

ng-book

The Complete Book on AngularJS



FULLSTACK.io

Ari Lerner

ng-book

The Complete Book on AngularJS

Ari Lerner

©2013 Ari Lerner

Tweet This Book!

Please help Ari Lerner by spreading the word about this book on [Twitter](#)!

The suggested tweet for this book is:

I just bought #ngbook, the Complete Book on AngularJS! I'm ready to build advanced, modern webapps!

The suggested hashtag for this book is [#ngbook](#).

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

<https://twitter.com/search?q=#ngbook>

Also By Ari Lerner

[The Rails 4 Way](#)

[Riding Rails with AngularJS](#)

Contents

Introduction	1
Foreword	1
Acknowledgments	2
About the author	2
About this book	2
Organization of this book	3
Additional resources	3
Conventions used in this book	4
Development environment	4
The Basics of AngularJS	5
How Web Pages Get to Your Browser	5
What is a browser?	6
What is AngularJS	7
Data binding and your first AngularJS web application	9
Introducing data binding in AngularJS	10
Simple data-binding	12
Best data-binding practices	15
Modules	17
Scopes	18
The \$scope view of the world	18
It's just HTML	20
What can scopes do?	21
\$scope lifecycle	22
Directives and scopes	23
Controllers	24
Expressions	30
Interpolating a string	31
Filters	35

CONTENTS

Introduction to Directives	44
Directives: Custom HTML Elements and Attributes	46
Passing Data Into a Directive	55
Built In Directives	64
Basic ng attribute directives	64
Directives with child scope	68
Directive's Explained	87
Directive Definition	87
Directive Scope	94
Angular JS Life Cycle	106
Angular module loading	112
Configuration	112
Run blocks	114
Multiple views and routing	116
Installation	116
Layout template	117
Routes	118
\$location service	122
Routing modes	125
Other advanced routing topics	128
Form validation	128
Dependency injection	141
Annotation by inference	143
Explicit annotation	143
Inline annotation	144
\$inject api	144
ngMin	146
Services	148
Registering a service	149
Using services	150
Options for creating services	155
Communicating with the outside world: XHR and server-side communication	164
Using \$http	164
Configuration object	169
Response object	171
Caching http requests	171
interceptors	173
Configuring the \$httpProvider	175

CONTENTS

Using \$resource	176
Installation	177
Using \$resource	177
Custom \$resource methods	182
\$resource configuration object	183
\$resource services	185
Using Restangular	187
The what and the why	188
Installation	189
Intro to the Restangular object	190
Using Restangular	191
Configuring Restangular	194
XHR in practice	200
Cross-origin and same-origin policy	200
JSONP	200
Using CORS	202
Server-side proxies	205
Working with json	206
Working with xml	206
Authentication with AngularJS	208
Talking to MongoDB	215
Promises	218
What's a promise?	218
Why promises?	219
Promises in Angular	220
chaining requests	225
Server communication	228
Custom Server-side	228
Install NodeJS	228
Install Express	229
Calling APIs	233
Server-less with Amazon AWS	237
AWSJS + Angular	238
Getting started	239
Introduction	241
Installation	242
Running	243
User authorization/authentication	244
UserService	250
All aboard AWS	252

CONTENTS

AWSService	256
Starting on dynamo	259
\$cacheFactory	259
Saving our currentUser	260
Uploading to s3	264
Handling file uploads	266
Querying dynamo	270
Showing the listing in HTML	271
Selling our work	272
Using Stripe	274
Conclusion	279
Testing	280
Why test?	280
Testing strategies	280
Getting started testing	281
Types of AngularJS tests	281
Getting started	283
Initializing Karma config file	284
Configuration options	287
Using RequireJS	294
Jasmine	297
Expectations	298
End to end introduction	304
Mocking and test helpers	316
Mocking the \$httpBackend	317
Testing an app	325
Testing events	350
Continuous Integration for angular	351
Protractor	352
Configuration	354
Configuration options	355
Writing tests	357
Page Objects	359
Angular Animation	361
Installation	361
How it works	362
Using CSS3 Transitions	363
Using CSS3 Animations	364
Using JavaScript animations	366
Animating built-in directives	367
Building custom animations	388

CONTENTS

Integrating with third-party libraries	395
The digest loop and \$apply	398
\$watch list	399
Dirty checking	399
\$watch	401
\$watchCollection	403
The \$digest loop in a page	404
\$evalAsync list	405
\$apply	406
When to use \$apply()	407
Demystifying angular	409
How the view works	410
Essential AngularJS extensions	412
AngularUI	412
Installation	413
ui-router	413
ui-utils	422
Mobile apps	426
Responsive web apps	426
Interaction	426
Native applications with Cordova	433
Getting started with Cordova	434
Including Angular	444
Building with Yeoman	445
Localization	455
angular-translate	455
Installation	455
Teaching your app a new language	456
Multi-language support	457
Switching the language at runtime	458
Loading languages	459
angular-gettext	460
Installation	460
Usage	461
String extraction	462
Translating our strings	464
Compiling our new language	468
Changing languages	469
Security	471

CONTENTS

Strict Contextual Escaping, the \$sce service	471
Whitelisting urls	474
Blacklisting urls	475
\$sce API	476
Configuring \$sce	478
Trusted context types	478
AngularJS and Internet Explorer	480
Ajax caching	482
SEO with AngularJS	482
Getting angular apps indexed	483
Server-side	483
Options for handling SEO from the server-side	484
Taking snapshots	487
Using Zombie.js to grab html snapshots	487
Using grunt-html-snapshot	490
Prerender.io	491
.	492
Building Angular Chrome apps	493
Understanding the Chrome apps	493
Building our Chrome app	494
Building the skeleton	495
manifest.json	496
tab.html	497
Loading the app in Chrome	498
The main module	499
Building the homepage	499
Sign up for wunderground's weather API	503
A settings screen	510
Implementing a User service	512
City autofill/autocomplete	516
Sprinkling in timezone support	520
Debugging AngularJS	524
Debugging from the DOM	524
debugger	526
Angular Batarang	526
Next steps	532
jqLite and jQuery	532
Essential tools to know about	534
Grunt	534
grunt-angular-templates	538

CONTENTS

Lineman	543
Bower	546
Yeoman	551
Configuring the angular generator	558
Testing our app	559
Packaging our app	560
Packaging our templates	560
Our app	563
Downloading angular	566
Booting it up	568
Modules	569
Creating our-calendar directive	584
Drawing the calendar	584

Dedication

I dedicate this book to my parents, Lisa and Nelson Lerner for without their support and encouragement none of this would have been possible.

Special thanks

To the lovely Q for the constant motivation and incredibly talented editing and my cofounder and friend, Nate Murray.

Introduction

Foreword

I've become somewhat numb to all of the JavaScript libraries and frameworks being released on a seemingly daily basis. While the ability to choose from a variety of libraries and frameworks is a good thing, including too many scripts in an application can be a bad thing for maintenance – at least in my opinion. I've always been concerned about the dependencies that are created as more and more scripts are added into an application and often longed for a single script (or two) that could provide the core functionality I wanted.

When I first heard about AngularJS it caught my attention immediately because it appeared to offer a single framework that could be used to build a variety of dynamic, client-centric applications. After researching it more my initial impressions were confirmed and I was hooked. AngularJS includes a robust set of features and offers a way to break-up code into modules which is good for re-use, maintenance, and testability. It provides key features such as support for DOM manipulation, animations, templating, two-way data binding, routing, history, Ajax, testing, and much more.

While having a core framework to build on is great, it can also be intimidating and challenging to learn. As I dove into AngularJS I became overwhelmed with different topics and quickly became a little frustrated and wondered if it was the framework for me. What was a service and how was it different from a factory? How did scope fit into the overall picture? What was a directive and why would I use one? Putting the pieces together and seeing the big picture was the initial hurdle that I had to get over. It definitely would've been nice to have a concise resource to consult that flattened out the learning curve.

Fortunately, you have an excellent resource at your disposal in The ng-book: The complete Book on AngularJS that will help make you productive right away. Ari Lerner has taken the knowledge and expertise that he's gained and laid it out in a way that is easy to follow and understand. If you're looking to learn more about data binding, how "live" templates work, the process for testing AngularJS applications, the role of services and factories, how scope and controllers fit together, and much more then you're in the right place. AngularJS is an extremely powerful and fun framework to work with and the examples shown throughout this book will help you get up-to-speed quickly on the framework. Best of luck with your AngularJS development projects!

Dan Wahlin Wahlin Consulting <http://weblogs.asp.net/dwahlin>¹ <http://twitter.com/DanWahlin>²

¹<http://weblogs.asp.net/dwahlin>

²<http://twitter.com/DanWahlin>

Acknowledgments

First, I want to thank everyone who has encouraged me along the way to write this book. Whomever said authoring a book was easy has not written one themselves.

I want to personally thank Q Kuhns for her tireless grammatical editing and support, Erik Trom for his patience and attention to detail, Nate Murray for his neverending optimism and clarity of thought.

Big thanks go out to the entire [Hack Reactor](#)³ staff and the Summer class of 2013 for giving me the space to explore how to teach AngularJS in a formal setting.

I want to thank my 30x500 alumni, Sean Iams, Michael Fairchild, Bradly Green, Misko Hevery, and the Airpair team.

About the author

Ari Lerner is the co-founder of [fullstack.io](#)⁴ based in San Francisco, CA. Working at AT&T's innovation center in Palo Alto, CA for a total of five years, building large-scale cloud infrastructure and helping architect the bleeding-edge developer center, including designing publicly facing APIs and developer toolsets.

He and his team was featured in the AT&T annual report for 2012 for their work in modernizing the company workflow and internal processes.

He left his job at AT&T to pursue building fullstack.io, a fullstack software development product and services company that specializes in the entire stack, from hardware to the browser.

He lives in San Francisco with his dog and lovely girlfriend.

About this book

ng-book: The Complete Book on AngularJS is packed with the solutions you need to be an [AngularJS](#)⁵ ninja. AngularJS is an advanced front-end framework released by the team at [Google](#)⁶. It enables you to build a rich front-end experience, quickly and easily.

The Absolute Beginners Guide to AngularJS gives you the cutting-edge tools you need to get you up and running on AngularJS and creating impressive web experiences in no time. It addresses challenges and provides real-world techniques that you can use immediately in your web applications. In this book, we will cover topics that enable you to build professional web apps that perform perfectly. These topics include:

³<http://www.hackreactor.com>

⁴<http://fullstack.io>

⁵<http://angularjs.org>

⁶<http://google.com>

- Interacting with a RESTful web service
- Building custom reusable components
- Testing
- Asynchronous programming
- Building services
- Providing advanced visualizations
- And much more

The goal of this book is not only to give you a deep understanding of how AngularJS works, but also to give you professional snippets of code so that you can build and modify your own applications.

With these tools and tests, you can dive into making your own dynamic web applications with AngularJS while being confident that you are building a scalable application.

Audience

We have written this book for those who have never used AngularJS to build a web application and are curious about how to get started with an awesome JavaScript framework. We assume that you have a working knowledge of HTML and CSS and a familiarity with basic JavaScript (and possibly other JavaScript frameworks).

Organization of this book

This book covers the basics of getting started and aims to get you comfortable with writing dynamic web applications with AngularJS right away.

Then we'll take a look at how AngularJS works and what sets it apart from other popular JavaScript web frameworks. We'll dive deeply into detail about the underpinnings of the flow of an AngularJS application.

Finally, we'll take all of our knowledge and build a relatively large application.

Additional resources

We'll refer to the official documentation on the [AngularJS⁷](http://angularjs.org) website. The official AngularJS documentation is a great resource, and we'll be using it quite often.

We suggest that you take a look at the AngularJS API documentation, as it gives you direct access to the recommended methods of writing AngularJS applications. Of course, it also gives you the most up-to-date documentation available.

⁷<http://angularjs.org>

Conventions used in this book

Throughout this book, you will see the following typographical conventions that indicate different types of information:

In-line code references will look like: <h1>Hello</h1>.

A block of code looks like so:

```
1 var App = angular.module('App', []);
2
3 function FirstCtrl($scope) {
4   $scope.data = "Hello";
5 }
```

Any command at the command line will look like:

```
1 $ ls -la
```

Any command in the developer console in Chrome (the browser with which we will primarily be developing) will look like:

```
1 > var obj = {message: "hello"};
```

Important words will be shown in **bold**.

Finally, tips and tricks will be shown as:

Tip: This is a tip

Development environment

In order to write any applications using AngularJS, we first need to have a comfortable development environment. Throughout this book, we'll be spending most of our time in two places: our text editor and our browser.

We'll refer to the text editor as your `editor` throughout the book, while we'll refer to the browser as the `browser`. For this book, we highly recommend you download the Google Chrome browser, as it provides a great development environment using the developer tools.

We'll only need to install a few libraries to get going. To run our tests, we'll need the `Karma` library and `node.js`. It's also a good idea to have `git` installed, although this is not a strict requirement.

This book won't cover how to install NodeJS. Visit nodejs.org⁸ for more information.

While most of our work will be done in the browser, parts of this book will focus on building Restful APIs to service our front end with data endpoints.

⁸<http://nodejs.org>

The Basics of AngularJS

In this chapter, we will cover the very basics of what AngularJS is and how it works. We'll look at the main features of the framework and how they work.

The goal of this chapter is to get you comfortable with the terminology and the technology and to give you an understanding of how AngularJS works. In this chapter, we'll start putting the pieces together that will enable you to build an AngularJS application, even if you've never written one before.

How Web Pages Get to Your Browser

Let's think of the Internet as a post office. When you want to send a letter to your friend, you first write your message on a piece of paper. Then you write your friend's address on an envelope and place the letter inside.

When you drop the letter off at the post office, the mail sorter looks at the postal code and address and tries to find where your friend lives. If she lives in a giant apartment complex, the postal service might deliver the mail to your friend's front desk and let the building's employees sort it out by apartments.

The Internet works in a similar way. Instead of a bunch of houses and apartments connected by streets, it is a bunch of computers connected by routers and wire. Every computer has a unique address that tells the network how to reach it.

Similar to the apartment building analogy above, where we have many apartments that share the same address, several computers can exist on the same network or router (as when you connect to wifi at a Starbucks). In this case, your computer shares the same IP address as the other computers. Your computer can be reached individually, however, by its "internal IP address" (like the *apartment number* in our analogy), about which the router is aware (as the apartment building employees in our analogy are aware of your friend's apartment number).

IP stands for Internet Protocol. An IP address is a numerical identifier assigned to each device participating in a network. Computers, printers, and even cell phones have IP addresses.

There are two main types of IP addresses: *ipv4* and *ipv6* addresses. The most common addresses today are *ipv4* addresses. These look like 192.168.0.199. *Ipv6* addresses look like 2001:0db8:0000:0000:0000:ff00:0042:8329.

When you open your web browser on your computer and type in `http://google.com`, your web browser “asks” the internet (more precisely, it “asks” a DNS server) where `google.com`’s address is. If the DNS server knows the IP address you’re looking for, it responds with the address. If not, it passes the request along to other DNS servers until the IP address is found and served to your computer. You can see the DNS server response by typing this code into a terminal:

```
1 $ dig google.com
```

If you are on a Mac, you can open the terminal program called `Terminal`, which is usually located in your `/Applications/Utilities`. If you are using Windows, you can find your terminal by going to the Start Button and typing `cmd` in the Run option.

Once the DNS server responds with the IP address of the computer you’re trying to reach (i.e., once it finds `google.com`), it also sends a message to the computer located at that IP address asking for the web page you’re requesting.

Every path of a web page is written with its own HTML (with a few exceptions). For example, when your browser requests `http://google.com`⁹, it receives different HTML than if it were to request `http://google.com/images`¹⁰.

Now that your computer has the IP address it needs to get `http://google.com`, it asks the Google server for the HTML it needs to display the page.

Once the remote server sends back that HTML, your web browser *renders* it (i.e., the browser works to make the HTML look as `google.com` is designed to look).

What is a browser?

Before we jump straight into our coverage of Angular, it’s important to know what your browser is doing when it renders a web page.

There are many different web browsers; the most common browsers today include Chrome, Safari, and Internet Explorer. At their core, they all basically do the same thing: fetch web pages and display them to the user.

Your browser gets the HTML text of the page, parses it into a structure that is internally meaningful to the browser, lays out the content of the page, and styles the content before displaying it to you. All of this work happens behind the scenes.

Our goal as web developers is to build the structure and content of our web page so that the browser will make it look great for our users.

With Angular, we’re not only building the structure, but we’re constructing the interaction between the user and our app as a web application.

⁹<http://google.com>

¹⁰<http://google.com/images>

What is AngularJS

The official AngularJS introduction describes AngularJS as a:

client-side technology, written entirely in JavaScript. It works with the long-established technologies of the web (HTML, CSS, and JavaScript) to make the development of web apps easier and faster than ever before.

It is a framework that is primarily used to build single-page web applications. AngularJS makes it easy to build interactive, modern web applications by increasing the level of abstraction between the developer and common web app development tasks.

The AngularJS team describes it as a “structural framework for dynamic web apps.”

AngularJS makes it incredibly easy to build web applications; it also makes it easy to build complex applications. AngularJS takes care of advanced features that users have become accustomed to in modern web applications, such as:

- Separation of application logic, data models, and views
- Ajax services
- Dependency injection
- Browser history (makes bookmarking and back/forward buttons work like normal web apps)
- Testing
- And more

How is it different?

