the library. This applies to each development branch such as a “stable” 2.Y.Z branch and the latest, 3.Y.Z branch.

---

<sup>39</sup><http://semver.org/>

## Salvaging Legacy Applications with AngularJS UI Components

We'll finish this chapter by taking a bit of a detour from the topic of building and delivering our UI component library, and discuss a unique and very useful thing we can do with the UI components that we create.

If you are reading this book, it is very likely that you are one of two types of engineer. The first type are those who are fortunate to have the luxury of being able to build new applications using the latest front-end technologies. The second type are those who have the burden of maintaining legacy front-end applications that suffer from multiple revolving developers doing their own thing, bloated and repeated functionality everywhere, and performance so bad that loading and user responsiveness can be measured in seconds rather than milliseconds. This section is for the second type.

In the first few chapters, we extolled the benefits of AngularJS compared to older frameworks, especially in regards to drastically reducing binding boilerplate and other repeated code. Bloated applications full of jQuery spaghetti are the direct result of poor planning, short term focus, and the dysfunctional layers of management found in most enterprise organization of significant size and age. Marketing driven organizations often suffer from the misguided belief that you can produce a "Cloud" product faster by throwing scores of offshore contractors at the project. A year later, the web based product might be functional, but it far from maintainable given all the corners that were cut to push the product out the door including testing, documentation, and other best practices. Within two years, development grinds to a halt due to bloat and the inability to fix a bug without causing three more in the process.

In certain cases, it may be possible to salvage the application from the inside out by systematically replacing repeated functionality and code with common, reusable UI components built with AngularJS. The assumption is that the page is under the control of a legacy framework, but if the application meets certain conditions it is salvageable including:

- The app allows the developer control over lifecycle events resulting in DOM redraws
- DOM redraws are template based, not JavaScript based
- AngularJS core and UI component modules can be loaded with some control over order
- Any jQuery in the page is a relatively recent version
- The root element(s) that would contain the components are accessible

Existing frameworks that are highly configuration based (i.e. Extjs) are likely beyond repair because most of the DOM rendering is scattered deep in layers of its core JavaScript which is not accessible without modifying the framework. Frameworks that are jQuery based (imperative), on the other hand, make excellent salvage candidates including Backbone.js.

Our example will include code snippets that illustrate how simple it is to embed an AngularJS UI component in a Backbone.js / Require.js application. The first task is to make sure that AngularJS

core is loaded and angular is available as a global. If loading AngularJS core will be done via AMD with Require.js, it should be done early in the file used by Require.js to bootstrap all of the modules. If jQuery is loaded here, AngularJS must be after jQuery if it is a dependency. Likewise, the code containing the AngularJS modules must load after AngularJS core. Loading these files asynchronously without a dependency loader that guarantees order will not work.

### 7.10 Embedding an AngularJS UI Component in Require/Backbone

---

```
// main.js
require([
    'underscore',
    'jquery',
    'backbone',
    'angular',
    'angularComponents',
    '...'
], function( mods ){
    ...
});

<!-- template_for_view_below.hbs.html -->
...
<div id="attach_angular_here">
    <uic-dropdown ...></uic-dropdown>
</div>
...

// view.js
define([
    'deps'
], function (deps) {
    return View.extend({
        className: 'name',
        initialize: function () {
            ...
        },
        render: function () {
            // give the parent view time to attach $el to the DOM
            _.defer(function () {
                try{
                    // this replaces <div ngApp=" uicDropdown "
                    angular.bootstrap('#attach_angular_here', ["uicDropdown"]);
                }catch(e){
            
```

```
        console.error('there is a problem');
    }
    });
});
}
```

---

The pseudo code above demonstrates how and where to place the AngularJS code within the lifecycle of the Backbone.js application. The important pieces include making sure the dependencies get loaded in the correct order and that the DOM element that contains the custom element and where we will bootstrap AngularJS is appended to the DOM *before* `angular.bootstrap()` is executed. Using `lodash defer` or `setTimeout` places the contained code at the end of the JavaScript execution stack. For robustness, the `try` block is included so any problems can fail gracefully without halting the UI.

AngularJS apps can be bootstrapped imperatively on multiple DOM elements assuming they are not part of the same hierarchy. For best performance, however, the number of AngularJS app instantiations should be kept to a minimum by attaching to elements higher in the DOM hierarchy if possible. The pattern above can be used to replace any of possibly several different dropdowns implementations throughout the application. In the process, lines of source code will be reduced and performance will increase.

This is one scenario of injecting AngularJS into an existing application. In practice, it takes some creativity and a few attempts to find a system that works for your particular application. However, the time spent can be well worth it given that the alternative would be to start from scratch and develop an entirely new application while having to maintain the old one until it is complete.

## Summary

This was the chapter where we bridged the gap between creating source code that is “functionally” complete and placing the “product” in the hands of our customers. In the real world there is quite a bit of “bridging”. Unfortunately, most of the how-to-build-a-webapp books written these days skip over these vital steps.

We covered the *concepts* of source code maintenance, modern repository tools, automating code quality checks, automating local and central builds, and methods of automatic deployment or delivery. The concepts are what we want to stress, rather than the actual toolsets used. In the fickle world of web UI development, the hot tool this year becomes “sooo last year” so quickly that attempting to publish books about the tools themselves is futile. The steps, patterns, and concepts are what tend to be more static and last the length of a full “technology iteration” which in software and web can be from 7-12 years.

Finally, we touched on the concept of utilizing reusable UI components to replace chunks of legacy applications from the inside out, with the goal of avoiding complete application rewrites from scratch.

In the final third of this book we will talk about the near future of web development. Web components are a set of planned and proposed additions to the DOM and JavaScript that will greatly shift how we develop for the web. AngularJS 2.0 plans to utilize some of these technologies, but we can explore these new technologies now with polyfills including Polymer and X-tags.



# 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, techniques 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](http://www.w3.org/standards/techs/components#w3c_all)<sup>40</sup>

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.

---

<sup>40</sup>[http://www.w3.org/standards/techs/components#w3c\\_all](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><sup>41</sup>

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 wish to use Canary for some of the new JavaScript ES6/7 features.

---

<sup>41</sup><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/><sup>42</sup>

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.

---

<sup>42</sup><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 extension 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>