In other JavaScript frameworks, we are forced to extend from custom JavaScript objects and manipulate the DOM from the outside in. For instance, using [jQuery¹¹](#), to add a button in the DOM, we'll have to *know* where we're putting the element and insert it in the appropriate place:

```
1 var btn = $("<button>Hi</button>");  
2 btn.on('click', function(evt) { console.log("Clicked button") });  
3 $("#checkoutHolder").append(btn);
```

¹¹<http://jquery.com/>

Although this process is not complex, it requires the developer to have knowledge of the entire DOM and force our complex logic inside JavaScript code to manipulate a foreign DOM.

AngularJS, on the other hand augments HTML to give it native Model-View-Controller (MVC) capabilities. This choice, as it turns out, makes building impressive and expressive client-side applications quick and enjoyable.

It enables you, the developer, to encapsulate a portion of your entire page as one application, rather than forcing the entire page to be an AngularJS application. This distinction is particularly beneficial if your workflow already includes another framework or if you want to make a portion of the page dynamic while the rest operates as a static page or is controlled by another JavaScript framework.

Additionally, the AngularJS team has made it a point to keep the library small when compressed, such that it does not impose heavy penalties for using it (the compressed, minified version weighs in under 9KB at the time of this writing). This feature makes AngularJS particularly good for prototyping new features.

License

The AngularJS source code is made freely available on [Github¹²](#) under the MIT license. That means you can contribute to the source and help make AngularJS even better.

In order to contribute, the Angular team has made the process relatively straightforward. Any major changes should be discussed on the mailing list <https://groups.google.com/forum/?hl=en#!forum/angular¹³>, thus making the potential change available for modification, allowing other developers to join in the discussion, and preventing code/work duplication.

More information on the contribution process can be found here: <http://docs.angularjs.org/misc/contribute¹⁴>.

¹²<http://github.com>

¹³<https://groups.google.com/forum/?hl=en#!forum/angular>

¹⁴<http://docs.angularjs.org/misc/contribute>

Data binding and your first AngularJS web application

Hello world

The quintessential place to start writing an AngularJS app is with a *hello world* application. To write our *hello world* application, we'll start with the simplest, most basic HTML we can possibly write.

We'll take a more in-depth look into AngularJS as we dive into the framework. For now, let's build our *hello world* application.

```
1 <!DOCTYPE html>
2 <html ng-app>
3 <head>
4   <title>Simple app</title>
5   <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.0rc1/angular.js"></script>
6 </head>
7 <body>
8   <input ng-model="name" type="text" placeholder="Your name">
9   <h1>Hello {{ name }}</h1>
10 </body>
11 </html>
```

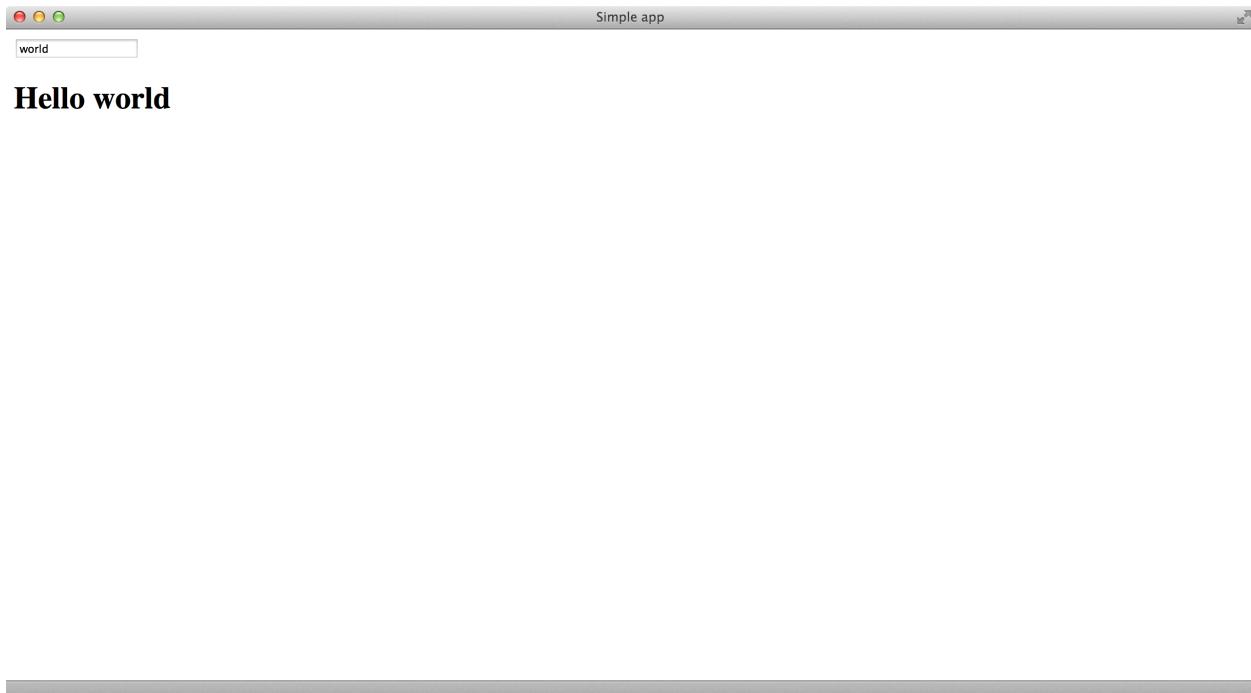


Figure 1

Although this demo isn't incredibly interesting or exciting, it does show one of the most basic and impressive features of AngularJS: *data binding*.

Introducing data binding in AngularJS

In classic web frameworks, such as Rails, the controller combines data from models and mashes them together with templates to deliver a view to the user. This combination produces a single-way view. Without building any custom JavaScript components, the view will only reflect the data the model exposes at the time of the view rendering. At the time of this writing, there are several JavaScript frameworks that promise automatic data binding of the view and the model.

AngularJS takes a different approach. Instead of merging data into a template and replacing a DOM element, AngularJS creates *live* templates as a view. Individual components of the views are dynamically interpolated live. This feature is arguably one of the most important in *AngularJS* and allows us to write the *Hello world* app we just wrote in only 10 lines of code without a single line of JavaScript.

This feature works by simply including `angular.js` in our HTML and explicitly setting the `ng-app` attribute on an element in the DOM. The `ng-app` attribute declares that everything inside of it belongs to this Angular app.

That's how we can nest an Angular app inside of a web app. The only components that will be affected by Angular are the DOM elements that we declare inside of the one with the `ng-app` attribute.



Views are interpolated when the view is evaluated with one or more variable substitutions; the result is that the variables in our string are replaced with values. For instance, if there is a variable named `name` and it is equal to "Ari", string interpolation on a view of "Hello {{ name }}" will return "Hello Ari".

Automatic data binding gives us the ability to consider the view to be a projection of the model state. Any time the model is changed in the client-side model, the view reflects these changes without writing any custom code. It just *works*.

In the *Model View Controller* (or *MVC*) view of the world, the controller doesn't have to worry about being in the mix of rendering the view. This fact virtually eliminates the concern of separating view and controller logic, and it has the corollary side effect of making testing simple and enjoyable.



MVC, which stands for Model-View-Controller, is a software architecture pattern that separates representation from user interaction. Generally, the *model* consists of application data and functions that interact with it, while the *view* presents this data to the user; the *controller* mediates between the two.



This [separation presentation¹⁵](#) makes a clear division between objects in our web app so that the view doesn't need to *know* how to save an object; it just needs to know how to display it. Meanwhile, the model doesn't need to interact with the view; it just needs to contain the data and methods to manipulate the view. The controller is where we'll place the logic to bind the two together.

Without getting into the source (available at AngularJS.org¹⁶), Angular simply remembers the value that the model contains at any given time (in our example from *hello world*, the value of `name`).

When Angular thinks that the value could change, it will call `$digest()` on the value to check whether the value is "dirty." Hence, when the Angular runtime is running, it will look for potential changes on the value.

This process is `dirty checking`. Dirty checking is a relatively *efficient* approach to checking for changes on a model. Every time there could be a potential change, Angular will do a dirty check inside its event loop (discussed in-depth in the *under the hood* chapter) to ensure everything is consistent.

When using frameworks like KnockoutJS, which attaches a function to the change event (known as change listeners), the process is significantly more complex and relatively more inefficient. Dealing with change coalescence, dependency tracking, and the multitude of event firing is complex and often causes problems in performance.

¹⁵<http://martinfowler.com/eaaDev/uiArchs.html>

¹⁶<http://angularjs.org>



Although there are more efficient ways to do it, it always works in every browser and is predictable. Additionally, a lot of software that needs speed and efficiency use the dirty checking approach.

AngularJS removes the need to build complex and novel features in javascript to build fake automatic synchronization in views.

Simple data-binding

Reviewing the code we just wrote, what we did was bind the “name” attribute to the input field using the `ng-model` directive on the containing model object (`$scope`).

Simply, what this means is that whatever value is placed in the input field will be reflected in the model object.



The *model object* that we are referring to is the `$scope` object. The `$scope` object is simply a javascript object whose properties are all available to the view and the controller can interact with. Don’t worry if this doesn’t make sense quite yet, it’ll make sense with a few examples.

Bi-directional in this context means that if the view changes the value, the model *observes* the change through dirty-checking, and if the model changes the value, then the view will be updated with the change.

To set up this binding, we used the `ng-model` function on the input, like so:

```
1 <input ng-model="name" type="text" placeholder="Your name">
2 <h1>Hello {{ name }}</h1>
```

Now that we have a binding set up (yes, it’s *that* easy), we can see how the view changes the model. When the value in the *input* field changes, the `person.name` will be updated and the view will reflect the change.

Now we can see that we’re setting up a bi-directional binding purely in the view. To illustrate the bi-directional binding from the other way (back-end to front-end), we’ll have to dive into Controllers, which we’ll cover shortly.

Just as `ng-app` declares that all elements inside of the DOM element it is declared upon belong to the angular app, declaring the `ng-controller` attribute on a DOM element says that all of the element inside of it belong to the controller.

To declare our above example inside of a controller, we’ll change the HTML to look like:

```
1 <div ng-controller='MyCtrl'>
2   <input ng-model="name" type="text" placeholder="Your name">
3   <h1>Hello {{ name }}</h1>
4 </div>
```

In this example, we'll create a clock that will tick every second (as clocks usually do) and change the data on the `clock` variable:

```
1 function MyController($scope) {
2   var updateClock = function() {
3     $scope.clock = new Date();
4   };
5   setInterval(function() {
6     $scope.$apply(updateClock);
7   }, 1000);
8   updateClock();
9 }
```



The controller function takes one parameter, the `$scope` of the DOM element. This `$scope` object is available on the element and the controller (as we can see) and it will be the bridge for how we'll communicate from the controller to the view.

In this example, as the timer fires, it will call the `updateClock` function which will set the new `$scope.clock` variable to the current time.

We can show the `clock` variable that's attached on the `$scope` in the view simply by surrounding it in `{{ }}`:

```
1 <div ng-controller="MyController">
2   <h5>{{ clock }}</h5>
3 </div>
```

At this point, our sample webapp looks like:

```
1 <!doctype html>
2 <html ng-app>
3   <head>
4     <script
5       src="https://ajax.googleapis.com/ajax/
6         libs/angularjs/1.2.0rc1/angular.js">
7     </script>
8   </head>
9   <body>
10    <div ng-controller="MyController">
11      <h1>Hello {{ clock.now }}!</h1>
12    </div>
13    <script type="text/javascript">
14      function MyController($scope) {
15        $scope.clock = {};
16        var updateClock = function() {
17          $scope.clock.now = new Date();
18        };
19        setInterval(function() {
20          $scope.$apply(updateClock);
21        }, 1000);
22        updateClock();
23      };
24    </script>
25  </body>
26</html>
```



Although this works in a single file, it will become tough to collaborate on the web app with other people as well as separate out the functionality of the different components. Instead of containing all of our code in the `index.html` file, it's usually a good idea to include javascript in a separate file.

The above code will change to:

```

1 <!doctype html>
2 <html ng-app>
3   <head>
4     <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.0rc1/angular\
5 .js"></script>
6   </head>
7   <body>
8     <div ng-controller="MyController">
9       <h1>Hello {{ clock.now }}!</h1>
10    </div>
11    <script type="text/javascript" src="js/app.js"></script>
12  </body>
13 </html>

```

We will place the javascript from above in the `js/app.js` file instead of embedding it directly into the HTML.

```

1 // In app.js
2 function MyController($scope) {
3   $scope.clock = {};
4   var updateClock = function() {
5     $scope.clock.now = new Date();
6   };
7   setInterval(function() {
8     $scope.$apply(updateClock);
9   }, 1000);
10  updateClock();
11 }

```

Best data-binding practices

Due to the nature of javascript itself and how it passes by value vs. reference, it's considered a *best-practice* in Angular to bind references in the views by an attribute on an object, rather than the raw object itself.

Best-practices in the case of the above example with our clock, would change the usage of the clock in our view to:

```
1 <!doctype html>
2 <html ng-app>
3   <head>
4     <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.0rc1/angular.js"></script>
5   </head>
6   <body>
7     <div ng-controller="MyController">
8       <h1>Hello {{ clock.now }}!</h1>
9     </div>
10    <script type="text/javascript" src="js/app.js"></script>
11  </body>
12 </html>
```

In this case, rather than updating the `$scope.clock` every second we can update the `clock.now` property. With this optimization, we can then change our backend to reflect the change with:

```
1 // In app.js
2 function MyController($scope) {
3   $scope.clock = {
4     now: new Date()
5   };
6   var updateClock = function() {
7     $scope.clock.now = new Date()
8   };
9   setInterval(function() {
10     $scope.$apply(updateClock);
11   }, 1000);
12   updateClock();
13 };
```



It's a good idea to try to place all of our bindings in the view in this manner.

Modules

In Angular, a module is the *main* way to define an AngularJS app. The module(s) of an app is where we'll contain all of our application code.

Using modules gives us a lot of advantages, such as:

- Keeps our global namespace clean
- Eases writing tests as well as keeps them clean as easy to target isolated functionality
- Eases to share code between applications
- Allows different parts of the code to be loaded in any order

The angular module api allows us to declare a module using the `angular.module` api method. When declaring a module, we'll need to pass two parameters to the method. The first will be the name of the module we are creating. The second is the list of dependencies, or injectables.

```
1 var app = angular.module('myApp', []);
```

We can always reference our module by using the same method with only one parameter. For instance, we can reference the `myApp` module like so:

```
1 // app is the same as the app in the previous example
2 // this method fetches the app
3 var app = angular.module('myApp')
```

From here, we can create our applications on top of the `app` variable. When writing large applications, we'll create several different modules to contain our logic.

For an in depth example of setting up an app with modules, refer to the modules section of [our_app](#).

Scopes

Scopes are a **core** fundamental of any angular app. They are used all over the framework, so it's important to know them and how they work.

The scopes of the application refer to the application model. Scopes are the execution context for expressions. The `$scope` object is where we define the business-logic of the application, the methods in our controllers and reference properties in the views.

Scopes serve as the `glue` between the controller and the view. Just before the view is rendered to the user, the view template is linked to the scope and the DOM is set up to notify Angular for property changes. This feature makes it easy to account for promises to be fulfilled, such as an XHR call. See the [promises](#) for more details.

Scopes are the **source-of-truth** for the application state. Because of this *live-binding*, we can depend on the `$scope` to be updated immediately when the view modifies it as well as depending on the view updating when the `$scope` changes.

`$scopes` in AngularJS are arranged in a hierarchical structure that mimics the DOM and thus are nestable and we can reference properties on parent `$scopes`.

If you are familiar with javascript, then this hierarchical concept shouldn't be foreign. When we create a new execution context in javascript, we create a new function, that effectively creates a new "local" context. The Angular concept of `$scopes` are similar in that as we create new scopes for child DOM elements, we are creating a new *execution context* for the DOM to live in.

Scopes provide the ability to `watch` for model changes. They give the developer the ability to propagate model changes throughout the application using the `apply` mechanism available on the scope. This is where expressions can be defined and executed as well as propagate events onward to other controllers and parts of the application.

It is ideal to contain the application logic in a controller and the working data on the scope of the controller.

The `$scope` view of the world

When Angular starts running and starts to generate the view, it will create a binding from the root `ng-app` element to the `$rootScope`. This `$rootScope` is the eventual parent of all `$scope` objects.

The `$rootScope` object is the closest object we have to the *global* context in an angular app. It's a bad idea to attach too much logic to this *global* context, just like it's not a good idea to dirty the javascript global scope.

This `$scope` object is a plain ole' javascript object. We can add and change properties on the `$scope` object however we see fit.

This `$scope` object is the *data model* in Angular. Unlike traditional data models, which are the gatekeepers of data and are responsible for handling and manipulating the data the `$scope` object is simply a connection between the view and the HTML. It's the *glue* between the view and the controller.

All properties found on the `$scope` object are *automatically* accessible to the view.