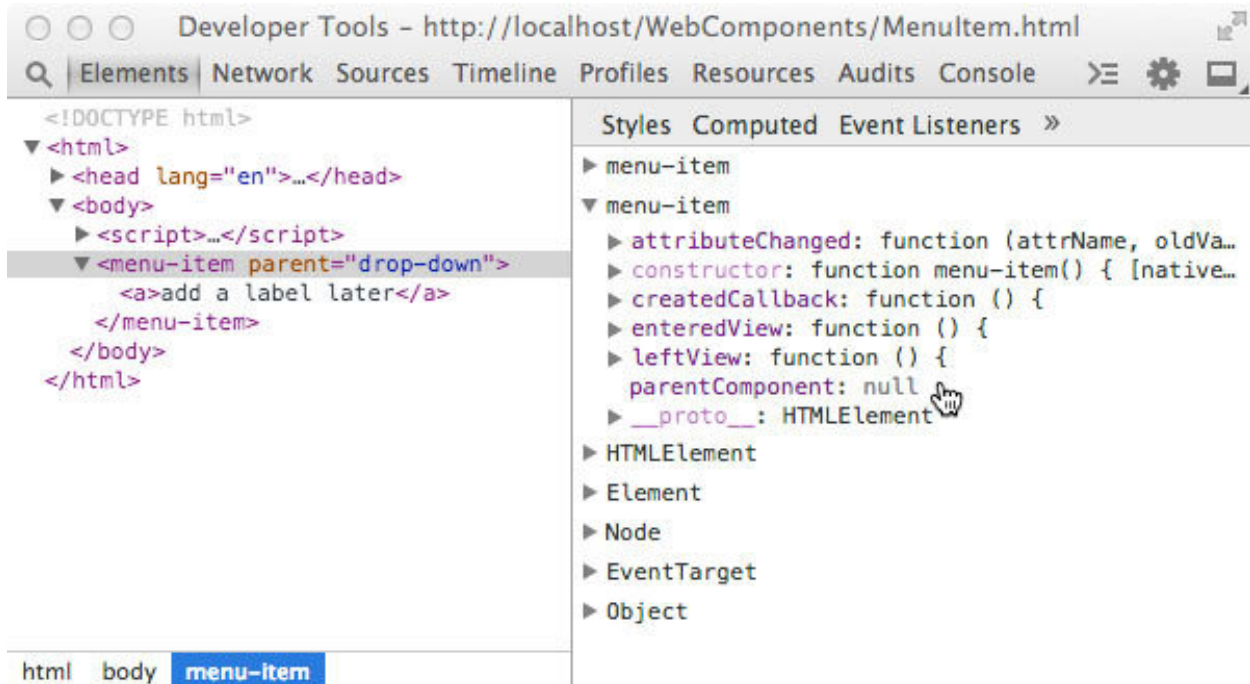
<menu-item parent="drop-down"></menu-item>

```

---

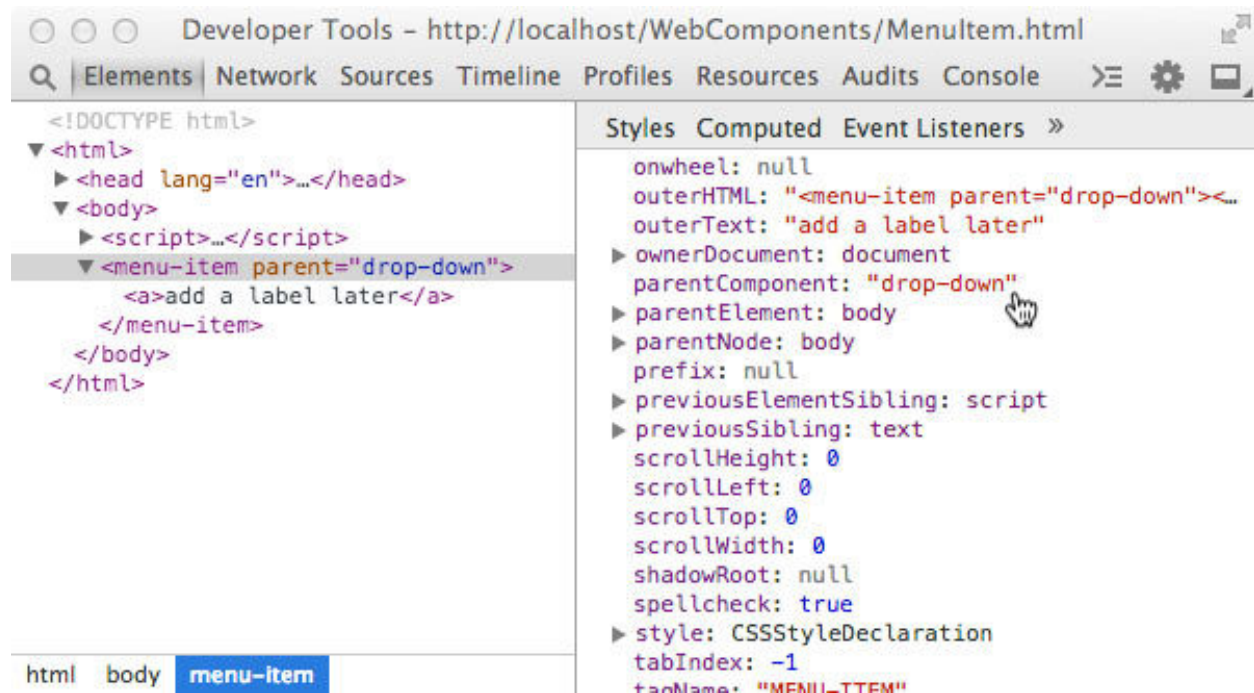
The proposed custom element specifications include a set of lifecycle callback method names. The actual names are in a state of flux. The ones in bold in the example are current as of mid 2014. Whatever the final names are, there will most likely be a callback upon element registration where

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 prototype 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/><sup>43</sup>

<http://w3c.github.io/webcomponents/spec/shadow><sup>44</sup>

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 separate 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,

---

<sup>43</sup><http://www.w3.org/TR/shadow-dom/>

<sup>44</sup><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 corresponding 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 referred 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 referred to as **distributed nodes**.

Not only may a **shadow host** element have multiple **content insertion points**, it may also have multiple **shadow insertion points**. Multiple **shadow insertion points** 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 points** 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 refers 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><sup>45</sup>

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 development 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.

---

<sup>45</sup><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>

```

---

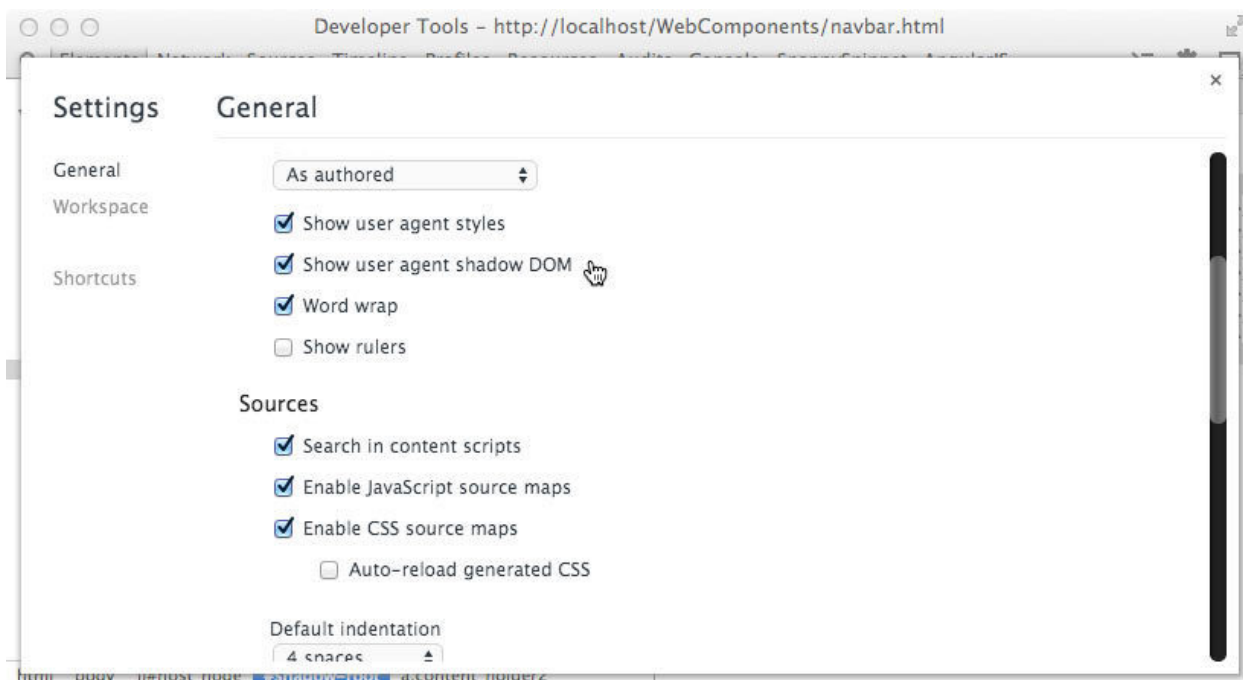
This is a minimalist example of creating, populating, and attaching shadow DOM to a custom element. The created or createdCallback (depending on final name) function is called when the instantiation process of a custom element is complete which makes this moment in the lifecycle ideal for instantiating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

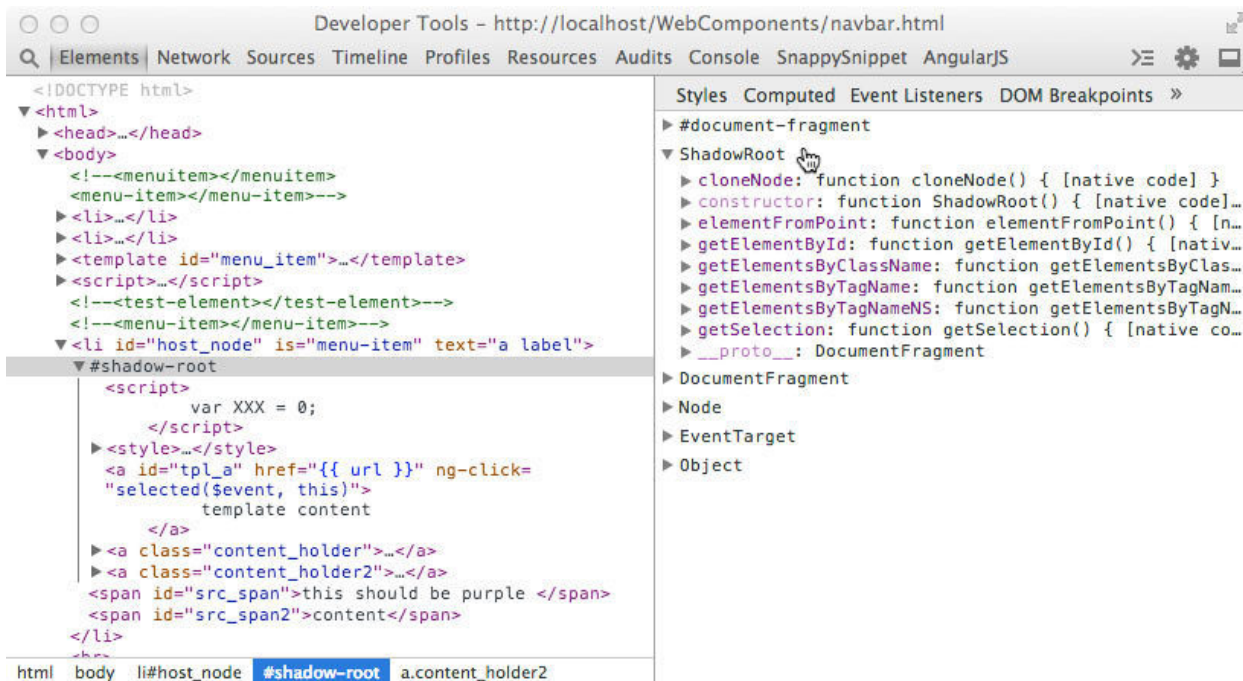
## Shadow DOM Inspection and Debugging

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><sup>46</sup>

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><sup>47</sup>

## 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.

---

<sup>46</sup><http://html5-demos.appspot.com/static/shadowdom-visualizer/index.html>