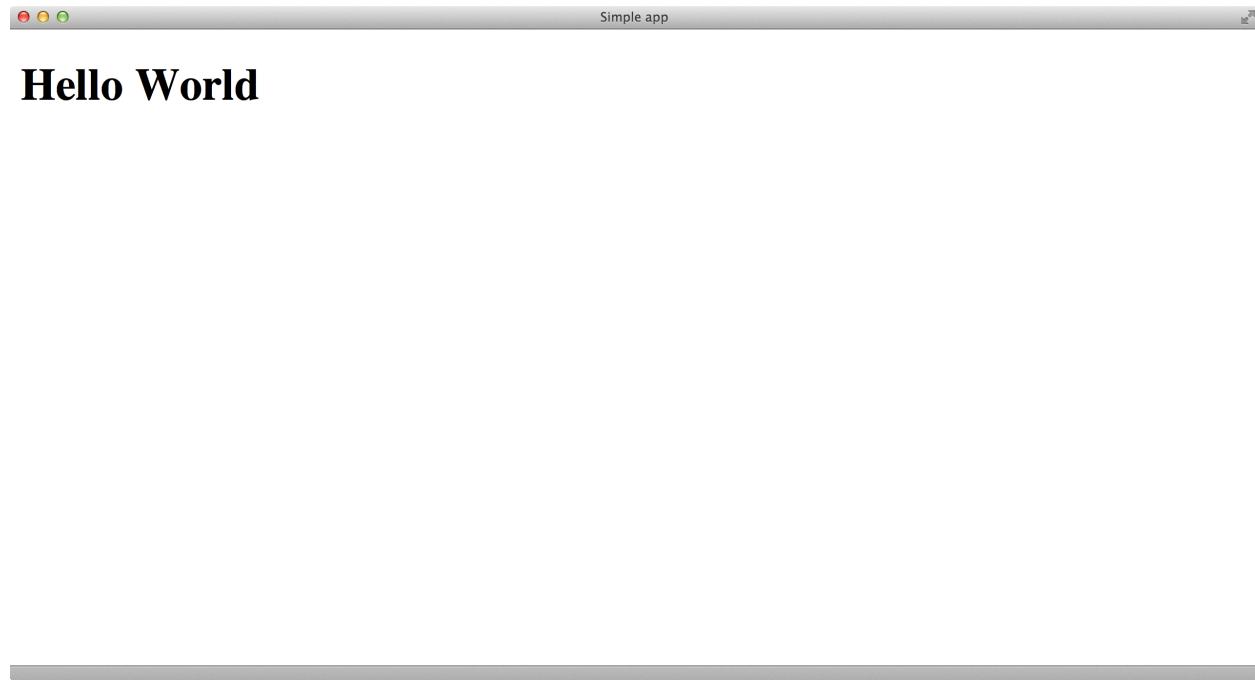
For instance, let's say we have the HTML:

```
1 <div ng-app="myApp">
2   <h1>Hello {{ name }}</h1>
3 </div>
```

The `{{ name }}` variable is expected to be a property of the containing `$scope`:

```
1 angular.module('myApp', [])
2   .run(function($rootScope) {
3     $rootScope.name = "World";
4   });

```



Simple \$rootScope binding

As we saw in the [scopes](#) chapter, this \$scope object's properties are available to be bound to the view.

It's just HTML

This HTML gets rendered and delivered to the browser for presentation. This HTML contains all standard HTML elements, both angular-specific and non angular-specific elements. For the elements that do **not** contain angular-specific declarations, they are left modified.

```
1 <h2>Hello world</h2>
2 <h3>Hello {{ name }}</h3>
```

In the previous example, the `<h2>` element is left untouched by angular, while the `<h3>` will be updated with any scope modifications.

There are the different types of markup that are able to be used in a template through Angular. These include the following:

- Directives – the elements (attributes or elements) that augment the existing DOM element into a reusable DOM component.
- Markup – the template syntax `{{ }}` binds expressions to the view.
- Filters – formatting functions available in the view
- Form controls – user input validation controls.

What can scopes do?

Scopes have the following basic functions:

- They provide observers to watch for model changes
- They provide the ability to propagate model changes through the application as well as outside the system to other components
- They can be nested such that they can isolate functionality and model properties.
- They provide an execution environment in which expressions are evaluated.

The majority of the work we'll do in developing an Angular app is building out the functionality of a scope.

Scopes are objects that contain functionality and data to use when rendering the view. It is the single source of truth for all views. You can think of scopes as `view models`.

Here's an example of scopes in practice. We'll set a variable name on the `$rootScope` and reference it in a view, like so:

```
1 angular.module("myApp", [])
2   .run(['$rootScope', function($rootScope) {
3     $rootScope.movie = "Duck Tails";
4   }]);
5 
```

And our view can now reference this `movie` property to show to the user:

```
1 <div ng-app="myApp">
2   My favorite childhood movie: {{ movie }}
3 </div>
```

Instead of placing variables on the `$rootScope`, we can explicitly create a child `$scope` object using a controller. We can attach a controller object to a DOM element using the `ng-controller` directive on a DOM element, like so:

```
1 <div ng-app="myApp">
2   <div ng-controller="MyController">
3     My favorite childhood movie: {{ movie }}
4   </div>
5 </div>
```

Now, instead of attaching the `movie` variable on the `$rootScope`, we can create a controller that will manage our variable:

```
1 angular.module("myApp", [])
2 .controller('MyController',
3 ['$scope', function($scope) {
4   $scope.movie = "Duck Tails";
5 }]);
```

The `ng-controller` directive creates a new `$scope` object for the DOM element and nests it in the containing `$rootScope`.

\$scope lifecycle

When the browser receives a javascript callback that executes *inside* of the angular execution context (for more information on the angular execution context, check out the [digest loop](#) chapter), the `$scope` will be made aware of the model mutation.



If the callback executes outside of the angular context, we can force the `$scope` to have knowledge of the change using the `$apply` method.

After the expression is evaluated and the `$digest` loop runs, the `$scope`'s watch expressions will run dirty checking (see the [digest loop](#) for more details on dirty checking).

Created

Scopes are created by the `$injector` using the `$rootScope.$scopes` are created with new controllers and directives.

Linking

When the `$scope` is linked to the view, all directives that create `$scope`'s will register their watches on the scope. These watches, just like controller watches and propagates model changes from the view to the directive.

Updates

During the `$digest` cycle, which executes on the `$rootScope`, all of the children scopes will perform dirty digest checking. All of the watching expressions are checked for any changes and the scope then calls the listener callback when they are changed.

Destruction

When a `$scope` is no longer needed, the child scope creator will need to call `scope.$destroy()` to clean up the child scope.

Note that when a scope is destroyed, the `$destroy` event will be broadcasted.

Directives and scopes

Directives, which are used all throughout our angular apps generally do not create their own `$scopes`, but there are cases when they do. For instance, the `ng-controller` and `ng-repeat` directives create their own child scopes and attach them to the DOM element.

Before we get too far, let's take a look at what controllers are and how we can use them in our applications.

Controllers

Controllers in AngularJS exist to augment the view of an AngularJS application. As we saw in our Hello world example application, we did not use a controller, but only an implicit controller.

The controller in AngularJS is a function that adds additional functionality to the scope of the view. They are used to set up an initial state and to add custom behavior to the scope object.

When a new controller is created on a page, it is passed a new \$scope generated by Angular. This new \$scope is where we can set up the initial state of the scope on our controller. Since Angular takes care of handling the controller for us, we only need to write the constructor function.

Setting up an initial controller looks like this:

```
1 function FirstCtrl($scope) {  
2   $scope.message = "hello";  
3 }
```



It is considered a *best-practice* to name our controllers [Name]Ctrl in camelcase.

As we can see, the controller method will be called with the scope AngularJS creates for us.

The observant reader will notice that this function is created in the global scope. This is usually poor form as we don't want to dirty the global namespace. To do this properly, we'll create a module and then create the controller atop our module, like so:

```
1 var app = angular.module('app', []);  
2 app.controller('FirstCtrl', ['$scope', function($scope) {  
3   $scope.message = "hello";  
4 }]);
```

The odd syntax with the [] bracket notation is a really nifty feature of AngularJS and will eliminate a lot of dependency headaches. We'll get to that in the dependency injection section. For the time being, just know that that syntax sets the dependency of the controller to rely on the \$scope provided by AngularJS.

To create custom actions we can call in our views, we can just create functions on the scope of the controller. Luckily for us, AngularJS allows our views to call functions on the \$scope, just as if we are calling data.

To bind buttons or links (or any DOM element, really), we'll use another built-in directive, `ng-click`. The `ng-click` directive binds the click event to the method (the `mouseup` browser event) to the DOM element (i.e., when the browser fires a click event on the DOM element, the method is called). Similar to our previous example, the binding looks like:

```
1 <div ng-controller="FirstCtrl">
2   <h4>The simplest adding machine ever</h4>
3   <button ng-click="add(1)" class="button">Add</button>
4   <a ng-click="subtract(1)" class="button alert">Subtract</a>
5   <h4>Current count: {{ counter }}</h4>
6 </div>
```

Both the button and the link will be bound to an action on the containing `$scope`, so when they are pressed (clicked), Angular will call the method. Note that when we are telling Angular what method to call, we're putting it in a string *with* the parentheses.

Now, let's create an action on our `FirstCtrl`.

```
1 app.controller('FirstCtrl', ['$scope', function($scope) {
2   $scope.counter = 0;
3   $scope.add = function(amount) { $scope.counter += amount; };
4   $scope.subtract = function(amount) { $scope.counter -= amount; };
5 }]);
```

Setting our `FirstCtrl` in this manner will allow us to call `add` or `subtract` functions (as we've seen above) that are defined on the `FirstCtrl` scope or a containing parent `$scope`.

Using controllers allows us to contain the logic of a single view in a single container. It's good practice to keep slim controllers. One way that we as AngularJS developers can do so is by using the dependency injection feature of AngularJS to access services.

One major difference between other JavaScript frameworks and AngularJS is that the controller is not the appropriate place to do any DOM manipulation or formatting, data manipulation, or state maintenance beyond holding the model data. It is simply the glue between the view and the `$scope` model.

Now, we've only set simple types on the `$scope` object. AngularJS also makes it possible to set objects on the `$scope` and show properties in the view.

To do this, we will simply create a `person` object on the controller `MyController` that has a single attribute of `name`:

```
1 app.controller('MyCtrl', function($scope) {  
2   $scope.person = {  
3     name: "Ari Lerner"  
4   };  
5 });
```

Now we can access this `person` object in any child element of the `div` where `ng-controller='MyCtrl'` is written *because* it is on the `$scope`.

For instance, now we can simply reference `person` or `person.name` in our view.

```
1 <div ng-app="myApp">  
2   <div ng-controller="MyController">  
3     <h1>{{ person }}</h1>  
4     and their name:  
5     <h2>{{ person.name }}</h2>  
6   </div>  
7 </div>
```



Controller object

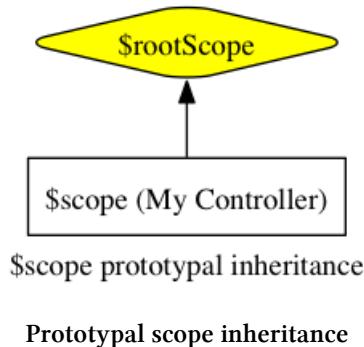
As we can see, the `$scope` object is how we'll pass along information from the model to the view. It is also how we will set up watch events, interact with other parts of the application, and how we will create application-specific logic.

Angular uses scopes to isolate functionality of the view, controllers, and directives (we'll cover these later in the book). This makes it very easy to write tests for a specific piece of functionality.

Scopes within scopes Every part of an AngularJS application has a parent scope (at the `ng-app` level, this is called the `$rootScope` as we've seen), regardless of the context that it is rendered within.



There is one exception in the case that a scope is created inside of a directive, this is called the *isolate* scope.



With that one exception, all scopes are created with prototypal inheritance, meaning that they have access to their parent scopes. If we are familiar with object-oriented programming, this behavior should look familiar. By default, for any property that AngularJS cannot find on a local scope, AngularJS will crawl up to the containing (parent) scope and look for the property or method there. If AngularJS can't find the property there, it will walk to that scope's parent and so on and so forth until it reaches the `$rootScope`. If it doesn't find it on the `$rootScope`, then it moves on and is unable to update the view.

To see this behavior in action, let's create a `ParentController` that contains the `user` object and a `ChildController` that wants to reference that object:

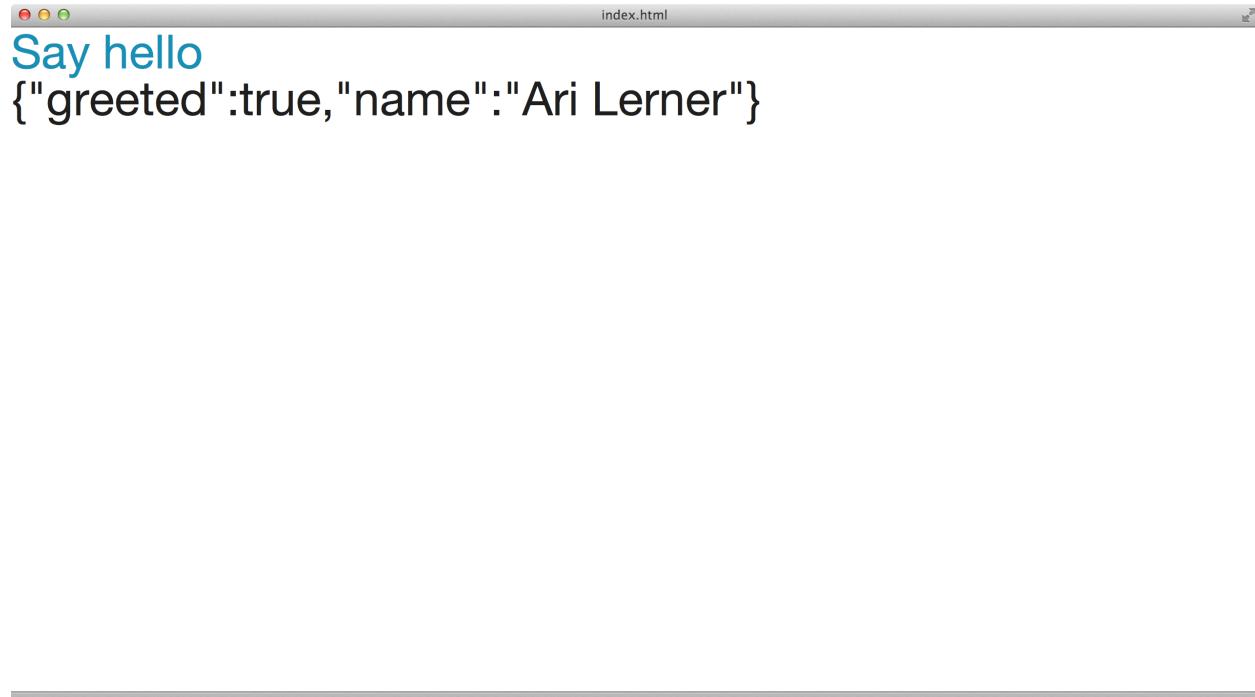
```

1 app.controller('ParentCtrl', function($scope) {
2   $scope.person = {greeted: false};
3 });
4
5 app.controller('ChildCtrl', function($scope) {
6   $scope.sayHello = function() {
7     $scope.person.name = "Ari Lerner";
8     $scope.person.greeted = true;
9   }
10 });
  
```

If we bind the `ChildController` under the `ParentController` in our view, then the *parent* of the `ChildController`'s `$scope` object will be the `ParentController`'s `$scope` object. Due to the prototypal behavior, we can then reference data on the `ParentController`'s containing `$scope` on the child scope.

For instance, we can reference the `person` object that is defined on the `ParentController` inside the DOM element of the `ChildController`.

```
1 <div ng-controller="ParentController">
2   <div ng-controller="ChildController">
3     <a ng-click="sayHello()">Say hello</a>
4   </div>
5   {{ person }}
6 </div>
```

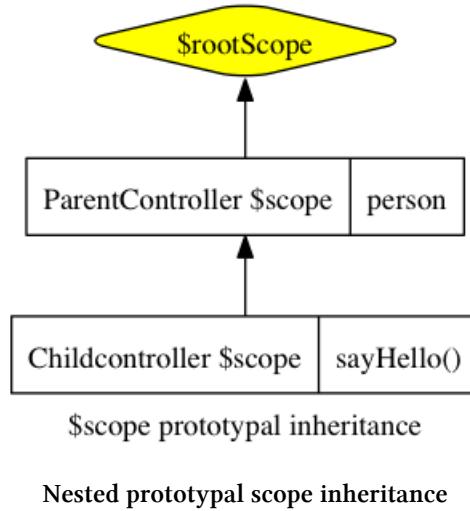


Nested controllers



This nested structure of controllers closely resembles the nested structure of the DOM itself.

As we can see, once we press the button, we can reference the `ParentController`'s `$scope.person` value inside the `ChildController` just as though `person` was defined in the `ChildController`'s `$scope` object.



It is a best practice to keep our controllers as slim as possible. It's bad practice to do any DOM interaction or data manipulating inside the controller.

As we previously mentioned, \$scopes have the ability to *watch* for changes of a particular property and react to the change.

Similar to how we can do the same for browser-side javascript with `addEventListener`, the \$scope can do the same for *AngularJS* variables.

Expressions

Expressions are used all over AngularJS apps, so it's important we get a good solid understanding of what expressions are and how they are used/evaluated by AngularJS.

We've seen examples of Angular expressions already. The `{{ }}` notation for showing a variable attached to a `$scope` is actually an expression: `{{ expression }}`. When setting up a `$watch`, we use an expression (or a function) that will be evaluated by Angular.

Expressions are similar to like the result of an `eval(javascript)` (roughly). They are processed by Angular and, therefore, have these important, distinct properties:

- All expressions are executed in the context of the scope and have access to local `$scope` variables.
- They do not throw errors if an expression results in a `TypeError` or a `ReferenceError`.
- They do not allow for any control flow functions (conditionals; e.g., `if/else`).
- They can accept a filter and/or filter chains

Expressions all operate on the containing scope within which they are called. This fact enables you to call variables bound to the containing scope inside of an expression, which, in turn, enables you to loop over variables (we'll see this with `ng-repeat`), call a function, or use variables for math expressions from the scope.



In our example above, we are setting a `watchExpression` on the 'full_name' attribute of the `$scope` object. You can think about this as though the `watchExpression` is saying:
`$scope.$watch($scope.full_name, function(newVal) {});`

Parsing an angular expression

Although your angular app will run parse for you automatically when running the `$digest` loop, sometimes it's useful to parse an angular expression manually.

Expressions are evaluated by the `$parse` service, a service that is baked into AngularJS core, so we can get access to the raw functions that are actually run to evaluate expressions using the `$parse` service.

To manually parse an expression, we can inject the `$parse` service into a controller and call the service to do the parsing for us. For instance, if we have an input box in our page that's bound to the `expr` variable, like so:

```

1 <div ng-controller="MyController">
2   <input ng-model="expr"
3     type="text"
4     placeholder="Enter an expression" />
5 </div>

```

In MyController, we can then set a \$watch and parse the expression of expr.

```

1 angular.module("myApp")
2 .controller('MyCtrl',
3 ['$scope', '$parse', function($scope, $parse) {
4   $scope.$watch('expr', function(newVal, oldVal, scope) {
5     if (newVal !== oldVal) {
6       // Let's set up our parseFun with the expression
7       var parseFun = $parse(newVal);
8       // Get the value of the parsed expression
9       var parsedValue = parseFun(scope);
10    }
11  });
12 }]);

```

Interpolating a string

Although it's uncommon to need to manually interpolate a string template in angular, we do have the ability to manually run the template compilation.

To run an interpolation on a string template, we'll need to inject the \$interpolate service in our object. In this example, we'll be injecting it into a controller:

```

1 angular.module('myApp')
2 .controller('MyCtrl',
3 ['$scope', '$interpolate',
4   function($scope, $interpolate) {
5     // We have access to both the $scope
6     // and the $interpolate services
7 }]);

```

The \$interpolate service takes up to three parameters, with only one required function.

- text (string) - the text with markup to interpolate

- mustHaveExpression (boolean) - if we set this to true, then the text will return null if there is no expression
- trustedContext (string) - the result of the interpolation context will send the result through the \$sce.getTrusted() method that provides strict contextual escaping



See `$sce` for more details about the last parameter

The `$interpolate` service will return an interpolation function that takes a context object against which the expressions will be evaluated against.

With this set up, we can now run an interpolation inside the controller. For instance, let's say we want to show *live editing* of the body of text in an email, we can run an interpolation when the text changes to show the given output.

```

1 <div id="emailEditor">
2   <input ng-model="to"
3     type="email"
4     placeholder="Recipient" />
5   <textarea ng-model="emailBody"></textarea>
6 </div>
7 <div id="emailPreview">
8   <pre>{{ previewText }}</pre>
9 </div>
```

In our controller, we'll set up a `$watch` to monitor changes on the email body and interpolate the `emailBody` into our `previewText` property.

```

1 angular.module('myApp')
2   .controller('MyCtrl',
3     ['$scope', '$interpolate',
4      function($scope, $interpolate) {
5        // Set up a watch
6        $scope.$watch('emailBody', function(body) {
7          if (body) {
8            var template = $interpolate(body);
9            $scope.previewText =
10              template({to: $scope.to});
11        }
12      });
13    }]);
```

If it's desirable to use different beginning and ending symbols in our text, we can modify them by configuring the `$interpolateProvider`.

To modify the beginning string, we can set the starting symbol with the `startSymbol()` method.

The `startSymbol()` takes a single argument:

- value (string) - the value to set the starting symbol

To modify the ending symbol, we can use the `endSymbol()` function. This function takes a single argument as well:

- value (string) - the value to set the end symbol

To modify the starting symbol, we can create a new module and inject the `$interpolateProvider` into the `config()` function.

We'll also create a `service` which we will cover in-depth in the `service` chapter.

```
1 angular.module('emailParser', [])
2   .config(['$interpolateProvider',
3     function($interpolateProvider) {
4       $interpolateProvider.startSymbol('__');
5       $interpolateProvider.endSymbol('__');
6     }])
7   .factory('EmailParser', ['$interpolate',
8     function($interpolate) {
9       // a service to handle parsing
10      return {
11        parse: function(text, context) {
12          var template = $interpolate(text);
13          return template(context);
14        }
15      }
16    }]);

```

With this module created, we can inject it into our app and run the email parser on the text on our email body:

```
1 angular.module('myApp', ['emailParser'])
2   .controller('MyCtrl',
3     ['$scope', 'EmailParser',
4      function($scope, EmailParser) {
5        // Set up a watch
6        $scope.$watch('emailBody', function(body) {
7          if (body) {
8            $scope.previewText =
9              EmailParser.parse(body, {
10                to: $scope.to
11              });
12          }
13        });
14      }]);
```

Now, instead of requiring the text to use the default syntax with the {{ }} symbols, we can define our symbols to use __ instead.

Filters

In AngularJS, a filter provides a way to format data to display to the user. Angular gives us several built-in filters as well as an easy way to create our own.

Filters are invoked in the HTML with the | (pipe) character in the template binding characters {{ }}. For instance, let's say we want to capitalize our string. We can either change all the characters in a string to be capitalized, or we can use a filter.

```
1 {{ name | uppercase }}
```

We can also use filters from within JavaScript by using the \$filter service. For instance, to use the uppercase JavaScript filter:

```
1 app.controller('DemoCtrl', ['$scope', '$filter',
2   function($scope, $filter) {
3
4     $scope.name = $filter('lowercase')('Ari');
5   }]);

```

To pass an argument to a filter in the html form, we pass it with a colon after the filter name (for multiple arguments, we can simply append a colon after each argument). For example, the number filter will allow us to limit the number of decimal places a number can show. To pass the argument 2, we'll append :2 to the number filter:

```
1 {{ 123.456789 | number:2 }} <!-- Displays: 123.45 -->
```

We can use multiple filters at the same time by using two or more pipes. We'll see such an example in a minute when we build a custom filter. Before we get to that, however, let's look at the built-in filters that come out-of-the-box with AngularJS.

currency

The currency filter formats a number as currency. In other words, 123 as currency will look like: {{ 123 | currency }}.

Currency has the option of a currency symbol or identifier to display the currency. The default currency option is that of the current locale; however, you can pass in a currency to display.

date

The date filter allows us to format a date based upon a requested format style. The date formatter provides us several built-in options. If no date format is passed, then it defaults to showing `mediumDate` (as you can see below).

Here are the built-in localizable formats:

```

1 {{ today | date:'medium' }}      <!-- Aug 09, 2013 12:09:02 PM -->
2 {{ today | date:'short' }}       <!-- 8/9/13 12:09 PM -->
3 {{ today | date:'fullDate' }}    <!-- Thursday, August 09, 2013 -->
4 {{ today | date:'longDate' }}    <!-- August 09, 2013 -->
5 {{ today | date:'mediumDate' }}   <!-- Aug 09, 2013 -->
6 {{ today | date:'shortDate' }}    <!-- 8/9/13 -->
7 {{ today | date:'mediumTime' }}   <!-- 12:09:02 PM -->
8 {{ today | date:'shortTime' }}    <!-- 12:09 PM -->
```

The date formatter also enables you to customize your date format to your own liking. The format options allow you to format the date with the different components of a date. These format options can be combined and chained together to create one single date format, as well:

Year formatting

```

1 Four digit year: {{ today | date:'yyyy' }} <!-- 2013 -->
2 Two digit padded year: {{ today | date:'yy' }} <!-- 13 -->
3 One digit year: {{ today | date:'y' }} <!-- 2013 -->
```

Month formatting

```

1 Month in year: {{ today | date:'MMMM' }} <!-- August -->
2 Short month in year: {{ today | date:'MMM' }} <!-- Aug -->
3 Padded month in year: {{ today | date:'MM' }} <!-- 08 -->
4 Month in year: {{ today | date:'M' }} <!-- 8 -->
```

Day formatting

```

1 Padded day in month: {{ today | date:'dd' }} <!-- 09 -->
2 Day in month: {{ today | date:'d' }} <!-- 9 -->
3 Day in week: {{ today | date:'EEEE' }} <!-- Thursday -->
4 Short day in week: {{ today | date:'EEE' }} <!-- Thu -->
```

Hour formatting

```

1 Padded hour in day: {{ today | date:'HH' }} <!-- 00 -->
2 Hour in day: {{ today | date:'H' }} <!-- 0 -->
3 Padded hour in am/pm: {{ today | date:'hh' }} <!-- 12 -->
4 Hour in am/pm: {{ today | date:'h' }} <!-- 12 -->
```

Minute formatting

```

1 Padded minute in hour: {{ today | date:'mm' }} <!-- 09 -->
2 Minute in hour: {{ today | date:'m' }} <!-- 9 -->
```

Second formatting

```

1 Padded second in minute: {{ today | date:'ss' }} <!-- 02 -->
2 Second in minute: {{ today | date:'s' }} <!-- 2 -->
3 Padded millisecond in second: {{ today | date:'sss' }} <!-- .995 -->
```

String formatting

```

1 am/pm character: {{ today | date:'a' }} <!-- AM -->
2 4 digit representation of timezone offset: {{ today | date:'Z' }} <!-- -0700 -->
```

Some examples of custom date formatting:

```

1 {{ today | date:'MMM d, y' }} <!-- Aug 09, 2013 -->
2 {{ today | date:'EEEE, d, M' }} <!-- Thursday, 9, 8 -->
3 {{ today | date:'hh:mm:ss.sss' }} <!-- 12:09:02.995 -->
```

filter

The `filter` filter selects a subset of items from an array of items and returns a new array. This filter is generally used as a way to *filter* out items for display. For instance, when using client-side searching, we can filter out items from an array immediately.

The `filter` method takes a string, object, or function that it will run to select or reject array elements. If the first parameter passed in is a:

string

It will accept all elements that match against the string. If we want all the elements that do **not** match the string, we can prepend the string with a `!`.

object

It will compare objects that have a property name that match like the simple substring match if only a string is passed in. If we want to match against *all* properties, we can use the \$ as the key.

function

It will run the function over each element of the array and the results that return as non-falsy will be in the new array.

For instance, selecting all of the words that have the letter e in them, we could run our filter like so:

```
1 {{ ['Ari', 'Lerner', 'Likes', 'To', 'Eat', 'Pizza'] | filter:'e' }}
2 <!-- ["Lerner", "Likes", "Eat"] -->
```

If we want to filter on objects, we can use the object filter notation as we discussed above. For instance, if we have an array of people objects with a list of their favorite foods, we could filter them like so:

```
1 {{ [
2   {
3     'name': 'Ari',
4     'City': 'San Francisco',
5     'favorite food': 'Pizza'
6   },
7   {
8     'name': 'Nate',
9     'City': 'San Francisco',
10    'favorite food': 'indian food'
11  }] | filter:{'favorite food': 'Pizza'} }}
12 <!-- [{"name": "Ari", "City": "San Francisco", "favorite food": "Pizza"}] -->
```

We can also filter based on a function that we define (in this example, on the containing \$scope object):

```
1 {{ ['Ari', 'likes', 'to', 'travel'] | filter:isCapitalized }}
2 <!-- ["Ari"] -->
```

The isCapitalized function that returns true if the first character is a capital letter and false if it is not is defined as:

```
1 $scope.isCapitalized =  
2   function(str) { return str[0] == str[0].toUpperCase(); }
```

You can also pass a second parameter into the filter method that will be used to determine if the expected value and the actual value should be considered a match.

If the second parameter passed in is:

true

It runs a strict comparison on the two using `angular.equals(expected, actual)`.

false

It will look for a case-insensitive substring match

function

It will run the function and accept an element if the result of the function is truthy.

json

The `json` filter will take a JSON, or JavaScript object, and turn it into a string. This transformation is very useful for debugging purposes:

```
1 {{ {'name': 'Ari', 'City': 'San Francisco'} | json }}  
2 <!--  
3 {  
4   "name": "Ari",  
5   "City": "San Francisco"  
6 }  
7 -->
```

limitTo

The `limitTo` filter creates a new array or string that contains only the specified number of elements, either taken from the beginning or end, depending on whether the value is positive or negative.

If the limit exceeds the value of the string, then the entire array or string will be returned.

For instance, we can take the first three letters of a string:

```
1 {{ San Francisco is very cloudy | limitTo:3 }} <!-- San -->
```

Or we can take the last 6 characters of a string:

```
1 {{ San Francisco is very cloudy | limitTo:-6 }} <!-- cloudy -->
```

We can do the same with an array. Here we'll return only the first element of the array:

```
1 {{ ['a', 'b', 'c', 'd', 'e', 'f'] | limitTo:1 }} <!-- ["a"] -->
```

lowercase

The `lowercase` filter simply lowercases the entire string.

```
1 {{ "San Francisco is very cloudy" | lowercase }} <!-- san francisco is very cloud\y -->
```

number

The `number` filter formats a number as text. It can take a second parameter (optional) that will format the number to the specified number of decimal places (rounded).

If a non-numeric character is given, it will return an empty string.

```
1 {{ 123456789 | number }} <!-- 1,234,567,890 -->
2 {{ 1.234567 | number:2 }} <!-- 1.23 -->
```

orderBy

The `orderBy` filter orders the specific array using an expression.

The `orderBy` function can take two parameters: The first one is required, while the second is optional.

The first parameter is the predicate used to determine the order of the sorted array.

If the first parameter passed in is a:

function

It will use the function as the getter function for the object.

string

It will parse the string and use the result as the key by which to order the elements of the array. We can pass either a + or a - to force the sort in ascending or descending order.

array

It will use the elements as predicates in the sort expression. It will use the first predicate for every element that is **not** strictly equal to the expression result.

The second parameter controls the sort order of the array (either reversed or not).

For instance, let's sort an array of objects by their name. Say we have an array of people, we can order the array of objects with the +name value:

```
1 {{ [{"  
2   "name": "Ari",  
3   "status": "awake"  
4 }, {"  
5   "name": "Q",  
6   "status": "sleeping"  
7 }, {"  
8   "name": "Nate",  
9   "status": "awake"  
10 }]} | orderBy: '+name' }}  
11 <!--  
12 [  
13 {"name": "Ari", "status": "awake"},  
14 {"name": "Nate", "status": "awake"},  
15 {"name": "Q", "status": "sleeping"}  
16 ]  
17 -->
```

We can also reverse-sort the object. For instance, reverse-sorting the previous object, we simply add the second parameter as true:

```

1  {{ [
2    { 'name': 'Ari',
3      'status': 'awake'
4    }, {
5      'name': 'Q',
6      'status': 'sleeping'
7    }, {
8      'name': 'Nate',
9      'status': 'awake'
10     }] | orderBy:'name':true }}
11 <!--
12   [
13     {"name": "Q", "status": "sleeping"},
14     {"name": "Nate", "status": "awake"},
15     {"name": "Ari", "status": "awake"}
16   ]
17 -->

```

uppercase

The uppercase filter simply uppercases the entire string:

```

1  {{ "San Francisco is very cloudy" | uppercase }} <!-- SAN FRANCISCO IS VERY CLOUD\
2  Y -->

```

Making our own filter

As we saw above, it's really easy to create our own custom filter. To create a filter, we put it under its own module. Let's create one together: a filter that capitalizes the first character of a string.

First, we need to create it in a module that we'll require in our app (this step is good practice):

```

1 angular.module('myApp.filters', [])
2 .filter('capitalize', function() {
3   return function(input) {}
4 });

```

Filters are just functions to which we pass input. In the function above, we simply take the input as the string on which we are calling the filter. We can do some error-checking inside the function:

```
1 angular.module('myApp.filters', [])
2 .filter('capitalize', function() {
3   return function(input) {
4     // input will be the string we pass in
5     if (input)
6       return input[0].toUpperCase() + input.slice(1);
7   }
8 });


```

Now, if we want to capitalize the first letter of a sentence, we can first lowercase the entire string and then capitalize the first letter with our filter:

```
1 <!-- Ginger loves dog bones -->
2 {{ 'ginger loves dog bones' | lowercase | capitalize }}
```

For an extended example of using filters make sure to check out the [full calendar application](#) we be building together.

Introduction to Directives

As web developer's, we're all familiar with HTML. Let's take a moment to review and synchronize our terminology around this most fundamental of web technologies.

HTML Document

An HTML document is a plain text document that contains structure and may be styled through css or manipulated via JavaScript.

HTML Node

An HTML node is an element or chunk of text nested inside another element. All elements are also nodes, however a text node is not an element.

HTML Element

An element is made up of an opening and closing tag.

HTML tag

An HTML tag is responsible for marking the begin and end of an element. A tag itself is declared using angle brackets.