<sup>47</sup><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 -->
  
  <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>

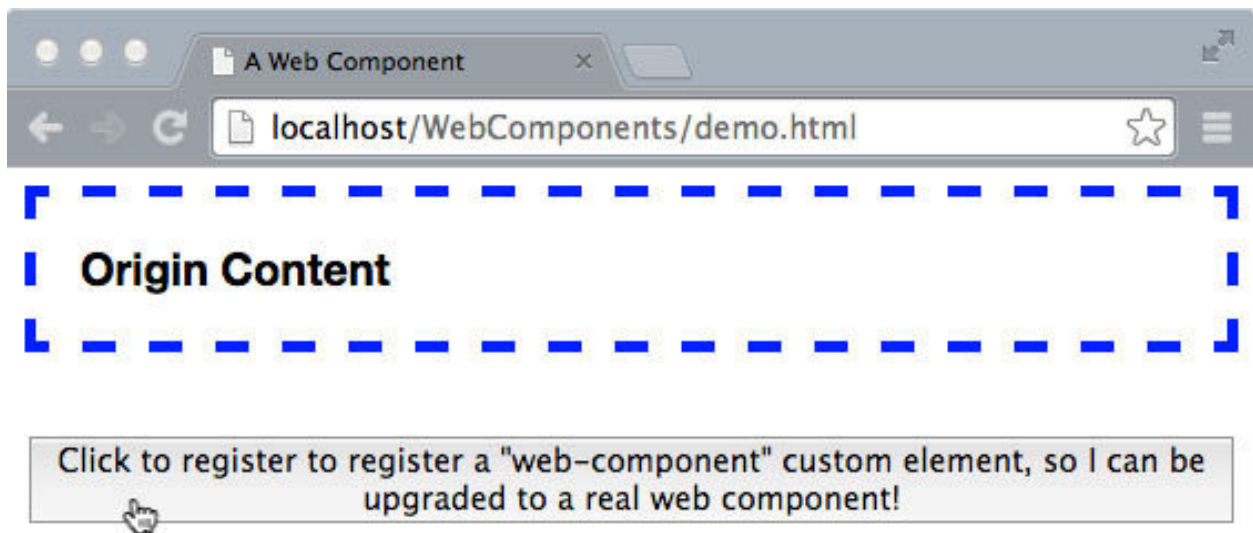
<!-- concepts covered: basic custom element registration, <template>,
<content> tags, single shadow root/host/DOM, :unresolved,
:host pseudo classes, ::content pseudo element -->

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

```

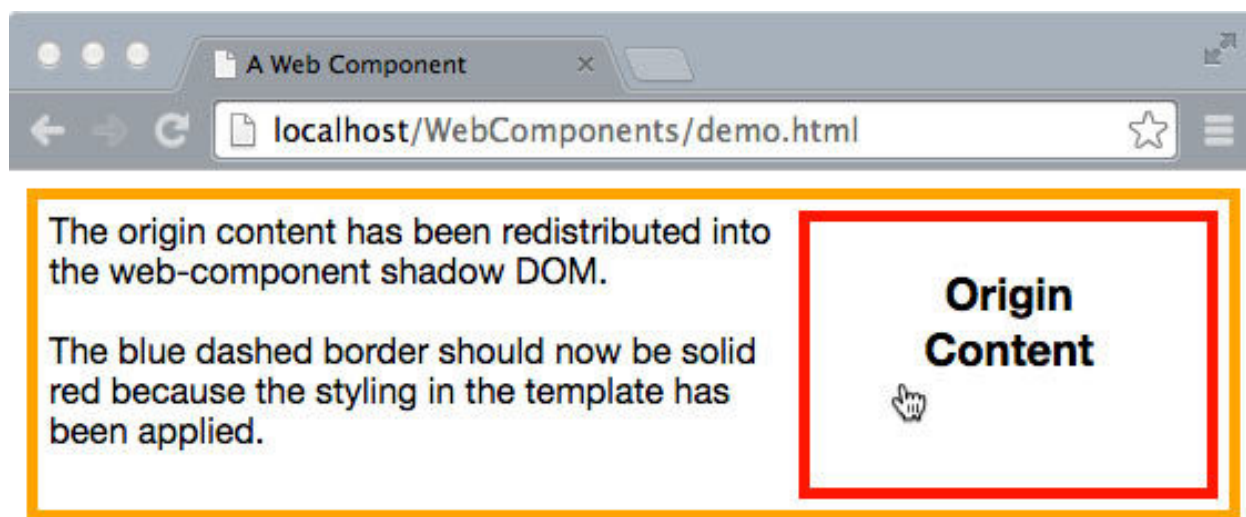
---

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.observe()` 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



server-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 vehicle 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/><sup>48</sup>

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.

---

<sup>48</sup><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-tp1');
```

---

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><sup>49</sup>

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:

---

<sup>49</sup><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 containing 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>

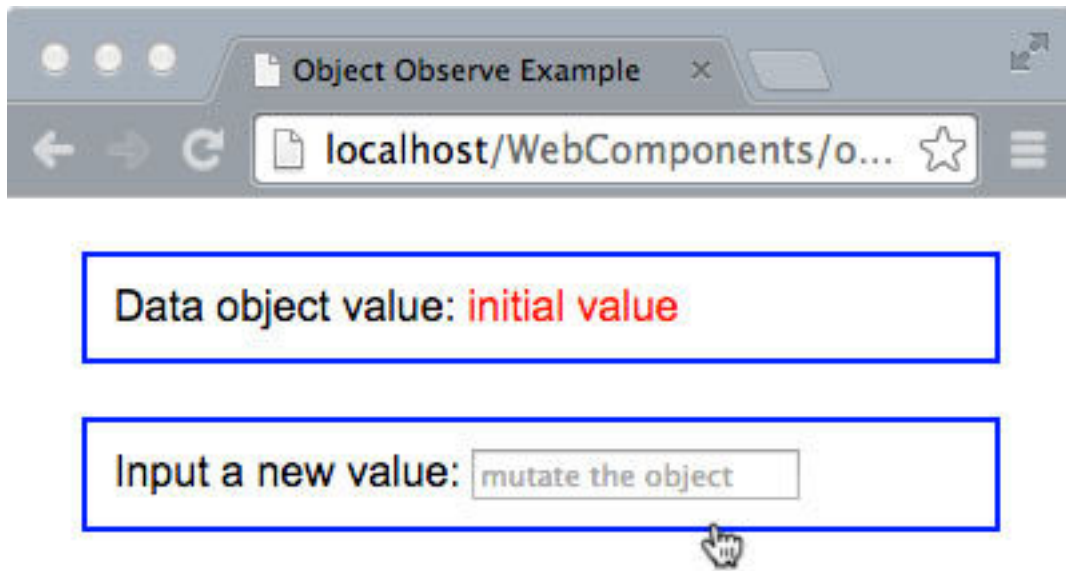
<script>
// the data object to be "observed"
var dataObject = {
    textValue: 'initial value'
};
// user input
var inputElem = document.querySelector('#input');
// this element's textValue node will be "data bound"
// to the dataObject.textValue
var outputElem = document.querySelector('#output');
// 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]);
  });
}