An opening tag contains a name, which will become the name of the element. It can also contain attributes which decorate the element.

attributes

To provide additional information about an element, HTML elements can contain attributes. These attributes are always set in the opening tag. Attributes can be set in a key/value pair, like `key="value"` or they can be set as just a key.

Let's take a look at the `<a>` hyperlink tag, which is used to create a link from one page to another:

Some tags, like the hyperlink, tag have special attributes that act much like arguments to the tag. For example the `href` attribute of a link tag enable behavior of the link tag and also turn the text node in between the opening and closing tags blue by default on all browsers.

```
1 <a href="http://google.com">Click me to go to Google</a>
```

The `<a>` tag defines a link between another page on our site or off our site, depending on the contents of the `href` attribute, which defines the link's destination.

It is noticeably different than the following HTML element, the button:

```
1 <button type="submit">Click me</button>
```

The link tag is, by default underlined and blue while the button, by default looks like a click-able button in our browser.

The link tag knows that when provided an `href` attribute that points to `http://google.com`, it should change the URL in the address bar and load Google's home page when a user clicks on the link.

The button tag on the other hand is completely oblivious when provided an `href` attribute and does not perform the same behavior (it is ignored).

Thus, the ability for a link to change the URL in the address bar and bring you to a new page is part of a link's pre-programmed behavior, but not the button's pre-programmed behavior.

Finally, both tags perform the same behavior when provided a `title` attribute - they provide a tooltip to the user upon hover.

```
1 <a href="http://google.com" title="click me">Click me to go to Google</a>
2 <button type="submit" title="click me">Click me</button>
```

In summary, HTML elements have both style and behavior rendered by the web browser. This is one of the fundamental strengths of the web.

Each vendor, whether it be Google or Microsoft tries to adhere to the same HTML spec, therefore making programming for the web consistent across devices and operating systems.

Past versions of Internet Explorer have not complied with the common HTML spec., so we need to perform some tricks to get older versions of IE to work. See the [Internet Explorer](#) chapter for more details.

Recently, new HTML tags have begun to emerge. These are a part of the HTML5 spec. For example, the `video` tag, which specifies a video, a movie clip or streaming video:

```
1 <video href="/goofy-video.mp4"></video>
```

These new HTML5 tags work on on newer browsers (and are generally *not* supported by Internet Explorer version 8 and lower).

Directives: Custom HTML Elements and Attributes

Given what we know about HTML elements, directives are Angular's method of creating new HTML elements with their own custom functionality. For instance, we can create our own custom element that implements the video tag that works across all browsers:

```
1 <my-better-video my-href="/goofy-video.mp4">Can even take text</my-better-video>
```

Notice our custom element has a custom open and closing tag, `my-better-video`, and a custom attribute, `my-href`.

To make our tag more usable, we could just override the browser provided `video` tag, which means we could instead use: `{lang="html"}`

As we can see, directives can be combined with other directives and attributes. This is called composition.

To effectively understand how to compose a system from smaller parts, we must first understand the primitive pieces. That'll be the underlying goal of the next few of chapters. Let's get started.

Bootstrapped HTML

When the browser loads our HTML page along with Angular, we'll only one need snippets of code to boot our application, which we learned about in the introductory chapter.

In our HTML we'll need to markup the root of our app using the *built in* directive `ng-app`. It's meant to be used as an attribute, thus we could stick it anywhere, but choose the opening `<html>` tag, which is normative:

A built in directive is one that ship's out of the box with Angular. All built in directives are prefixed with the `ng` namespace. Do not prefix the name of your own directives with `ng` to avoid namespace collisions.

```
1 <html ng-root="myApp">
2   <!-- $rootScope of our application -->
3 </html>
```

Inside of our `<html>` element, we can now use any of the built in or custom directives we desire. Furthermore, all directives we use within this root element will have access to `$rootScope` via prototypical inheritance in our JavaScript code *if the method of the directive has access to scope*, meaning scope has been linked to the DOM, which is done late in the [directive lifecycle](#).

Because the life cycle of a directive is sufficiently complex, it warrants its own section, where we'll also discuss which methods within a directive have access to scope, and how scope is shared between one directive and the next. See the [directives explained](#) chapter for more information.

Our First Directive

The quickest way to get our feet wet is to just dive right in. Let's go ahead and create a very basic custom directive.

Consider the following HTML element, which we'll define in a moment:

```
1 <my-directive></my-directive>
```

Provided we've created an HTML document and included Angular as well as the `ng-app` directive in the DOM to mark the root of our app, when Angular *compiles* our HTML it will *invoke* this directive.



We'll learn more about the *compile* stage of the directive lifecycle in [understanding compile](#).



Invoking a directive means to run the associated JavaScript that sits behind our directive, which is defined using a [directive definition](#).

The `myDirective` directive definition looks like:

```
1 angular.module('myApp', [])
2   .directive('myDirective', function() {
3     return {
4       restrict: 'E',
5       template: '<a href="http://google.com">Click me to go to Google</a>'
6     }
7   });

```



The above javascript is called a *directive definition*. We'll see all the options for defining a directive in [directive definition](#).



Click me to go to Google

Simple directive in action

With the `.directive()` method provided by the angular module API, we can register new directives by providing a name as a string and function. The name of the directive should always be camelCased and the function we provide should return an object.



Camelcasing words is the practice of writing compound words or phrases such that each word, except the first word, begins with a capital letter and the phrase becomes a single word. For instance: `bumpy roads` in camelcase notation would be `bumpyRoads`.



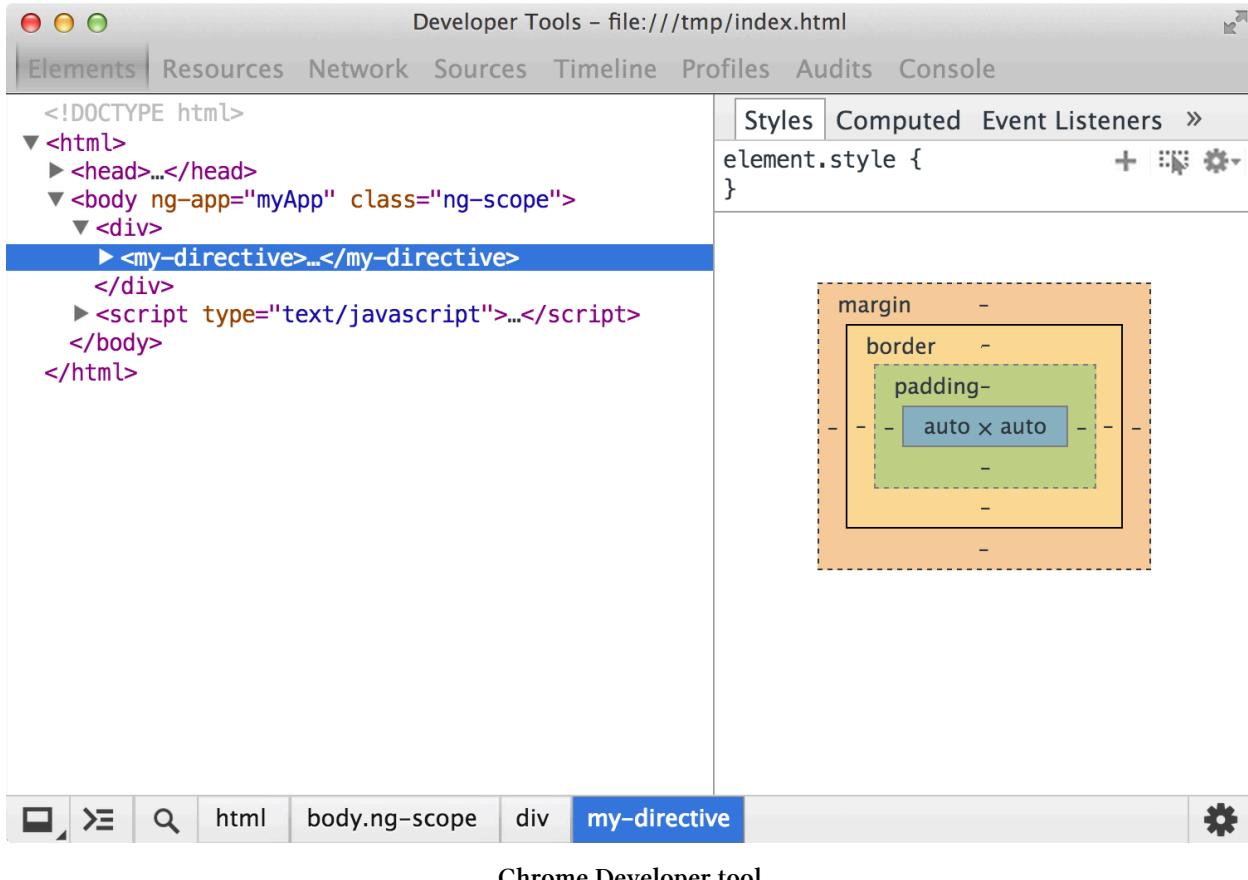
In our case, the directive is declared in HTML via `my-directive`, but the directive definition is `myDirective`.

The object that we return from the `.directive()` method is made up of methods and properties that we use to define and configure our directive.

In attempt to master the simplest directive possible, we've only defined our directive with two options: `restrict` and `template`.

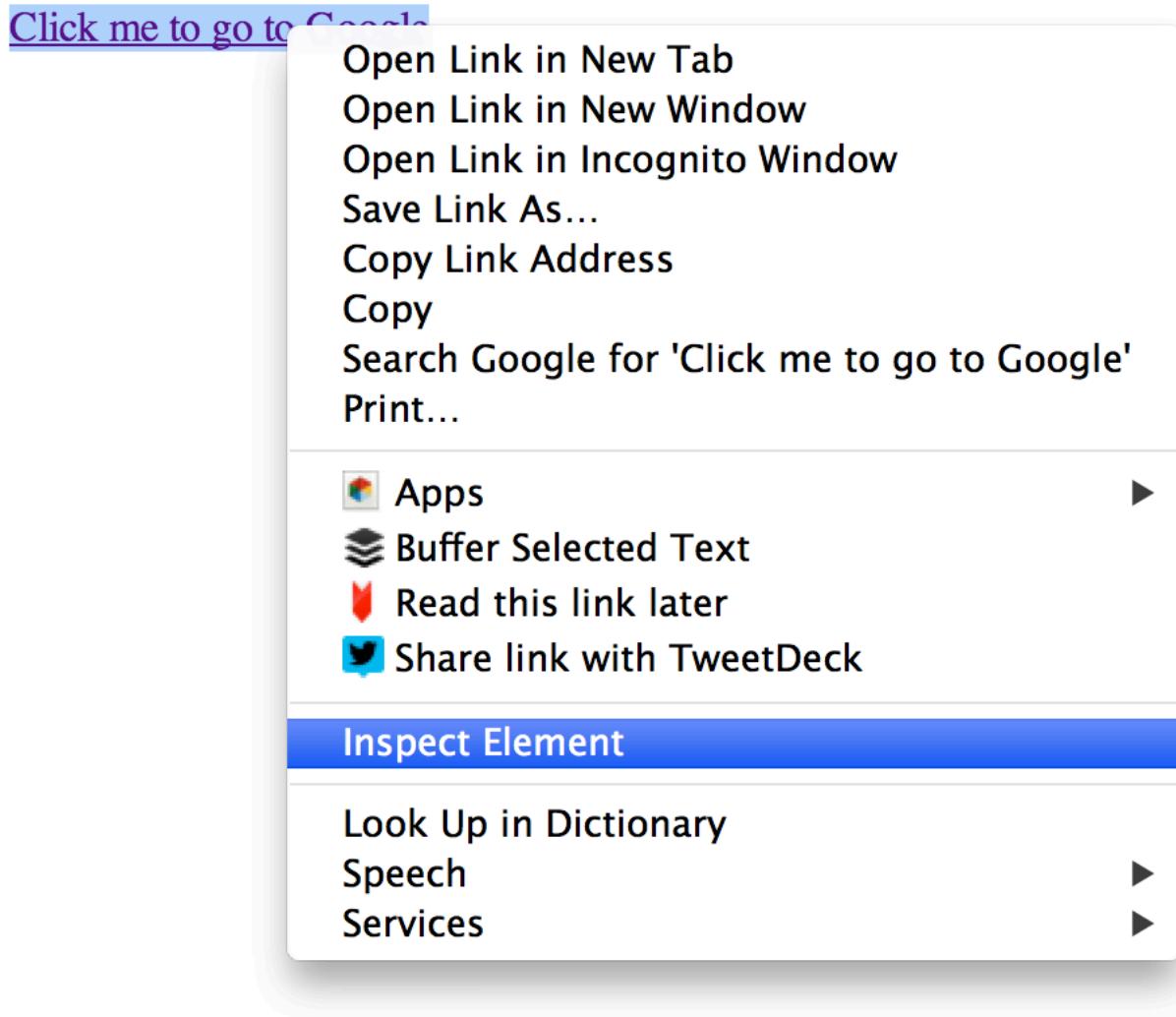
In [directives explained](#) we'll cover all the available methods and properties we can use when defining our own directive's, but for the moment, let's check out the input HTML as compared to the output HTML by using google chrome and it's developer tools.

First, open up your HTML document using Chrome. You'll see a blue link that says click here. Take a look at the source code by going to View > Developer > View Source. You should see the following picture:



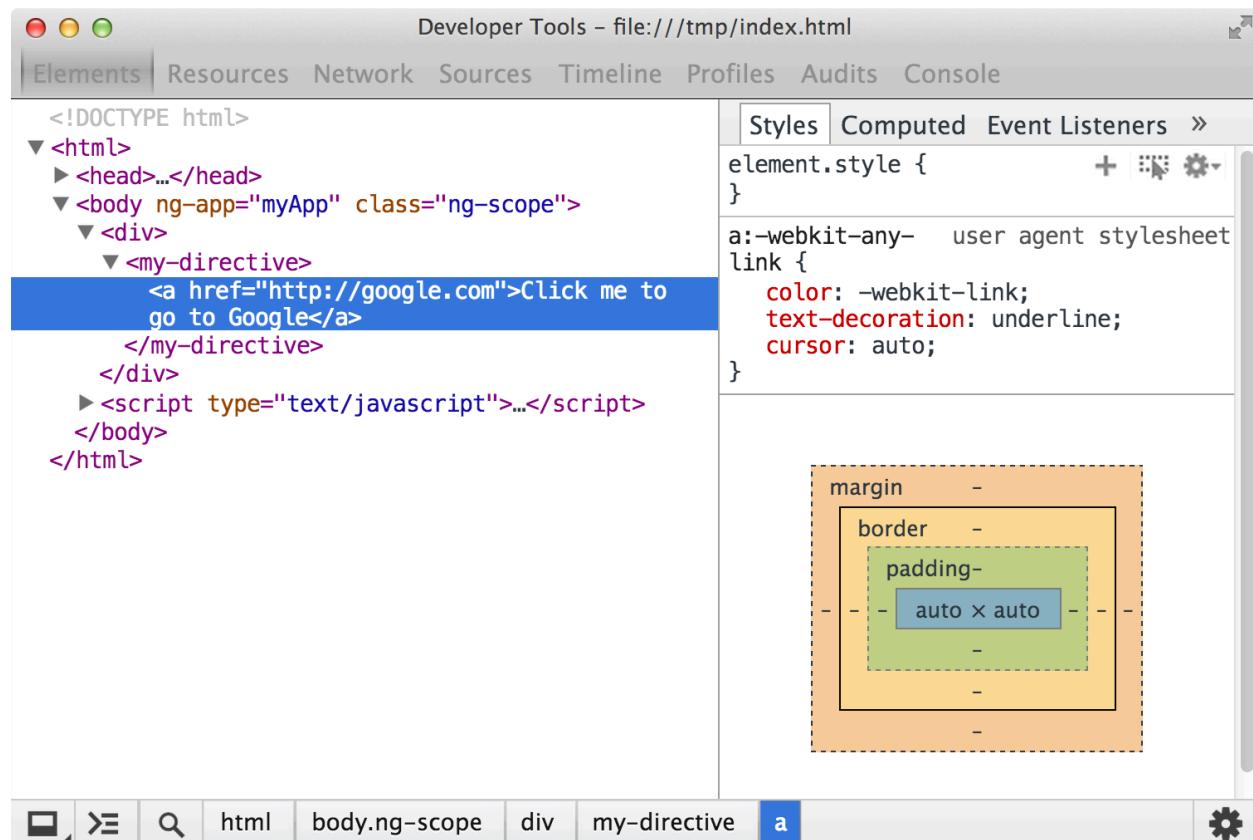
Notice it's no different than the source code you entered into your text editor. However, notice there is no link tag output to the screen yet. Clearly there's supposed to be a link that says "Click here". What's going on?

To investigate, right click on the link and in the drop down menu provided by Chrome, left click on Inspect Element:



Inspecting element

This will open up Chrome Developer tools and provide you with the generated source, which Angular provides to Chrome after the page is loaded and after Angular has invoked our directive's definition. Let's take a look:



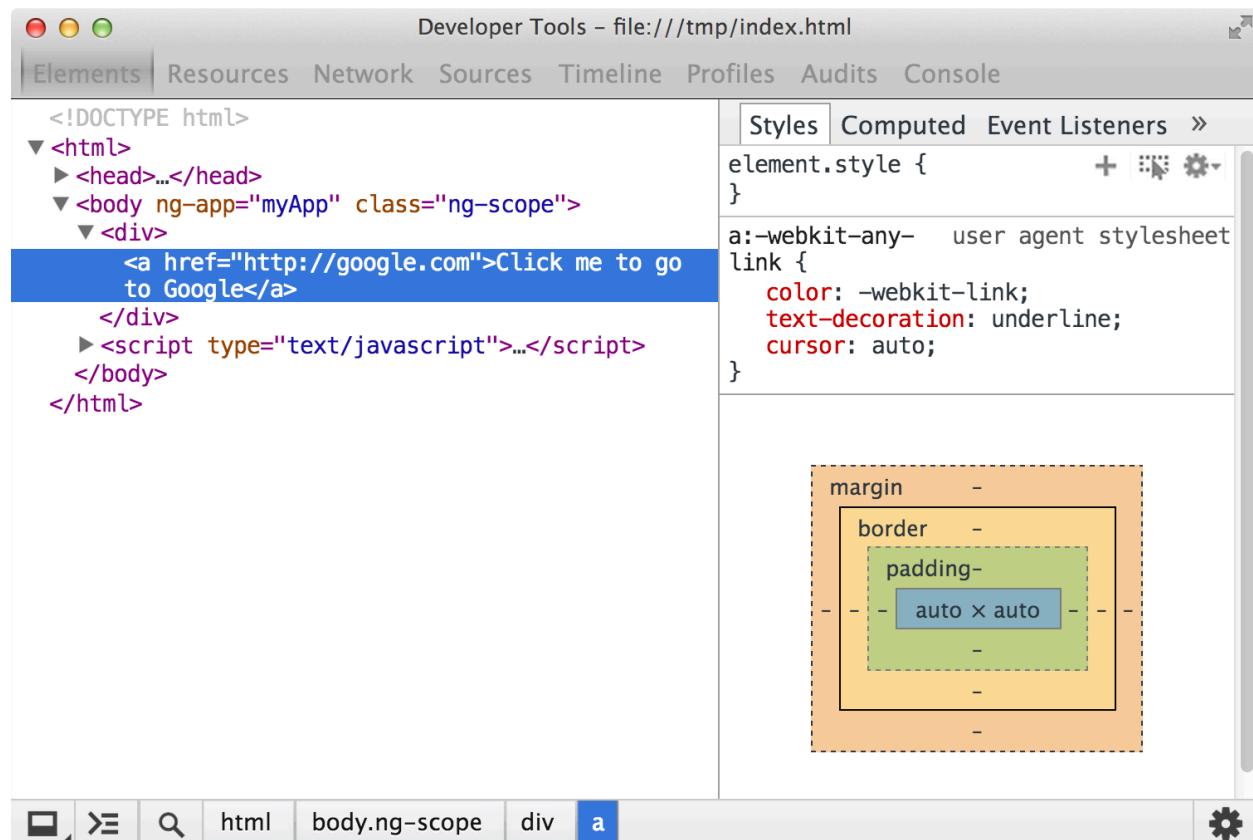
Unwrapping the directive

By default, Angular nests the HTML provided by our template string inside of our custom HTML tag `<my-directive>` in the generated source code.

Adding one more option to our directive definition, we can remove our custom element (`<my-directive>`) from the generated DOM completely and output only the link we're providing to the template option. To do this, set the `replace` option to `true`:

```
1 angular.module('myApp', [])
2 .directive('myDirective', function() {
3   return {
4     restrict: 'E',
5     replace: true,
6     template: '<a href="http://google.com">Click me to go to Google</a>'
7   }
8 })
```

Look at the generated source again. We can see that we no longer have the original call to the directive, but only the source that we set as the template. This `replace` method replaces the custom element, instead of wrapping it in our directive call.



Replacing existing element

From now on, we'll refer to these custom elements we create (using the `.directive()` method) as directives, because in fact, we do not need to make a new custom element to *declare* our directive.



Declaring a directive is the act of placing within our HTML as an element, attribute, class or comment.

The following are valid formats for declaring the directive we built above:

```
1 <my-directive></my-directive>
2 <div my-directive></div>
3 <div class="my-directive"></div>
4 <!-- directive: my-directive -->
```

In order to allow for our directive to be invoked by angular, we'll need to change the `restrict` option inside our directive definition. This tells Angular which declaration format(s) to look for when compiling our HTML. We can specify one or many formats.

For example, in the directive we're building we can specify that we want our directive to be invoked if it is an element(E) or an attribute(A) or a class(C) or a comment(M) or any of the four(letters EACM):

```

1 angular.module('myApp', [])
2 .directive('myDirective', function() {
3   return {
4     restrict: 'EAC',
5     replace: true,
6     template: '<a href="http://google.com">Click me to go to Google</a>'
7   };
8 });

```

Regardless of how many ways we *can* declare a directive, we'll stick to the most cross-browser compliant way by using an attribute:

```
1 <div my-directive></div>
```

And to make that clear to rest of our team, we'll set the `restrict` option to the letter `A`(for attribute):

```
1 restrict: 'A'
```

When following this convention however, be aware of browser built in styles and make a decision about whether to wrap or replace your directive's template.

A note on Internet Explorer

If you've got a copy of Internet Explorer handy, try opening [this live example on jsbin¹⁷](#). You'll notice that despite declaring your directive twice, only one link is showing up.

Technically, we can fix this by declaring new tags in the head of our document(see [Angular w/ IE](#)) but doing so can cause us headaches in the future if we forget to do so.

Thus, a good rule of thumb to follow is to always declare our directive as an attribute(as we've done). It'll save us hassle later.

A noteworthy exception is when extending built in HTML tags, for example Angular overrides `<a>`, `<form>` and `<input>`. Such cases don't cause browser compatibility issues because these tags already have browser support.

Expressions

Given a directive can(and usually should) be invoked as an attribute, we're inclined to ask about the value passed to that attribute:

¹⁷<http://jsbin.com/IJAzUJE/1/edit>

```
1 <h1 ng-init="greeting = 'Hello World'">The greeting is: {{ greeting }}</h1>
```

Live Example¹⁸

Notice that we've passed the *expression* `greeting = Hello World` to the built in directive `ng-init`. Inside the expression, we've set a property named `greeting` to the value `Hello World`. Then we're once again evaluating the expression `greeting` inside brackets `{{ greeting }}`.

In both cases we're evaluating a normal JavaScript expression on the current *scope* which is either the `$rootScope`, instantiated when angular *invokes* `ng-app` during application boot, or a child object often simply referred as `scope` or `$scope` or sometimes even context.

Declaring Our Directive with an Expression

Given we now know that we can declare a directive with or without an expression, let's revisit the valid ways of declaring an expression:

```
1 <my-directive="someExpression"></my-directive>
2 <div my-directive="someExpression"></div>
3 <div class="my-directive:someExpression"></div>
4 <!-- directive: my-directive someExpression -->
```

A reasonable question at this point is what environment the expression given to a directive runs within. The answer lies in familiarizing ourselves a bit with the elusive but extremely important concept of *current scope*, provided by the controller hierarchy of the surrounding DOM.

Current Scope Introduction

Let's quickly familiarize ourselves with scope provided by the DOM via the built in directive `ng-controller` which exists for the purpose of creating a new child scope in the DOM (and therefore explains the concept well):

```
1 <p>We can access: {{ rootProperty }}</p>
2 <div ng-controller="ParentCtrl">
3   <p>We can access: {{ rootProperty }} and {{ parentProperty }}</p>
4   <div ng-controller="ChildCtrl">
5     <p>
6       We can access:
7       {{ rootProperty }} and
8       {{ parentProperty }} and
9       {{ childProperty }}</p>
```

¹⁸<http://jsbin.com/IdUYExO/2/edit>

```

10    </p>
11    <p>{{ fullSentenceFromChild }}</p>
12  </div>
13</div>

1 angular.module('myApp', [])
2 .run(function($rootScope) {
3   // use .run to access $rootScope
4   $rootScope.rootProperty = 'root scope';
5 })
6 .controller('ParentCtrl', function($scope) {
7   // use .controller to access properties inside `ng-controller`
8   // in the DOM omit $scope, it is inferred based on the current controller
9   $scope.parentProperty = 'parent scope';
10 })
11 .controller('ChildCtrl', function($scope) {
12   $scope.childProperty = 'child scope';
13   // just like in the DOM, we can access any of the properties in the
14   // prototype chain directly from the current $scope
15   $scope.fullSentenceFromChild = 'Same $scope: We can access: ' +
16                           $scope.rootProperty + ' and ' +
17                           $scope.parentProperty + ' and ' +
18                           $scope.childProperty
19 });

```

Live example, with colored scopes for learning purposes¹⁹

More detailed information on ng-controller itself is available in [ng-controller](#) section of the [built in directives](#) chapter.

Be aware that there are other built in directives, like [ng-include](#) and [ng-view](#) that also create a new child scope, meaning they behave similar to ng-controller when invoked. We can even create a new child scope when building a custom directive of our own. For the nuts and bolts on contextual scope provided by the surrounding DOM, see [contextual scope explained](#).

Passing Data Into a Directive

Let's recall our directive definition:

¹⁹<http://jsbin.com/URuyoG/1/edit>

```
1 angular.module('myApp', [])
2 .directive('myDirective', function() {
3   return {
4     restrict: 'A',
5     replace: true,
6     template: '<a href="http://google.com">Click me to go to Google</a>'
7   }
8 })
```

Notice that in our template we are hard coding the url and the text of our link:

```
1 template: '<a href="http://google.com">Click me to go to Google</a>'
```

With Angular, we aren't limited to hard coding strings in the template of our directive.

We can provide a nicer experience for others using our directive if we specify the URL and link text without messing with the internal guts of our directive. Our goal here is to pay attention to the public interface of our directive, just as we would in any programming language.

In essence, we'd like to turn the above template string into one that takes two variables, one for the URL and one for the link's text:

```
1 template: '<a href="{{myUrl}}>{{myLinkText}}</a>'
```

Looking at our main HTML document, we can declare our directive with attributes that will become the properties `myUrl` and `myLinkText` set on the inner scope of our directive:

```
1 <div my-directive
2   my-url="http://google.com"
3   my-link-text="Click me to go to Google">
4 </div>
```

Reload the page and notice that the div where we declared our directive has been replaced by its template, however the link's href is empty, and there's no text inside the bracket's.

The screenshot shows the Google Chrome Developer Tools interface. The left pane displays the DOM tree with the following code:

```
<!DOCTYPE html>
<html>
  <head>...</head>
  <body ng-app="myApp" class="ng-scope">
    <div>
      <a href my-directive my-url="http://google.com" my-link-text="Click me" class="ng-binding"></a>
    </div>
    <script type="text/javascript">...</script>
  </body>
</html>
```

The right pane shows the Styles tab with the following CSS rules:

```
element.style {
}
a:-webkit-any-link {
  color: -webkit-link;
  text-decoration: underline;
  cursor: auto;
}
```

A large, semi-transparent diagram of the CSS box model is overlaid on the right side of the styles panel. The diagram illustrates the structure of a box with nested padding, border, and margin layers.

The bottom navigation bar shows the selected element: **a.ng-binding**.

Updating template

To set properties on the inner scope of our directive, we have a few options. The simplest option is to simply use the existing scope currently provided by the controller(`ng-controller`) we're nested inside of.

However, sharing state leaves us vulnerable. If that controller is removed or if a property with the name `myUrl` is later defined on its scope, we'll be forced to change our code, which is costly and frustrating.

To overcome this common issue, Angular provides the ability to create a new child scope(which we talked about earlier) or create an *isolate scope*.

In contrast to inherited scope(child scope) discussed earlier(in [current scope introduction](#)), an **isolate scope*** is **completely** separate from the current scope of the DOM. In order to set properties on this fresh object, we'll need explicitly pass data in via attributes, similar to the way we pass arguments into a method in JavaScript or Ruby.

When we set the scope of our directive to a clean object with its own properties

```
scope: { someProperty: "needs to be set" }
```

we're creating what is referred to as an *isolate scope*. Essentially, this means the directive gets its own \$scope object which we can only use inside other methods of the directive or inside the directives template string:

```

1 template: '<div>we have access to {{ someProperty }}</div>',
2 controller: function($scope) {
3   // a directive can have its own controller, in which case we can do
4   $scope.someProperty === "needs to be set" //=> ERROR!!!
5 }
```

ERROR?

Up until now we've left one minor detail out. Inside the scope object we can't actually set someProperty like we've done above.

```

1 scope: {
2   someProperty: "needs to be set" // this won't work!, source of error
3 }
```

Instead, the value is set in the DOM, via an attribute, which we mentioned acts similar to how passing an argument to a function works:

```

1 <div my-directive
2   some-property="copied to someProperty via @ binding strategy"></div>
```

Now, inside of our fresh scope object, we set the value of someProperty to the binding strategy, @, which tells angular to copy the value provided by the attribute some-property in the DOM to the value of someProperty on our fresh scope object:

```

1 scope: {
2   someProperty: '@'
3 }
```

Note that by default someProperty maps to the DOM attribute some-property. If we wanted to explicitly specify the name of the attribute we wanted to bind to, we could do so:

```

1 scope: {
2   someProperty: '@someAttr'
3 }
```

In this case the HTML attribute would be named some-attr instead of some-property:

```

1 <div my-directive
2   some-attr="copied to someProperty via @ binding strategy"></div>

```

Now, when we try to access `someProperty` inside our directive, for example in the directive's template or controller (like we did above), we'll be given value copied from the DOM attribute:

```

1 template: '<div>we have access to {{ someProperty }}</div>',
2 controller: function($scope) {
3   // a directive can have its own controller, in which case we can do
4   $scope.someProperty === "copied to someProperty via @ binding strategy" //=> true
5 }
6

```

Let's return to our main example with our new found knowledge in hand.

To copy data from the DOM into the isolate scope of our directive use attributes:

```

1 <div my-directive
2   my-url="http://google.com"
3   my-link-text="Click me to go to Google"></div>

```

```

1 angular.module('myApp', [])
2 .directive('myDirective', function() {
3   return {
4     restrict: 'A',
5     replace: true,
6     scope: {
7       myUrl: '@',      // binding strategy
8       myLinkText: '@' // binding strategy
9     },
10    template: '<a href="{{myUrl}}>{{myLinkText}}</a>'
11  }
12 })

```

Live Example²⁰

The default convention (used above) is for the attribute name and the property name to be the same (except the property name is camel-cased).

Because the name of the property on our scope is often times private, it is possible (although unusual) to specify the name of the attribute we want to link our internal property to:

²⁰<http://jsbin.com/eloKoDI/1/edit>

```

1 scope: {
2   myUrl: '@someAttr',
3   myLinkText: '@'
4 }

```

The above code says:

```

1 set the value of our directive's private $scope.myUrl property to the value provi\
2 ded by the attribute `some-attr`. That value may be hard coded or it may be the r\
3 esult of evaluating an {{ expression }} within the current scope, e.g., some-attr\
4 ="{{ expression }}"

```

Now in our html, we would use `some-attr` instead of `my-url`:

```

1 <div my-directive
2   some-attr="http://google.com"
3   my-link-text="Click me to go to Google"></div>

```

Furthermore, we could evaluate an expression within the scope of the DOM and pass the result to our directive, where it would eventually be set as the value of our bound property:

```

1 <div my-directive some-attr="{{ 'http://' + 'google.com' }}"></div>

```

Taking this example one step further let's see what happens if we create a text field and bind the value entered in the text field to a property on the `$scope` of our isolate directive:

(Note the use of the built in directive `ng-model` on the input tag)

```

1 <input type="text" ng-model="myUrl" />
2 <div my-directive
3   some-attr="{{ myUrl }}"
4   my-link-text="Click me to go to Google"></div>

```

Interestingly, this just works. However, if we move our text field *inside* the scope of our directive and try to make a binding in the other direction it doesn't work:

```

1 <div my-directive some-attr="{{ myUrl }}"
2   my-link-text="Click me to go to Google"></div>

```

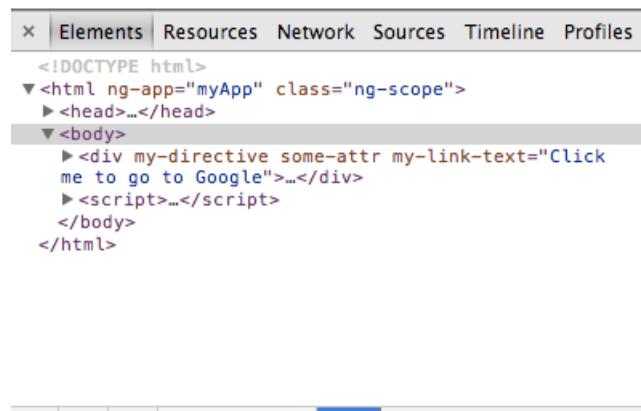
```

1 template: '\
2   <div>\
3     <input type="text" ng-model="myUrl" /> \
4     <a href="{{myUrl}}">{{myLinkText}}</a> \
5   </div> \
6 '

```

Observing the href within chrome dev tools hints we're not binding our internal \$scope property

<http://google.com> [Click me to go to Google](#)



myUrl to the external attribute some-attr in the opposite direction(it the value was copied onto our isolate scope via an attribute value, so shouldn't it also set that same attribute's value?)

This behavior occurs because the built in directive ng-model is setup with a 2 way binding between its own isolate scope and the scope of the DOM(provided by a controller).

Let's imitate that setup in an attempt to make our example work. Our goal is to understand two way bindings and the behavior ng-model in the process.