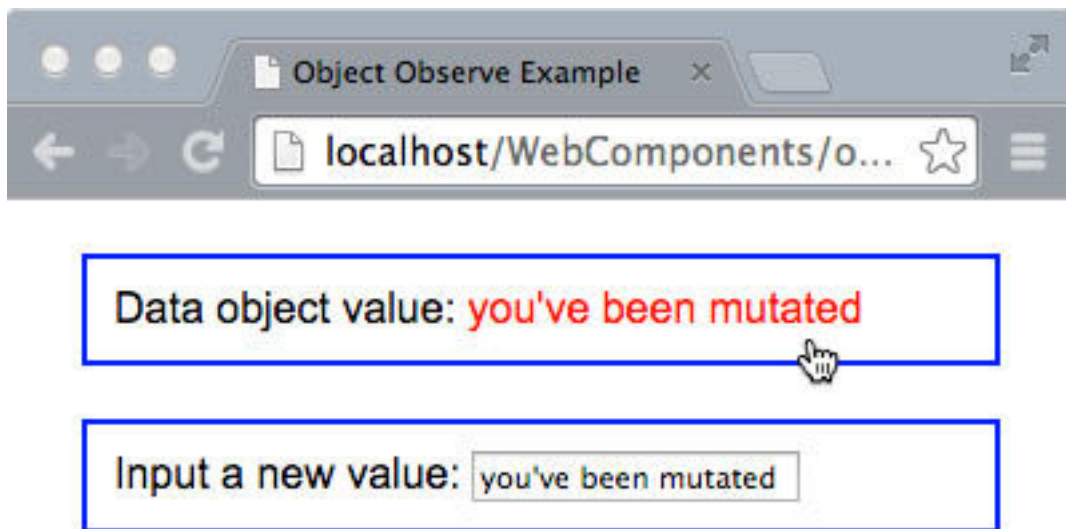
// set a keyup listener for user input
// this is similar to AngularJS input bindings
inputElem.addEventListener('keyup', mutator, false);
// set a mutation listener for the data object
// notice how similar this is to setting event listeners
Object.observe(dataObject, watcher);
</script>

</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 by 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 browsers can be found at:

<http://kangax.github.io/compat-table/es6/><sup>50</sup>

Our coverage is merely a quick summary 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:

---

<sup>50</sup><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 able 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/><sup>51</sup>

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](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy)<sup>52</sup>

---

<sup>51</sup><http://tc39wiki.calculist.org/es6/template-strings/>

<sup>52</sup>[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Proxy](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 browsers understand.

<https://github.com/google/traceur-compiler/wiki/GettingStarted><sup>53</sup>

<http://google.github.io/traceur-compiler/demo/repl.html><sup>54</sup>

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:

---

<sup>53</sup><https://github.com/google/traceur-compiler/wiki/GettingStarted>

<sup>54</sup><http://google.github.io/traceur-compiler/demo/repl.html>

<https://drive.google.com/?pli=1#folders/0B7Ovm8bUYiUDR29iSkEyMk5pVUk><sup>55</sup>

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><sup>56</sup>

## 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.

---

<sup>55</sup><https://drive.google.com/?pli=1#folders/0B7Ovm8bUYiUDR29iSkEyMk5pVUk>

<sup>56</sup><http://blog.angularjs.org/2014/03/angular-20.html>

# Chapter 9 - Building with Web Components Today

In chapter 8 we took a look at the W3C standards proposals that comprise “web components”. We also provided example code snippets that can be executed in Chrome Canary, the beta version of Chrome that has many experimental features not yet in production browsers.

While Chrome Canary can be fun for experimenting with new web technology, due to security and stability issues, that’s all it can be used for. For those interested in moving beyond Canary experimentation, Google’s Polymer team has created a number of polyfills (platform.js) that can simulate “most” of the web components standards in today’s production browsers. In addition to providing polyfills for `<template>`, custom elements, shadow dom, and object observables, the Polymer team has created a custom element based framework (polymer.js) that ties together these technologies with some syntactic sugar. The Polymer team has also created a set of “core elements” and UI elements (now called paper elements) that provide several SPA application features such as AJAX and local storage wrappers plus much of the UI page structure that is found in most web apps such as menu, dropdown, and header elements (sound familiar?).

<http://www.polymer-project.org/><sup>57</sup>



Update 10/2014: Web Component APIs are now fully integrated into production Chrome

The Polymer team has even created a “designer” web app that can be used by web designers to construct web applications out of Polymer elements with styling that conforms to Google’s new generation user experience guidelines called Material Design. Web developers should probably start considering other professions because our skills may soon become obsolete ;-)

The goal of this chapter is to take a look at the polyfills, framework, and element libraries from Polymer from several angles, high level, detailed, critical, and pragmatic. The pragmatism is especially important. There is universal consensus among web developers that “web components” are the future of web development and should be highly promoted via projects like Polymer (and X-Tag from Mozilla) to help force widespread browser implementation. But how do we balance that with the reality that today the majority of enterprise organizations use Internet Explorer as their standard browser, and IE regards web component technologies as merely “under consideration” on its development roadmap?

---

<sup>57</sup><http://www.polymer-project.org/>

The Polymer project, itself is still in a high state of flux with a consistent stream of breaking changes. For these reason we won't dive too deeply into detailed examples of building a web component library or web application using Polymer as it is in "alpha" state, and will be for quite some time.

## Platform.js (Webcomponents.js) - Web Components Polyfills



Update 10/2014: platform.js is being renamed to **webcomponents.js**. There should be no changes in functionality.

Platform.js, from the Polymer team, is the foundation for the Polymer framework. It provides cross-browser polyfills for all of the web component standards proposals we covered in the previous chapter including `<template>` tags, HTML imports, and the custom element and shadow DOM APIs. It also provides polyfills for ES6 WeakMap, DOM mutation observers, and web animations.

<http://www.polymer-project.org/docs/start/platform.html><sup>58</sup>

Web Animations are a separate W3C standard proposal that is meant to address deficiencies in the current web animation standards including CSS transitions, CSS animations, and SVG animations. More information on web animations can be found at:

<http://dev.w3.org/fxtf/web-animations/><sup>59</sup>

<http://www.polymer-project.org/platform/web-animations.html><sup>60</sup>

ES6 WeakMap will be a welcome addition to JavaScript. The dreaded *memory leak* in web applications occurs because unused objects cannot be garbage collected until all reference keys to the object are destroyed. The problem is that hard references to an object can easily end up in places other than the primary reference, often as a side effect of using a UI framework such as jQuery to wrap a DOM object. WeakMap keys, on the other hand, do not prevent object garbage collection when the primary reference is destroyed. As newer versions of popular frameworks start using WeakMap for tracking objects, web developers will not need to be as meticulous in performing all the "clean up" steps when a chunk of the page is replaced during the normal lifecycle of an application.

Platform.js also provided support for "pointer events" which smooth out the differences between desktop hardware that uses mouse events and mobile hardware that uses touch events. However, that support quietly disappeared right around the time support for Web Animations was added. Now we should point out that Platform.js is a bundle of individual polyfills. You can still visit the GitHub repos for the Polymer project, grab any of the individual polyfills, and roll your own custom Platform.js. The advantage with this approach is you can create a much smaller file with just the polyfills that work correctly across browsers, and in most cases, leave out Web Animations.

---

<sup>58</sup><http://www.polymer-project.org/docs/start/platform.html>

<sup>59</sup><http://dev.w3.org/fxtf/web-animations/>

<sup>60</sup><http://www.polymer-project.org/platform/web-animations.html>

<https://github.com/Polymer><sup>61</sup>

<http://www.polymer-project.org/resources/tooling-strategy.html#git><sup>62</sup>

## Browser Compatibility and Limitations

Recall in the previous chapter in code example 8.5 where we created an HTML document that demonstrated defining a custom element, and then creating a shadow DOM root and inserting `<template>` tag content into the shadow root as part of the element lifecycle. We did this using Chrome Canary with before and after screen shots that demonstrated the redistribution of origin DOM nodes in the shadow tree.

To test how the Platform.js polyfills would stack up across current production browsers, we added:

```
<script src="bower_components/platform/platform.js"></script>
```

to the head of the same document and loaded it in Chrome 35, Firefox 30, and Safari 6.1.3 (desktop versions). We also tested the HTML Imports polyfill with another document that imported this document.

In Chrome 35 everything worked just as it did in Chrome Canary, and loading Platform.js was actually not necessary unless using HTML Imports. The custom element and shadow DOM APIs, plus `<template>` tag, are already implemented natively.