Two way bindings are perhaps the most important feature in Angular that we can't get with say Backbone JS out of the box. To understand the magic however, we need to implement it on our own. Luckily, we're only a few keystrokes away.

Let's finish our example by creating a two way data binding between our isolate \$scope and the isolate \$scope inside ng-model by binding our internal \$scope.myUrl property with a property named theirUrl on the \$scope of the current controller(or \$rootScope) using scope lookup in the DOM.

In the process, let's also add an additional text field so that both sides of the binding have their own text field. This will make it easy to see how scope is being linked via prototype inheritance in DOM:

```

1 <label>Their URL field:</label>
2 <input type="text" ng-model="theirUrl">
3 <div my-directive
4     some-attr="theirUrl"
5     my-link-text="Click me to go to Google"></div>

```

```

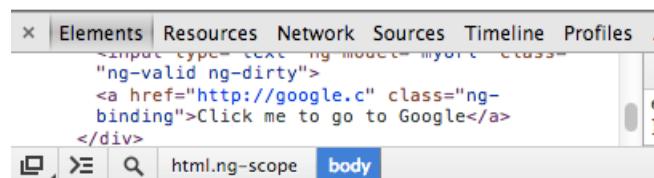
1 angular.module('myApp', [])
2 .directive('myDirective', function() {
3     return {
4         restrict: 'A',
5         replace: true,
6         scope: {
7             myUrl: '=someAttr', // MODIFIED
8             myLinkText: '@'
9         },
10        template: '\
11            <div>\
12                <label>My Url Field:</label> \
13                <input type="text" ng-model="myUrl" /> \
14                <a href="{{myUrl}}>{{myLinkText}}</a> \
15            </div> \
16        '
17    }
18 })

```

Live Example²¹

In chrome developer tools inspect the value of the href while typing into either text field. Awesome:

Their URL field: My Url Field: [Click me to go to Google](http://google.c)



The only change we really made besides adding our original text field back to the main HTML document is the use of the = binding strategy instead of @. (See [binding strategies explained](#)).

²¹<http://jsbin.com/IteNita/1/edit>

In summary, this example explains the magic of one of the fundamental selling points of Angular, two way data binding.

When working with built in directives it's useful to know what's going on so you can take into account their behavior when connecting them with your own directive's.

In the next chapter we'll take a closer look at more of the [built in directives](#) provided with Angular so that we're familiar with how to use them and why they exist.

Once we've been introduced to the built in directives, we'll be ready to dive into all the advanced options available when building a directive in [directives explained](#).

Finally we'll [build our own advanced directive](#) and then learn how to think about [architecture in our application](#) a subject where custom and built directives play an important role.

Built In Directives

Angular provides a suite of built in directives. Some directives override built in HTML elements, such as the `<form>` and `<a>` tags. When we use like this tag in our HTML, it may not be immediately obvious that we're in fact using a directive. For example, the `<form>` tag is augmented with a great deal of functionality under the hood such as validation behavior that we normally don't get with a standard HTML form.

Other built in directives are clearly visible via their `ng-` namespace prefix. For example, `ng-href`, which we'll cover below, prevents a link from becoming active until the expression provided to `ng-href="someExpression"` has been evaluated and returns a value.

Lastly, some built in directives do not have an HTML counter part, such as the `ng-controller` directive, which can be used as an attribute on any tag but is most often found on an element that has many children, where the child elements should have share the same scope.

Note that all directives prefixed with the `ng` namespace are part of the built in library of directives that ship with Angular. For this reason, never prefix directives you make with this namespace.

Basic ng attribute directives

Our first set of directives have similarly named standard HTML tags and are easy to remember because they simply add the `ng` prefix:

- `ng-href`
- `ng-src`
- `ng-disabled`
- `ng-checked`
- `ng-readonly`
- `ng-selected`
- `ng-class`
- `ng-style`

Boolean Attributes

The following Angular directives help make working with HTML boolean attributes easier.

As defined by the [HTML specification²²](#), a boolean attribute is an attribute that represents a true value. When the attribute is present, then the attributes value is assumed to be true(regardless of it's actual value). If it absent, the attribute is assumed to be false.

When working with dynamic data via data bindings in angular, we cannot simply set the value of the attribute to true or false, because by definition of the spec, the attribute is false if it *is not present*. Thus angular provides an ng prefixed version of these attributes that will evaluate the expression provided insert or remove the corresponding boolean attribute on the decorated element.

ng-disabled

Use ng-disabled to bind the disabled attribute to form input fields:

- <input> (text, checkbox, radio, number, url, email, submit)
- <textarea>
- <select>
- <button>

When writing normal HTML input fields, the presence of the disabled attribute on an input field makes the field disabled. To bind the presence(or not) of this attribute, use ng-disabled.

For example, let's disable the following button until the user enters text into the text field:

```
1 <input type="text" ng-model="someProperty" placeholder="Type to Enable">
2 <button ng-model="button" ng-disabled="!someProperty">A Button</button>
```

In the next example, we'll disable the textfield for 1 second until the isDisabled property becomes true inside the \$timeout:

```
1 <textarea ng-disabled="isDisabled">Wait 1 second</textarea>

1 angular.module('myApp', [])
2 .run(function($rootScope, $timeout) {
3   $rootScope.isDisabled = true;
4   $timeout(function() {
5     $rootScope.isDisabled = false;
6   }, 1000);
7 });


```

Both Live Examples²³

²²<http://www.w3.org/html/wg/drafts/html/master/infrastructure.html#boolean-attribute>

²³<http://jsbin.com/iHiYItu/1/edit>

ng-readonly

Similar to the other boolean attributes, the HTML spec only looks at the presence of the attribute `readonly`, not its value.

To allow angular the ability to bind to an expression that returns truthy or falsy and in turn output(or not) the `readonly` attribute use `ng-readonly`:

```
1 Type here to make sibling readonly: <input type="text" ng-model="someProperty"><b>
2 r/>
3 <input type="text" ng-readonly="someProperty" value="Some text here"/>
```

[Live Example²⁴](#)

ng-checked

The standard `checked` HTML attribute is a boolean attribute, and as such, is not required to take a value. In order for Angular to bind the presence of the `checked` attribute to the value of an expression, use `ng-checked`.

In the following example, we set the value of `someProperty` to true using the `ng-init` directive. Binding the value of `someProperty` to `ng-checked` then tells angular to output the standard HTML `checked` attribute, which will check the box by default.

```
1 <label>someProperty = {{someProperty}}</label>
2 <input type="checkbox"
3   ng-checked="someProperty"
4   ng-init="someProperty = true"
5   ng-model="someProperty">
```

In this example, we do the opposite:

```
1 <label>anotherProperty = {{anotherProperty}}</label>
2 <input type="checkbox"
3   ng-checked="anotherProperty"
4   ng-init="anotherProperty = false"
5   ng-model="anotherProperty">
```

Note that we also used `ng-model` to bind the value of `someProperty` and `anotherProperty` inside their respective label tags, for demonstration purposes.

[Live Example²⁵](#)

ng-selected

Use `ng-selected` to bind the presence(or not) of the `selected` attribute to the option tag:

²⁴<http://jsbin.com/etIviKI/1/edit>

²⁵<http://jsbin.com/IXOLIRA/1/edit>

```

1 <label>Select Two Fish:</label>
2 <input type="checkbox"
3     ng-model="isTwoFish"><br/>
4 <select>
5   <option>One Fish</option>
6   <option ng-selected="isTwoFish">Two Fish</option>
7 </select>
```

[Live Example²⁶](#)

Boolean like attributes

While not technically HTML boolean attributes the `ng-href` and `ng-src` act in a similar manner and are therefore defined along side the `ng` boolean attributes within the angular source code and presented here for the same reason.

Both `ng-href` and `ng-src` are so likely to help improve refactoring and prevent errors when changing code later in a project that it is recommended to use them in place of `href` and `src`, respectively.

ng-href

When dynamically creating a URL from a property on the current scope always use `ng-href` instead of `href`.

The issue here is that the user is able to click a link built with `href` before interpolation takes place, which would bring them to the wrong page(often a 404).

On the other hand, by using `ng-href`, Angular waits for the interpolation to take place, in our example after 2 seconds, and then activates the link's behavior:

```

1 <!-- Always use ng-href when href includes an {{ expression }} -->
2 <a ng-href="{{myHref}}>I'm feeling lucky, when I load</a>
3
4 <!-- href may not load before user clicks -->
5 <a href="{{myHref}}>I'm feeling 404</a>
```

Delay the interpolation of the string value for 2 seconds to see this in action:

²⁶<http://jsbin.com/oQazOQE/2/edit>

```

1 angular.module('myApp', [])
2 .run(function($rootScope, $timeout) {
3   $timeout(function() {
4     $rootScope.myHref = 'http://google.com';
5   }, 2000);
6 });

```

[Live Example²⁷](#)

ng-src

Angular will tell the browser to NOT fetch the image via the given URL until all expressions provided to ng-src have been interpolated:

```

1 <h1>Wrong Way</h1>
2 
3
4 <h1>Right way</h1>
5 

```

```

1 angular.module('myApp', [])
2 .run(function($rootScope, $timeout) {
3   $timeout(function() {
4     $rootScope.imgSrc = 'https://www.google.com/images/srpr/logo11w.png';
5   }, 2000);
6 });

```

[Live Example²⁸](#)

When viewing the live example, check out the network panel within chrome dev tools. You'll notice that one request is red meaning there was an error. That's from using src instead of ng-src under 'Wrong Way'.

Directives with child scope

The following directives create a child scope that prototypically inherit from their parent. This provides a layer of separation meant for storing methods and model objects that work together to achieve a common goal.

²⁷<http://jsbin.com/IgInopi/1/edit>

²⁸<http://jsbin.com/egucIqU/1/edit>

`ng-app` and `ng-controller` are special directives in that they modify the scope of directives nested inside of them.

`ng-app` creates the `$rootScope` of an angular application while `ng-controller` creates a child scope that prototypically inherits from either `$rootScope` or another `ng-controller`'s `$scope`.

ng-app

Placing `ng-app` on any DOM element will mark that element as the beginning of the `$rootScope`.

`$rootScope` is the beginning of the scope chain and all directives nested under the `ng-app` in your HTML inherit from it.

In your JavaScript code you can access the `$rootScope` via the `run` method:

```
1 <html ng-app="myApp">
2   <body>
3     {{ someProperty }}
4     <button ng-click="someAction()"></button>
5   </body>
6 </html>
```

```
1 angular.module('myApp', [])
2 .run(function($rootScope) {
3   $rootScope.someProperty = 'hello computer';
4   $rootScope.someAction = function() {
5     $rootScope.someProperty = 'hello human';
6   };
7 });
```

Live Example²⁹

While useful for demonstration purposes, using `$rootScope` on a regular basis is like using global scope, don't do it.

We can only use `ng-app` once per document. If we want to place multiple apps in a page, we'll need to manually bootstrap our applications. We will talk more about manually bootstrapping apps in the [under the hood](#) chapter.

ng-controller

Instead of defining *actions* and *models* on `$rootScope`, use `ng-controller` which is a built in directive who's purpose is to provide a child scope for the directives that are nested inside. We'll use this directive to place a controller on a DOM element.

`ng-controller` takes a single argument:

²⁹<http://jsbin.com/ICOzeFI/2/edit>

expression (required expression)

The expression is an [angular expression](#)

A child \$scope is simply a JavaScript object that prototypically inherits methods and properties from it's parent \$scope(s), including the application's \$rootScope.

Directives that are nested within an ng-controller have access to this new child \$scope, although be mindful that [the scoping rules](#) for each directive do apply.

Recall that the \$scope object within a controller should be responsible for *actions* and *models* shared by directives in the DOM.



An action refers to a traditional JavaScript method on the \$scope object



A model refers to a traditional JavaScript object {} where transient state should be stored. Persistent state should be bound to a service, which is then responsible for dealing with persisting that model.



It's important to **not** to set a **value*** object(string, boolean or number) directly on the \$scope of a controller for a number of technological and architectural reasons. Data in the DOM should **always** use a .(dot). Following this rule will keep you out of un-expected trouble.



Controllers should be as simple as possible. Although we can use the controller to prototype our functionality, it's a good idea to refactor the logic out using services and directives. See [application architecture](#) for more information.

Using a controller, we can modify our previous example by placing our data and action on a child scope:

```
1 <div ng-controller="SomeCtrl">
2   {{ someModel.someProperty }}
3   <button ng-click="someAction()">Communicate</button>
4 </div>
```

```

1 angular.module('myApp', [])
2 .controller('SomeCtrl', function($scope) {
3   // create a model
4   $scope.someModel = {
5     // with a property
6     someProperty: 'hello computer'
7   }
8   // set actions on $scope itself
9   $scope.someAction = function() {
10     $scope.someModel.someProperty = 'hello human';
11   };
12 });

```

Live Example³⁰

In this iteration of our example, notice two differences from the previous.

First, we're using a child scope of \$rootScope which provides a clean object to work with. This means that actions and models used on this scope will not be available everywhere in the app. They'll only be available to directives within this scope or child scopes.

Secondly, notice that we're explicit about our data model, which as we mentioned, is extremely important. To see why this is important, let's look at another iteration of this example, that nests a second controller inside our existing controller and doesn't set properties on a model object:

```

1 <div ng-controller="SomeCtrl">
2   {{ someBareValue }}
3   <button ng-click="someAction()">Communicate to child</button>
4   <div ng-controller="ChildCtrl">
5     {{ someBareValue }}
6     <button ng-click="childAction()">Communicate to parent</button>
7   </div>
8 </div>

```

³⁰<http://jsbin.com/OYikipe/1/edit>

```

1 angular.module('myApp', [])
2 .controller('SomeCtrl', function($scope) {
3   // anti-pattern, bare value
4   $scope.someBareValue = 'hello computer';
5   // set actions on $scope itself, this is okay
6   $scope.someAction = function() {
7     // sets {{ someBareValue }} inside SomeCtrl and ChildCtrl
8     $scope.someBareValue = 'hello human, from parent';
9   };
10 })
11 .controller('ChildCtrl', function($scope) {
12   $scope.childAction = function() {
13     // sets {{ someBareValue }} inside ChildCtrl
14     $scope.someBareValue = 'hello human, from child';
15   };
16 });

```

Live Example³¹

Because of the way prototype inheritance works with value objects in JavaScript, changing `someBareValue` via an action in the parent *does* change it in the child, but not vis-a-vis.

To see problem try clicking on the child button **first** and then the parent button. Doing so makes it clear that the child controller has *copy*, not a *reference* to `someBareValue`.



JavaScript objects are either copy by value or copy by reference. String, Number and Boolean are copy by value. Array, Object and Function are copy by reference.

Had we set our string as a property of on a model object it would have been shared via reference, which means changing the property on the child will change it on the parent. The following example shows the correct way:

```

1 <div ng-controller="SomeCtrl">
2   {{ someModel.someValue }}
3   <button ng-click="someAction()">Communicate to child</button>
4   <div ng-controller="ChildCtrl">
5     {{ someModel.someValue }}
6     <button ng-click="childAction()">Communicate to parent</button>
7   </div>
8 </div>

```

³¹<http://jsbin.com/UblRIH/a/1/>

```

1 angular.module('myApp', [])
2 .controller('SomeCtrl', function($scope) {
3   // best practice, always use a model
4   $scope.someModel = {
5     someValue: 'hello computer'
6   }
7   $scope.someAction = function() {
8     $scope.someModel.someValue = 'hello human, from parent';
9   };
10 })
11 .controller('ChildCtrl', function($scope) {
12   $scope.childAction = function() {
13     $scope.someModel.someValue = 'hello human, from child';
14   };
15 });

```

Live Example³²

Try clicking on either button. The value always remains in sync.

Note that while this behavior manifests itself most noticeably when using `ng-controller` it will also rear its ugly head when using any directive that creates a new child scope by setting the `scope` property inside its `directive definition` to true. The following built in directives do exactly that:

- `ng-include`
- `ng-switch`
- `ng-repeat`
- `ng-view`
- `ng-controller`
- `ng-if`
- `ng-repeat`

ng-include

Use `ng-include` to fetch, compile and include an external HTML fragment into your current application. The URL of the template is restricted to the same domain and protocol as the application document unless white listed or wrapped as trusted values. Furthermore, you'll need to account for Cross-Origin Resource Sharing and Same Origin Policy to ensure your template loads on all browsers. For example, it won't work for cross domain requests on all browsers and for file:// access on some browsers.

³²<http://jsbin.com/aflyeda/1/edit>



While developing you may run chrome from the command line with `chrome --allow-file-access-from-files` to disable the CORS error. Only go this route in an emergency, e.g., our boss is standing behind us and everything just broke.

Use the `onload` attribute within the same element to run an expression when the template is loaded.

Keep in mind that when using `ng-include` angular automatically creates a new child scope. If you want to use a particular scope, for instance the scope of `ControllerA`, you must invoke `ng-controller="ControllerA"` directive on the same DOM element itself, it will not be inherited from the surrounding scope like usual because a new scope is created when the template loads.