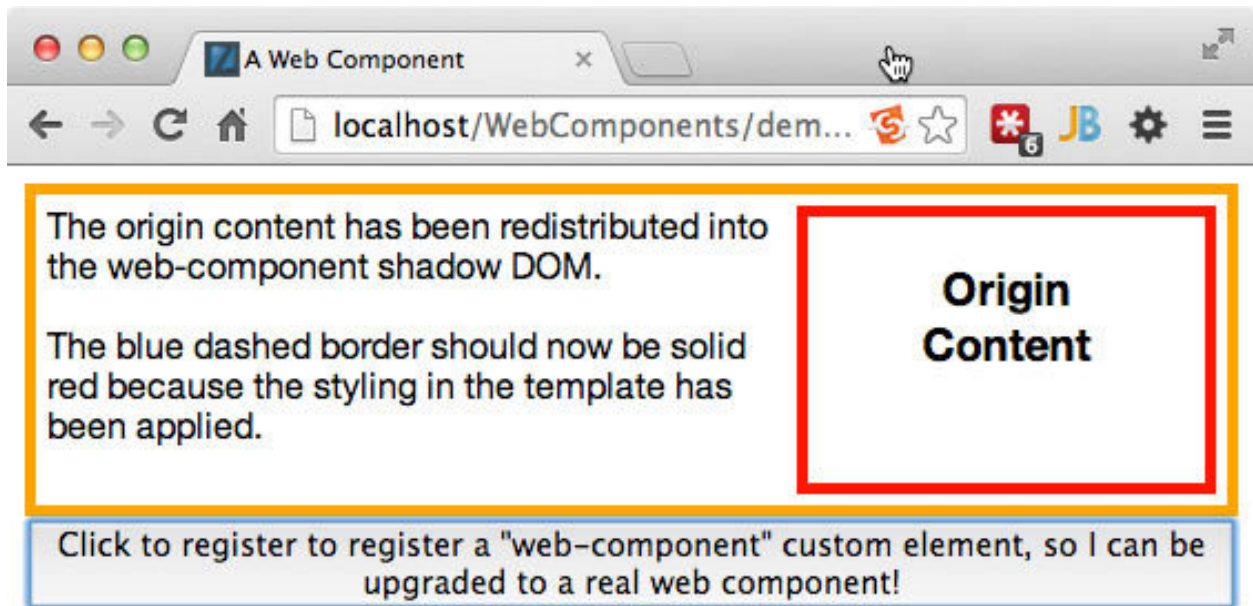
Unfortunately Firefox and Safari didn't fare quite so well. The shim for HTML Imports worked (via AJAX). `<template>` tags are already first class elements as part of HTML5. The custom element polyfill appears to support the API. Inspection of the custom element reveals that it is indeed inheriting from `HTMLElement` which is what we specified. We did not test extension of existing elements. The shadow DOM basically polyfill failed, and predictably so.

---

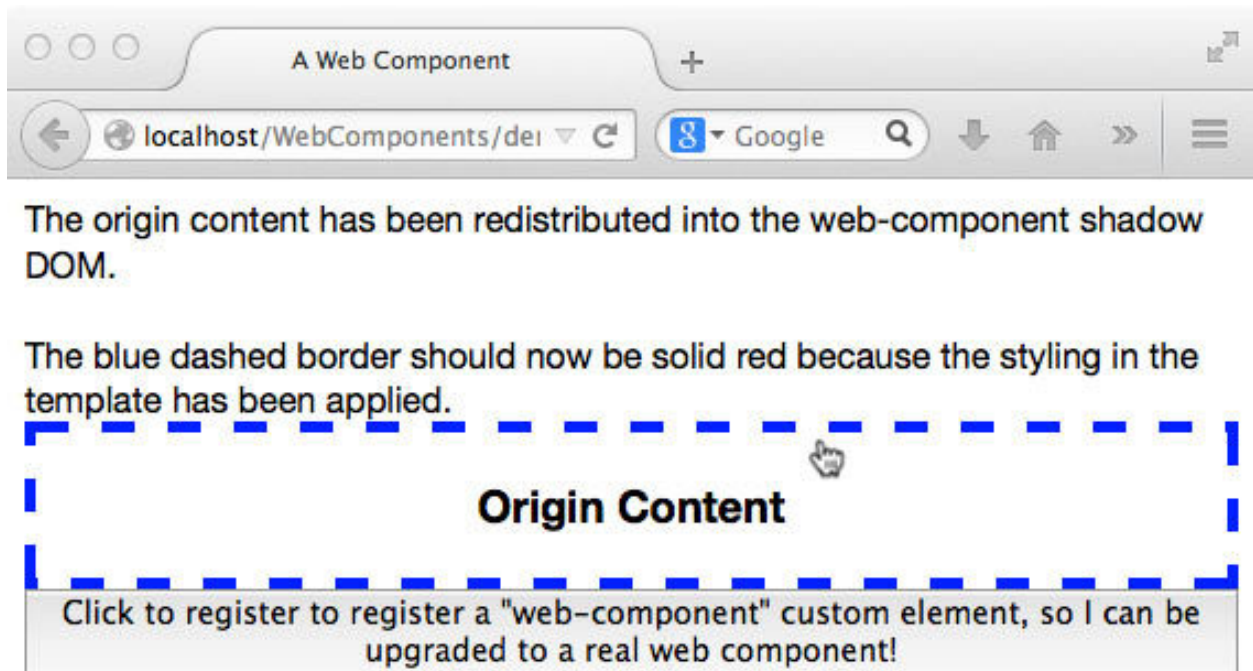
<sup>61</sup><https://github.com/Polymer>

<sup>62</sup><http://www.polymer-project.org/resources/tooling-strategy.html#git>

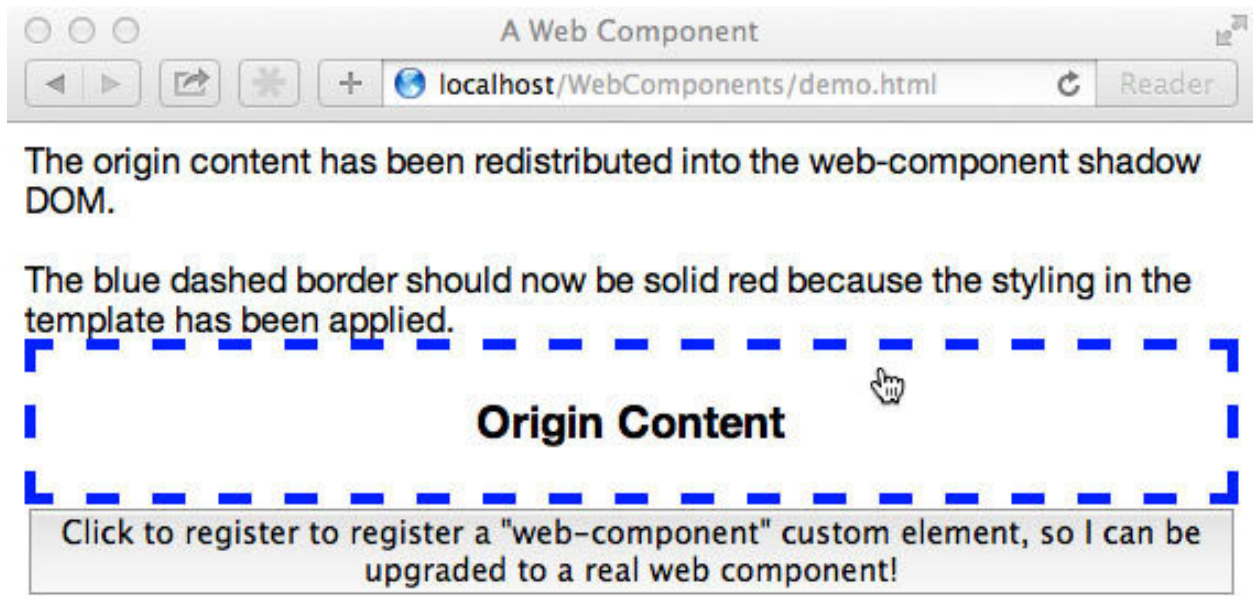




Platform.js test in Chrome 35, shadow CSS is applied



Platform.js test in Firefox 30, shadow CSS is not applied



Platform.js test in Safari 6.1.3, shadow CSS is not applied

#### 9.0 Shadow DOM CSS that cannot be polyfilled

```
<template id="web-component-tp1">
  <style>
    /* example of ::content pseudo element */
    ::content div {
      /* this overrides the border property in page CSS */
      border: 5px red solid;
      width: 300px;
    }
    /* this pseudo class styles the web-component element */
    :host {
      display: flex;
      height: 130px;
      padding: 5px;
      border: 5px orange solid;
      font-size: 16px;
    }
  </style>
```

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

```
<br><br>
```

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>
```

---

As you can see from the Firefox and Safari screen grabs above, the CSS rules in the template content that is attached to the shadow root fail to be applied. The parent document CSS including the dashed blue border still applies. The pseudo class, :host, and the pseudo element, ::content, which should style the shadow host node and the content distribution node respectively, have no effect.

Keep in mind that this was

While this is unfortunate, it does make sense. JavaScript polyfills can only affect what they have access to. This includes all native JavaScript and DOM objects. JavaScript can affect the style properties of any individual node, but it can't exert control over the recognition, influence or priority of CSS selectors. Only the native browser code, typically C++, has such control because CSS style application needs to happen at page rendering speed.

If we step back a bit and revisit the issues surrounding UI component encapsulation, you may recall that encapsulating CSS is the Achilles heel that prevents the construction of *fully* self-contained UI components. Even with the shadow DOM polyfill in Platform.js we still have the problem of CSS bleed across component boundaries whether web component or AngularJS UI component. This is the major reason for developer access to shadow DOM to begin with.

So here is what we can do usefully with Platform.js across browsers today (mid 2014). We can use <template> tags in Internet Explorer and we can use the custom element API to declare new elements with special purposes or extend existing elements with new functionality. HTML Imports are useful for the development of web components, but for a production application they need to be flattened for performance reasons.

Therefore, it is not surprising that the Polymer framework and the web component libraries (Polymer Core, Paper elements, X-Tag/Brick) are primarily built upon an architectural foundation of custom elements. The Polymer website maintains a browser compatibility matrix at:

<http://www.polymer-project.org/resources/compatibility.html><sup>63</sup>

Just keep in mind that a green rectangle could be interpreted as the Polymer team's definition of support especially in regard to shadow DOM which is clearly limited to the DOM/JavaScript API methods. Also, keep in mind that until custom elements, shadow DOM, HTML imports, and Object.observe become web standards, they and Platform.js are subject to breaking changes.

---

<sup>63</sup><http://www.polymer-project.org/resources/compatibility.html>

From the point of view of a UI architect of enterprise grade, production web applications it would clearly not make sense to build a major new web application on a foundation like Platform.js which is in the state of “developer preview” according to the Polymer team. The purpose of highlighting this point is due to recent statements and hype from Google and the Polymer team that has come close to misrepresentation.

## The Upside of Platform.js - Future Proofing Today's Application

First, it is a bummer that the previous sub-section had to be included. The reason for it is essentially due to professional responsibility (CYA) in evaluating new technology as a UI architect.

While the hype can be misleading to some less experienced web developers, it is also necessary in order to move web standards technology forward. In this regard, the Polymer team should get a standing ovation for their efforts. The educational and motivational value of the Platform.js polyfills, Chrome Canary, and Firefox Nightlies in turning on the web development community to these new standards is immense.

Every web developer who plans to be in this business three years from now should already be getting familiar with custom elements and shadow DOM APIs. It has practical value today in terms of structuring the code for current forms of UI components in such a way that it can be easily converted to take advantage of the web components APIs when ready. This includes managing the source code representations of behavior (JavaScript), structure (HTML templates), and presentation (LESS, SASS) together as a component, but separately within the component as a mini MVC application.

The advertising and hype about future technology is what counters the dead weight caused by archaic enterprise applications from companies like Oracle, SAP, and HP that keep legacy IE 6/7 from just going away. Even if Microsoft drags their feet on web components support they are now forced to acknowledge their lack of support:

<http://status.modern.ie/><sup>64</sup>

The ability to create applications based on web components rather than just reading the W3C drafts, even if they only work fully on Chrome, can also be quite valuable in choosing JavaScript frameworks today. For example, consider frameworks that have already announced upgrade paths towards web component technologies in their roadmaps such as AngularJS or Ember. These frameworks let you style your source code in a web component friendly manner. In fact, this is the reason the first two-thirds of this book focus on building UI component libraries with AngularJS.

Now consider other frameworks such as ExtJS and the shiny new one from Facebook called React.js. The source code you would create with these frameworks will be anything but web component friendly. They tightly intermix and embed HTML generation deep within the core JavaScript, and they do not support two-way data binding. This tight coupling is well proven to hamper source code maintenance. React.js was developed by Facebook not so much to allow *reactive* programming style, but as a *reaction* to the performance problems caused by packing too much functionality into every

---

<sup>64</sup><http://status.modern.ie/>

page view of the Facebook application. Facebook's use-case is far from the typical web application use-case.

As a UI architect, if I were tasked with designing an application architecture for something as monstrous as Facebook, then obviously AngularJS' dirty-checking would not work, and I would consider React.js' DOM "diffing" in order to get adequate performance out of the average browser used around the world. But for the average application my "go to" frameworks will those that support future technologies and best-practices.

## Polymer.js

The Polymer team has named their framework after large molecules composed of many repeated monomers. The basic philosophy of Polymer is that *everything is an element*.

I suppose this makes sense if you are a chemical engineer. From the perspective of a web engineer this, technically, is a statement of fact given that all of the framework and application logic ends up in a `<script>` element eventually. The same goes for all of the CSS rules loaded by `<link>` or inhabiting `<style>` elements.

The only problem with facts is that they are not debatable, which is why we all the fun comes from having opinions. That's what we are here to talk about. But first we need to understand what exactly comprises Polymer.js.

Polymer.js is the Google Polymer team's web application framework that sits directly on top of platform.js. The purpose is to add a layer of convention and tools upon the web components proposals that makes them easier to use in composing a web application. The author's opinion is that it is vital to have a solid understanding of the content and state of each of the underlying W3C standards drafts before spending time experimenting with Polymer.js. It's tempting to skip this learning and go straight to the tutorial project on the Polymer website. The mistake in doing this is that your time spent will be *uninformed*. To put it another way, most web developers would probably prefer to be aware that there is a good chance that what is in the shadow DOM CSS proposals as of mid-2014 will not be the version that is actually accepted as a cross-browser standard. Having this sort of knowledge upfront can allow you to better divide your time between *experimenting* with the coming standards and *producing* with the current standards.

### Know Where the Facts End and Opinions Begin

Another reason to gain a solid understanding of the W3C Web Components standard proposals prior to working with Polymer.js (or any other Web Component based framework) is to know where the browser standards end and the framework opinions begin. Having this understanding will allow you to make a more informed assessment as to the value the framework brings in fulfilling the business use case of your application.

Conceptually, Polymer.js provides three categories of tools and features. The first category are the usual set of web application tools provided by most modern JavaScript frameworks such as data-binding, custom events, and a version of OO style component extension. The second category are

tools specific to building custom elements (components) with encapsulated logic, style, and content. The third category might be considered advertisements for other web standards proposals from Google, currently “web animations” and a one-size fits all event wrappers for normalizing mouse and touch events. The third category is not in the scope of this book.

## The Usual Web Application Tools

Polymer.js provides web app construction tools very similar to AngularJS and other frameworks. Here’s a bullet list:

- Two-way data binding for “published properties” via `{{ }}` (mustache) syntax
- Declarative event mapping via *on-event* attributes to component functions
- Attribute or property value watchers
- Attribute or property group watchers
- Custom event bindings and triggers (publish and subscribe)
- Services wrapped as non-UI elements for commonalities like xhr and local storage

AngularJS uses an identical data binding syntax. Event bindings, individual and group watches, events triggers are all directly equivalent to AngularJS event directives like `ngClick`, `$scope.$watch('value to watch', callback)`, `$scope.$watchGroup()`, and `$scope.$emit()` and `$scope.$broadcast()` respectively.

A framework commonality that Polymer.js handles differently are application level services such as AJAX and local storage. Application and global services are typically implemented via some sort of service singleton. An example would be AngularJS’ `$http` service. This is where Polymer’s “everything is an element” philosophy ventures deeply into opinion. The Polymer opinion is that in order to develop an application the Polymer way you would either access the built-in global services by including and configuring the attributes of one of their core elements. Currently there are only a few, but it would be expected that for others, you would either create a custom element wrapper, or find one from a public component registry.

## Everything is an Element

The author’s opinion of this approach is that it is misguided in the typical form of “this is the way the world should be” as opposed to the way things are for most web developers. To put it more concretely, the *everything is an element* notion would lead to a code maintenance nightmare. Consider again the approach taken by ExtJS where almost all of the rendered DOM is spit out from deep within their “configured” JavaScript objects- objects that extend other objects six or seven levels deep with a dozen or so lifecycle callbacks. When it comes time for a corporate re-branding, all of the JavaScript including the core library must be ripped apart to figure out where a `<div>` begins life or where a CSS classname is applied. It’s quite common for enterprise organizations in bad marriages with ExtJS to dedicate entire development teams to the task of restyling a couple pages. This all stems from the snake oil Sencha sells to non-technical executives that you can create web applications entirely via configuration, no front-end developers needed.

If everything is an element, then expect to start seeing HTML like the following:



```
<database-element initialize='{ "name": "font-roboto", "homepage": "https://github.c\
om/Polymer/font-roboto", "version": "0.3.3", "_release": "0.3.3", "_resolution": { "typ\
e": "version", "tag": "0.3.3", "commit": "3515cb3b0d19e5df52ae8c4ad9606ab9a67b8fa6"}, \
"_source": "git://github.com/Polymer/font-roboto.git", "_target": "~0.3.3", "_origin\
alSource": "Polymer/font-roboto", "_direct": true}'>
```

Imagine an entire page full of elements like this ...get the point?

HTML is a direct descendent of SGML which was developed specifically to provide a declarative syntax for defining the structure of a document, not the processing of a document. The whole point of declarative markup is to provide a document that is readable both by machines and people. The element above is neither declarative nor readable which makes it unmaintainable by default. This is why complex data attributes belong in a .json or .js file, not an .html file. By complex, we are referring to non-primitives which include objects, functions, and arrays. It's the author's opinion that primitive vs. complex attribute values should be solid dividing line that determine the appropriate location for maintaining data and logic. Along these lines, its not uncommon for AngularJS templates to cross the same line and become unreadable as an element declaration can span multiple lines when a half dozen directives are included.

Embedding application level logic within an element is no different than embedding element creation in JavaScript. While some standard elements like `<meta>`, `<style>`, `<script>` do not render to the display, they are generally very specific in terms of making a declaration about the document or loading a resource and the attribute APIs accept primitive types (boolean, string, and number).

The real point is not to bash on Polymer's approach. Their ultimate goal is to publicize and educate the web community about coming web components standards that will benefit everyone instead of offering a serious web framework that would actually compete with the likes of AngularJS. This sub-section is the unfortunate by-product of a web architect calling out some bad architecture. The next section discusses some great ideas produced by the Polymer team for adding an abstraction layer to web components that make working with them more convenient and accessible for the average web developer.

## <polymer-element> Declarative Custom Element Definitions

Recall in the previous chapter that according to the current state of W3C web components proposals you create a custom element by extending an existing `HTMLElement` prototype, adding lifecycle callback methods, and then calling `document.registerElement(...)` to finalize the element type for instantiation. In addition to this *imperative* method, the proposal also included a *declarative* method via the new `<element>` tag which proved not to be feasible at the browser native code level.

## The Core

The Polymer framework still offers the declarative method of custom element definition via `<polymer-element>`, which is at the core of the framework. Under the covers, `<polymer-element>`

API takes care of the imperative steps above, as well as encapsulating additional framework features, albeit, not at the browser native speed originally intended by the `<element>` W3C proposal. This is the skeleton example from the Polymer project website:

<http://www.polymer-project.org/docs/polymer/polymer.html#element-declaration><sup>65</sup>

```
<polymer-element name="tag-name" constructor="TagName">
  <template>
    <!-- shadow DOM here -->
  </template>
  <script>Polymer('tag-name');</script>
</polymer-element>
```

Polymer allows a few variants on the declaration process to allow source code maintenance flexibility. The script can be referenced from an external resource either inside or outside the `<polymer-element>` tag, or not at all if the element is simple, static HTML and CSS. The `Polymer()` global JavaScript method can also be used for an imperative registration if desired.

For details on alternate element registration syntax, it's best to refer directly to the Polymer API developer guide given the framework's state of flux:

<http://www.polymer-project.org/docs/polymer/polymer.html><sup>66</sup>

In fact, when it comes to Polymer API details and syntax usage, we suggest your first resource is the official Polymer documentation. Most technical books on JavaScript frameworks have the luxury of discussing APIs that have matured and see production use throughout the Internet. So it is safe for them to include detailed API usage, and be reasonably certain that it's not likely to change for at least a year or two into the future.

When discussing pre-alpha technology, we don't have such a luxury. Our approach must be to discuss such technology at a level of abstraction above exact syntactic detail in order to explain the concepts without becoming inaccurate within a few months. The concepts will last much longer than the details.

## Attributes as API

This should sound familiar since AngularJS when used as a UI component library framework uses the same with custom element directives. `<polymer-element>` has a handful of "reserved" attributes for passing basic API parameters to the custom element declaration. Of these, `name` is currently the one that is required. The `name` attribute value becomes the name of the custom element and must include a hyphen in it to keep consistent with the `document.registerElement()` API. As mentioned previously, the prefix before the hyphen should be a unique three or more letters that distinguish your component library from others.

---

<sup>65</sup><http://www.polymer-project.org/docs/polymer/polymer.html#element-declaration>

<sup>66</sup><http://www.polymer-project.org/docs/polymer/polymer.html>



The `extends` attribute value would be any custom or built-in element to extend in an object oriented fashion. This API is optional, and is analogous to the `document.registerElement({extends:'elem'})` parameter. With a `<polymer-element extends="...">` definition, the parent element properties and methods are not just inherited, but also enhanced with data-binding unlike the standard `registerElement()`.

Another object-oriented feature Polymer currently provides is the ability to invoke an overridden method via `this.super()` when called from within the method override of the extended element.

Use of the optional `constructor` attribute value results in a JavaScript constructor method for the custom element being placed on the global namespace. Users can then imperatively create element instances via the new `ElemName()` operator. If you inspect the global DOM object with your developer tools, you will see constructor methods for all of the standard HTML5 elements already available. Using the `constructor="ElemName"` API just adds to the list.

The optional `attributes` attribute is a powerful way to include a space separated list of property names to be exposed as public properties or methods for configuring a custom-element instance. This is analogous to AngularJS' attribute directives. Public element properties (attributes) declared this way get the benefit of two-way data-binding.

`<polymer-element name="my-elem" attributes="name title age">` becomes

`<my-elem name="Joe" title="president" age="19">` or

`<my-elem name="{{dataName}}" title="{{dataTitle}}" age="{{dataAge}}">`

The second `my-elem` instance has data-bound properties just like you might see in AngularJS markup. One difference is that any undeclared defaults are set to null in Polymer, whereas, they are undefined in AngularJS.

Just as Polymer offers an imperative method of custom element definition, so do they offer imperative public property definition via an object named “publish” on the second parameter to a call to `Polymer()`. Using the imperative option is currently the only way to set default values other than null for public properties.

Speaking of public property (HTML attribute) values, they all must be strings, and Polymer will attempt to deserialize to the appropriate JavaScript type such as boolean, number, array or object if not a string. It's where the temptation to have attribute API transfer lengthy JSON arrays or object literals which leads to the anti-pattern displayed above of making “everything” an element. This is the case whether using Polymer, AngularJS, or any other declarative framework. HTML that becomes an unreadable mess full of JSON defeats the purpose of declarativeness. Do yourself a favor and avoid this temptation by using the rule-of-thumb that primitives can be handled declaratively and non-primitives *should* be handled imperatively.

The last and least of the optional attributes at this time of writing is `noscript`. It's a boolean attribute, so there is no need to include it as `noscript="true"`. Including the `noscript` attribute on a `<polymer-element>` declaration indicates that there is no need to invoke any JavaScript for all the Polymer bells and whistles on the element which is the default. You would use `noscript` in cases where

all you need is an element with some custom HTML and CSS wrapped in a `<template>` tag, but no custom logic.

## Lifecycle Callbacks

Polymer offers a *superset* of the of the custom element lifecycle callbacks that are part of the W3C specification which you would normally define on a custom element prototype object. The names are the same as the specification callbacks with the exception that they are shortened by dropping the “Callback” part in the Polymer:

```
ElemProto.createdCallback() ==> PolyElemProto.created()
```

```
ElemProto.attachedCallback() ==> PolyElemProto.attached()
```

```
ElemProto.detachedCallback() ==> PolyElemProto.detached()
```

```
ElemProto.attributeChangedCallback() ==> PolyElemProto.attributeChanged()
```

Polymer element definitions provide two additional callback methods specific to lifecycle states of the framework code execution not included in the W3C specification. The `ready()` callback is executed in between the `created()` and `attached()` callbacks. More specifically, `ready()` is called when all of the Polymer framework tasks beyond simple instantiation of an element have concluded including Shadow DOM creation, data-binding setup and event listener setup, but the element has yet to be inserted into the browser DOM. You might use this callback to provide any modifications or additions to the Polymer framework tasks.

The `domReady()` callback is executed after the `attached()` callback and after any child elements within the custom element have been attached to the DOM and are ready for any interaction. This is analogous to when AngularJS would call the `postLink()` method during a directive lifecycle. Therefore, you would include logic in this callback which depends on a Polymer element *hierarchy* that is attached to the DOM and is functional.

At the top of the Polymer element hierarchy in the DOM there is the `polymer-ready` event that is fired by the framework when all custom Polymer elements in the DOM have been registered and upgraded from `HTMLUnknownElement` status. If you needed to interact with Polymer elements from outside of the framework, you would set up a listener via something typical like:

```
window.addEventListener('polymer-ready', evtHandlerFn);
```

## Other Polymer Framework API Features

Wrapping a custom element with the Polymer framework is in many ways similar to wrapping a common element with jQuery. You get all kinds of stuff you may or may not need. We will run through the list briefly and refer you to the official documentation for exact details of Polymer’s implementation at the time of reading.

The Polymer API includes a method for observing mutations to child elements:

```
/*custom element instance*/ this.onMutation(childElem, callbackFn)
```

Also included in the custom element declaration API are inline event handler declarations with a syntax of `on-eventname="{{eventHandler}}"`. There is nothing special here. The idea is the same as AngularJS built-in event directives such as `ng-click="handlerFn"`. Rather than provide the handler function on the `$scope` object in the AngularJS controller, you provide the matching handler in the element configuration object.

You can also set the equivalent to AngularJS `$scope.$watch('propertyName', callbackFn)` by defining a function on the configuration object and following their convention that adds “Changed” to the property name followed by the function definition:

```
<polymer-element attributes="age" ...>
customElemConfigObj.ageChanged = function(oldVal, newVal){...}
```

They’ve even copied the `$scope.$watchGroup()` method from AngularJS and renamed it `observe = {object of properties}`.

Custom events can be triggered using:

```
/*custom element instance*/ this.fire('eventName', paramsObj);
```

Template nodes with id attributes can be referenced via the `$` hash like so:

```
<input type="text" id="elemId">
/*custom element instance*/ this.$.elemId;
```

Polymer’s version of AngularJS’ `$timeout` service or `window.setTimeout` is called `job`:

```
/*custom element instance*/ this.job('jobName', callbackFn, delayInMS);
```

## Polymer and `<template>` Tags with Data Binding

According to the Polymer documentation, the contents of an outer `<template>` tag within a Polymer element definition will become the tag’s shadow DOM upon instantiation. Any nested `<template>` tags can make use of a limited set of logic found in most UI template libraries like Handlebars.js, or certain AngularJS directives. Here are the available template functions at the time of writing along with the AngularJS equivalents:

Iterating over an array with Polymer:

```
<template repeat="{{item in menuItems}}">
  <li><a href="{{item.url}}">{{item.text}}</a></li>
</template>
```

AngularJS:

```
<li ng-repeat="item in menuItems"><a href="{{item.url}}">{{item.text}}</a></li>
```

Binding a View-Model object to `<template>` contents with Polymer:

```
<template bind="{{menuItem}}">
  <li><a href="{{ menuItem.url }}">{{ menuItem.text }}</a></li>
</template>
```

AngularJS:

```
<li ng-controller="menuItemCtrl"><a href="{{menuItem.url}}">{{menuItem.text}}</a></li>
```

This assumes the controller has an associated `$scope.menuItem` object. Template binding with AngularJS can also be more granular or specialized with `ngBind` for a single datum or `ngBindTemplate` for multiple `$scope` values.

Conditional inclusion with Polymer:

```
<template if="{{someValueIsTrue}}">
  <span>insert this</span>
</template>
```

AngularJS:

```
<span ng-if="someValueIsTrue">insert this</span>
```

Conditional attributes with Polymer:

```
<input type="checkbox" checked?="{{booleanValue}}">
```

AngularJS:

```
<input type="checkbox" ng-checked="booleanValue">
```

Polymer and AngularJS have almost identical pipe syntax “|” for filtering value or expression output. The major difference between the frameworks is that AngularJS filters are defined as modules that can be dependency injected anywhere in the application, whereas, Polymer filters must be defined on the custom element object. So it is unclear with Polymer how you would go about sharing a filter among different element types other than by referencing a global.

Speaking of globals, the Polymer documentation “suggests” that in keeping with their opinion that “everything is an element” any application or global level functionality or data should be defined as a custom element. In other words, the framework has no concept of models or controllers because everything is part of the view. Does anyone remember IE5 or earlier where you could make proprietary database queries directly in your markup? We’ll discuss the issue of UI patterns and anti-patterns in a later section.

You may have noticed that in cases where Polymer uses the `<template>` tag for binding a data scope, conditional or iterator, AngularJS does the same with a directive directly on the contained element. This is unfortunate in Polymer’s case because not only does this dependency typically result in two extra lines of source code, but it also adds another cross browser incompatibility. Internet Explorer will not be standards compliant in supporting the `<template>` tag until IE 13 at the earliest. This

means that on that platform any template content that references external assets will try to resolve them prematurely, and you would not be able to handle one of the most common iterator situations which is repeating `<tr>` table row elements inside a `<table>` tag. Polymer's "workaround" which is to include a "template" and a "repeat" attribute directly on a `<tr>` tag is unacceptable from a code maintenance standpoint.

This rounds out discussion of the Polymer framework features worth mentioning. This is not to say the other features are worthless. They just do not directly enhance the Web Components specifications or fill in the gaps that will still be there when Web Components are widely implemented. Left out of the discussion are "Layout Attributes" since they are labeled as experimental as of this writing. The shadow DOM styling hacks in `polymer.js` result in slightly better encapsulation than just the "polyfill" in `platform.js`, but the source is not maintainable. Finally, "touch and gesture" events which used to be called "pointer" events are a separate early specification proposal from Google the Polymer team is trying to promote with the framework.

Stepping back to a big picture view of Polymer again it should be clearer that pros of the framework are that it promotes education and experimentation with Web Components specification proposals which is very helpful in influencing the web developer community to demand that all browser vendors move towards standardization and implementation. Polymer also does a good job elucidating the gaps in the specification that frameworks will still need to cover in order to build truly full featured, high quality UI components and libraries.

The most glaring con of Polymer is that it just is not, nor will it be, a UI framework suitable for serious enterprise production applications as long as it promotes anti-patterns, disregards established front-end patterns, and ignores certain basic realities of all the browser platforms in substantial use now and likely to be in substantial use in the near future.

## Polymer Based Libraries

### Core Elements

- UI as well as non-UI elements

### Paper Elements

- UI elements encapsulated with Google's Material Design styling

### Brick

### Mozilla X-Tags

UNDER CONSTRUCTION

X-Tags from Mozilla also has a foundation of some Web Component APIs and browser polyfills from platform.js. Specifically, X-Tags is a toolkit for creating Custom Elements. It hides most of the boiler plate involved in defining them, and throws in several convenience methods for accessing and manipulating them.

The key area where X-Tags differs from Polymer is in compatibility with existing browsers. It does not include framework functionality such as data-binding, and it does not use or depend on Shadow DOM APIs and polyfills. Because the W3C Custom Elements specification is essentially one additional DOM method, `.registerElement()`, and overloading a couple others, it can be sanely polyfilled with JavaScript across all current production browsers. The exception is the additional CSS pseudo class, `:unresolved`, which is trivial to emulate if needed.

# Chapter 10 - Integrating Web Components with AngularJS using Angular Custom Element

## UNDER CONSTRUCTION

The content for this chapter is coming soon, but for now please see the README.md in the GitHub repo for [Angular Custom Element](https://github.com/dgs700/angular-custom-element)<sup>67</sup> for usage details.

- summary of relevant coverage so far
  - component architecture and BPs
  - using AngularJS directives as a functional encapsulation of the component architecture
  - general purpose header and nav components (non-portable)
  - intro to W3C specs
  - discussion of current WC frameworks
- bringing it all together
  - framework functionality with AngularJS 1.x.x
  - custom element APIs, polyfill
  - angular CE directives
  - can be used for major production projects now (focus)
- what is ACE?
  - hides boiler like polymer/x-tag
  - allows integration of framework and component at the AngularJS custom element directive level (yet to be determined boundary patterns and BPs)
  - allows this for the AngularJS 1.x.x branch (2.0 is n/a)
- how ACE is very similar to Polymer, but with certain key diffs
  - both: framework features on top of WC APIs
  - both: portable custom elements
  - P. focuses on showcasing ALL WC APIs, specs and other non-WC related items
  - ACE focuses only on production ready WC APIs (but does not exclude experimental APIs)
- Rework of the header and nav components code from ch6
  - menu item
  - dropdown
  - navbar

---

<sup>67</sup><https://github.com/dgs700/angular-custom-element>

The best way to describe the Angular Custom Element add on, is that it integrates W3C Custom Element APIs with AngularJS 1.x.x component directives. You can define custom elements in an application `.config()` block X-Tags style, and then in the matching directives, you can use AngularJS' data-binding plus all of its other framework features on them. The `$element` that is automatically injected into the directive `controller()` and `link()` methods is an *upgraded* Custom Element whose custom properties are available for data-binding via `$scope.el.customPropName`. You can think of this as a bridge to the official Component Directives that will have native Custom Element integration in AngularJS 2.0.