Let's look at an example:

```

1 <div ng-include="/myTemplateName.html"
2   ng-controller="MyController"
3   ng-init="name = 'World'"
4   Hello {{ name }}
5 </div>

```

ng-switch

This directive is used in conjunction with `ng-switch-when` and `on="propertyName"` to switch which directives in our view when the given `propertyName` changes. In the following example, when `person.name` is 'Erik' the div below the text field will be shown and the person will have won:

```

1 <input type="text" ng-model="person.name" />
2 <div ng-switch on="person.name"></div>
3 <p ng-switch-default>And the winner is</p>
4 <h1 ng-switch-when="Erik">{{ person.name }}</h1>

```

Note that we used `ng-switch-default` to output the name of the person until the switch occurred.

[Live Example³³](#)

ng-view

TODO

³³<http://jsbin.com/AVihUdi/2/>

ng-if {ng_if}

Use `ng-if` to completely remove or recreate an element in the DOM based on an expression. If the expression assigned to `ng-if` evaluates to a false value, then the element is removed from the DOM, otherwise a *clone* of the element is re-inserted into the DOM.

`ng-if` differs from `ng-show` and `ng-hide` in that it actually removes and recreates DOM nodes, rather than just showing and hiding them via CSS.

When an element is removed from the DOM using `ng-if` it's associated scope is destroyed. Furthermore, when it comes back into being, a new scope is created and inherits from its parent scope using prototypical inheritance.

It's also important to be aware that `ngIf` recreates elements using their compiled state. If code inside of `ng-if` is loaded, gets manipulated with jQuery, say using `.addClass`, then gets removed because the expression inside the `ng-if` becomes false, when the expression later becomes true again, the DOM element and its children will be re-inserted back into the DOM in their original state, not the state they had when they left the DOM. This means whatever class was added using jQuery's `.addClass` will no longer be present.

```
1 <div ng-if="2 + 2 === 5">
2   Won't see this DOM node, not even in the source code
3 </div>
4
5 <div ng-if="2 + 2 === 4">
6   Hi, I do exist
7 </div>
```

[Live Example³⁴](#)

ng-repeat

Use `ng-repeat` to iterate over a collection and instantiate a new template for each item in the collection. Each item in the collection is given its own template and therefore its own scope. Furthermore, there are a number of special properties exposed on the local scope of each template instance:

- `$index`: iterator offset of the repeated element (0..length-1)
- `$first`: true if the repeated element is first in the iterator.
- `$middle`: true if the repeated element is between the first and last in the iterator.
- `$last`: true if the repeated element is last in the iterator.

³⁴<http://jsbin.com/ezEcamo/1/>

- `$even`: true if the iterator position `$index` is even (otherwise false).
- `$odd`: true if the iterator position `$index` is odd (otherwise false).

We'll use `$odd` and `$even` in the following example to make a repeating list where even items are red and odd item are blue. Remember, that in javascript array's are indexed starting at 0, thus we use `!$even` and `!$odd` to flip the boolean value given by `$even` and `$odd` to account for this.

```

1 <ul ng-controller="PeopleController">
2   <li ng-repeat="person in people" ng-class="{even: !$even, odd: !$odd}">
3     {{person.name}} lives in {{person.city}}
4   </li>
5 </ul>

1 .odd {
2   background-color: blue;
3 }
4 .even {
5   background-color: red;
6 }

1 angular.module('myApp', [])
2 .controller('PeopleController', function($scope) {
3   $scope.people = [
4     {name: "Ari", city: "San Francisco"},
5     {name: "Erik", city: "Seattle"}
6   ];
7 })

```

[Live Example³⁵](#)

ng-init

Use `ng-init` to set up state inside the scope of a directive when that directive is invoked.

The most common use case for using `ng-init` is when creating small examples for educational purposes, like the examples in this chapter.

For anything substantial use create a controller and setup state within a model object.

³⁵<http://jsbin.com/akuYUkey/1/edit>

```

1 <div ng-init="greeting='Hello'; person='World'">
2   {{greeting}} {{person}}
3 </div>
```

Live Example³⁶

{}{ }}

```

1 <div>{{ name }}</div>
```

The {{ }} syntax is a templating syntax that's built into angular. It creates a binding from the containing \$scope to the view. Anytime that the \$scope changes, the view will update automatically due to this binding.

Although it doesn't look like a normal directive, it is in fact a shortcut for using `ng-bind` without needing to create an element. Therefore, it is most commonly used with inline text.

Be aware that using {{ }} within the visible view port of the screen while the page loads may cause a flash of un-rendered content. To prevent this use `ng-bind` instead.

```

1 <body ng-init="greeting = 'Hello World'">
2   {{ greeting }}
3 </body>
```

Live Example³⁷

ng-bind

Although we can use the {{ }} template syntax within our views (these are interpolated by Angular), we can mimic this behavior with the `ng-bind` directive.

```

1 <body ng-init="greeting = 'Hello World'">
2   <p ng-bind="greeting"></p>
3 </body>
```

Live Example³⁸

When we use the {{ }} syntax, our HTML document will load the element and not render it immediately, 'causing a flash of unrendered content (FOUC, for short). We can prevent this FOUC from being exposed by using `ng-bind` and binding our content to the element. The content will then be rendered as the child text node of the element on which `ng-bind` is declared.

³⁶<http://jsbin.com/OZENuhO/1/>

³⁷<http://jsbin.com/ODUxeho/1/edit>

³⁸<http://jsbin.com/esihUJ/1/edit>

ng-cloak

An alternative to using to using `ng-bind` to prevent a flash of un-rendered content is to use `ng-cloak` on the element containing `{} {}`:

```
1 <body ng-init="greeting = 'Hello World'">
2   <p ng-cloak>{{ greeting }}</p>
3 </body>
```

[Live Example³⁹](#)

ng-bind-template

Similar to the `ng-bind` directive, we can use the `ng-bind-template` directive if we want to *bind* multiple [expressions](#) to the view.

```
1 <div ng-bind-template="{{ message }} {{ name }}>
2 </div>
```

ng-model

The `ng-model` directive binds an `input`, `select`, `textarea` or custom form control to a property on the surrounding scope. It handles and provides validation, sets related css classes on the element(`ng-valid`, `ng-invalid`, etc) and registers the the control with its parent form. It will bind to the property given by evaluating the expression on the current scope. If the property doesn't already exist on this scope, it will be created implicitly and added to the scope.

`ngModel` should always be used with a `model` property on the `$scope`, not as a raw property on the scope itself. This is so that using `ng-model="someProperty"` twice in the same scope or inherited scope doesn't shadow the first. For example:

```
1 <input type="text" ng-model="modelName.someProperty" />
```

is the correct way to think about and practically use `ngModel` properly.

The bottom line is always have a `.` in your `ng-model`'s. For in depth discussion and example on this topic, see [ng-controller](#) earlier in this chapter.

³⁹<http://jsbin.com/AJEboLO/1/edit>

ng-show/ng-hide

ng-show and ng-hide show or hide the given HTML element based on the expression provided to the attribute. When the expression provided to the ng-show attribute is `false` the element is hidden. Similarly, when the expression given to ng-hide is true, the element is hidden.

The element is shown or hidden by removing or adding the ng-hide CSS class onto the element. The `.ng-hide` CSS class is predefined in AngularJS and sets the display style to none (using an !important flag).

```

1 <div ng-show="2 + 2 == 5">2 + 2 isn't 5, don't show</div>
2 <div ng-show="2 + 2 == 4">2 + 2 is 4, do show</div>
3
4 <div ng-hide="2 + 2 == 5">2 + 2 isn't 5, don't hide</div>
5 <div ng-hide="2 + 2 == 4">2 + 2 isn't 5, do hide</div>
```

[Live Example⁴⁰](#)

ng-change

Evaluate the given expression when the input changes. Because we're dealing with input, we must use this directive in conjunction with `ngModel`.

```

1 <div ng-controller="EquationController">
2   <input type="text" ng-model="equation.x" ng-change="change()" />
3   <code>{{ equation.output }}</code>
4 </div>
```

```

1 angular.module('myApp', [])
2 .controller('EquationController', function($scope) {
3   $scope.equation = {};
4   $scope.change = function() {
5     $scope.equation.output = Number($scope.equation.x) + 2;
6   };
7 });
```

[Live Example⁴¹](#)

In the above example, we run the `change()` function whenever `equation.x` is changed by entering text into the text field.

⁴⁰<http://jsbin.com/ihOkagE/1/>

⁴¹<http://jsbin.com/onUXuxO/1/edit>

ng-form

Use `ng-form` when you need to nest a form within another form. The normal html `<form>` tag doesn't allow you to nest your forms, but `ng-form` will.

This means that the outer form is valid when all of the child forms are valid as well. This is especially useful when dynamically generating forms using `ng-repeat`.

Because you cannot dynamically generate the `name` attribute of input elements using interpolation, you'll need to wrap each set of repeated inputs in an `ng-form` directive and nest these in an outer form element.

The following css classes are set, depending on the validity of the form:

- `ng-valid` when form is valid
- `ng-invalid` when form is invalid
- `ng-pristine` when form is pristine
- `ng-dirty` when form is dirty

Angular will not submit the form to the server unless the form has an `action` attribute specified.

To specify which javascript method should be called when a form is submitted use one of the following two directives:

- `ng-submit` directive on the form element
- `ng-click` directive on the first button or input field of type submit (`input[type=submit]`)

To prevent double execution of the handler, use only the `ng-submit` or `ng-click` directives.

In the following examples, we want to dynamically generate a form based on a JSON response from the server. We'll use `ng-repeat` to loop over the fields we get back from the server. Because we cannot dynamically generate the `name` attribute, and because we need the `name` attribute to perform validation, we'll loop over the fields and create a new form for each one.

Because Angular forms that use `ng-form` instead of `form` be nested, and because the parent form is not valid until it's child forms are valid we can both dynamically generate a form with child forms and use validation. Yes, we can have our cake and eat it too.

Let's first view the JSON we're hard coding, as though it came from the server:

```

1 angular.module('myApp', [])
2 .controller('FormController', function($scope) {
3   $scope.fields = [
4     {placeholder: 'Username',.isRequired: true},
5     {placeholder: 'Password',isRequired: true},
6     {placeholder: 'Email (optional)',isRequired: false}
7   ];
8
9   $scope.submitForm = function() {
10     alert("it works!");
11   };
12 });

```

Now, let's take a look at using that data to generate a dynamic form with validation:

```

1 <form name="signup_form" ng-controller="FormController" ng-submit="submitForm()" \ 
2 novalidate>
3   <div ng-repeat="field in fields" ng-form="signup_form_input">
4     <input type="text"
5       name="dynamic_input"
6       ng-required="field.isRequired"
7       ng-model="field.name"
8       placeholder="{{field.placeholder}}"/>
9     <div ng-show="signup_form_input.dynamic_input.$dirty && signup_form_input.dyn\ 
10 amic_input.$invalid">
11       <span class="error" ng-show="signup_form_input.dynamic_input.$error.require\ 
12 d">The field is required.</span>
13     </div>
14   </div>
15   <button type="submit" ng-disabled="signup_form.$invalid">Submit All</button>
16 </form>

1 input.ng-invalid {
2   border: 1px solid red;
3 }
4
5 input.ng-valid {
6   border: 1px solid green;
7 }

```

[Live Example⁴²](#)

⁴²<http://jsbin.com/UduNeCA/1/edit>

ng-click

Use `ng-click` to specify a method or expression to run on the containing scope when the element is clicked.

```

1 <div ng-controller="CounterController">
2   <button ng-click="count = count + 1" ng-init="count=0">
3     Increment
4   </button>
5   count: {{count}}
6
7   <button ng-click="decrement()">
8     Decrement
9   </button>
10 </div>
```

```

1 angular.module('myApp', [])
2 .controller('CounterController', function($scope) {
3   $scope.decrement = function() {
4     $scope.count = $scope.count - 1;
5   };
6 })
```

[Live Example⁴³](#)

ng-select

Use the `ng-select` directive to bind data to a HTML `<select>` element. This directive can be used in conjunction with `ng-model` and `ng-options` to provide sophisticated and highly performant dynamic forms.

`ng-options` takes a comprehension expression for its attribute value, which is just a fancy way of saying it can take an array or an object and loop over its contents to provide the options available when using the `select` tag. It comes in one of the following forms:

- for array data sources:
 - label for value in array
 - select as label for value in array
 - label group by group for value in array
 - select as label group by group for value in array track by trackexpr

⁴³<http://jsbin.com/uGipUBU/2/edit>

- for object data sources:
 - label for (key , value) in object
 - select as label for (key , value) in object
 - label group by group for (key, value) in object
 - select as label group by group for (key, value) in object

Let's look at an example of using `ng-select`:

```

1 <div ng-controller="CityController">
2   <select ng-model="city" ng-options="city.name for city in cities">
3     <option value="">Choose City</option>
4   </select>
5   Best City: {{ city.name }}
6 </div>

1 angular.module('myApp', [])
2 .controller('CityController', function($scope) {
3   $scope.cities = [
4     {name: 'Seattle'},
5     {name: 'San Francisco'},
6     {name: 'Chicago'},
7     {name: 'New York'},
8     {name: 'Boston'}
9   ];
10 });

```

[Live Example⁴⁴](#)

ng-submit

Use `ng-submit` to bind an expression to an `onsubmit` event. It also prevents the default action(sending the request and reloading the page) but *only if the form does not contain an action attribute.*

⁴⁴<http://jsbin.com/iQelOxi/1/edit>

```

1 <form ng-submit="submit()" ng-controller="FormController">
2   Enter text and hit enter:
3   <input type="text" ng-model="person.name" name="person.name" />
4   <input type="submit" name="person.name" value="Submit" />
5   <code>people={{people}}</code>
6   <ul ng-repeat="(index, object) in people">
7     <li>{{ object.name }}</li>
8   </ul>
9 </form>

1 angular.module('myApp', [])
2 .controller('FormController', function($scope) {
3
4   $scope.person = {
5     name: null
6   };
7
8   $scope.people = [];
9
10  $scope.submit = function() {
11    if ($scope.person.name) {
12      $scope.people.push({name: $scope.person.name});
13      $scope.person.name = '';
14    }
15  };
16 });

```

[Live Example⁴⁵](#)

ng-class

Use `ng-class` to dynamically set the class of an element by databinding an expression that represents all classes to be added. Duplicate classes will not be added. When the expression changes, the previously added classes are removed and only then the new classes are added.

Let's use `ng-class` to add the class `.red` to the div found below whenever the random number drawn is above 5.

⁴⁵<http://jsbin.com/ONlcAC/1/edit>

```

1 <div ng-controller="LotteryController">
2   <div ng-class="{red: x > 5}" ng-if="x > 5">
3     You won!
4   </div>
5   <button ng-click="x = generateNumber()" ng-init="x = 0">Draw Number</button>
6   <p>Number is: {{ x }}</p>
7 </div>
```

```

1 .red {
2   background-color: red;
3 }
```

```

1 angular.module('myApp', [])
2 .controller('LotteryController', function($scope) {
3   $scope.generateNumber = function() {
4     return Math.floor((Math.random()*10)+1);
5   }
6 })
```

Live Example⁴⁶

]

ng-attr-[suffix]

When Angular compiles the DOM it looks for expressions within `{{ some expression }}` brackets. These expressions automatically registered with the \$watch service and update as part of the normal \$digest cycle:

```

1 <!-- updated when `someExpression` on the $scope is updated -->
2 <h1>Hello {{ someExpression }}</h1>
```

Sometimes however web browsers are picky about what attributes they allow. SVG is one such instance:

⁴⁶<http://jsbin.com/IvEcUci/1/edit>

```
1 <svg>
2   <circle cx="{{cx}}"></circle>
3 </svg>
```

Running the code above will error, telling us we have an invalid attribute. To fix this we can use `ng-attr-cx`. Notice the `cx` is named after the attribute we would like to define. Within the string we can use an expression with `{{}}` and achieve the result we were looking for above.

```
1 <svg>
2   <circle ng-attr-cx="{{cx}}"><circle>
3 </svg>
```

Directive's Explained

The goal of this chapter is to explicitly lay out all of the options and capabilities directives have to offer when building mature client side applications.

Directive Definition

A directive is defined using the `.directive()` method, one of the many methods available on our application's angular module.

```
1 angular.module('myDirective', function($timeout, UserDefinedService) {  
2   // directive definition goes here  
3 })
```

The `directive()` method takes two arguments:

name (string)

The name of the directive as a string that we'll refer to inside of our views.

factory_function (function)

The factory function returns an object that defines how the directive behaves. It is expected to return an object providing options that tell the `$compile` service how the directive should behave when it is invoked in the DOM.

```
1 angular.application('myApp', [])  
2 .directive('myDirective', function() {  
3   // A directive definition object  
4   return {  
5     // directive definition is defined via options  
6     // which we'll override here  
7   };  
8 });
```

We can also return a function instead of an object to handle this directive definition, but it is best practice to return an object as we've done above. When a function is returned is often referred to as the `postLink` function, which only allows us to define the `link` function for the directive. Returning a function instead of an object limits us to a narrow ability to customize our directive, thus is good for only simple directives.

When angular bootstraps our app, it will register the returned object by the name provided as a string via the first argument. The angular compiler parses the DOM of our main HTML document looking for elements, attributes, comments or class names using that name looking for these directives. When it finds one that it knows about, it will use the directive definition to place the DOM element on the page.

```
1 <div my-directive></div>
```



To avoid collision with future HTML standard's it's best practice to prefix a custom directive with a custom namespace. Angular itself has chosen the `ng-` prefix, so use something other than that. In the examples that follow, we'll use the `my-` prefix, e.g., `my-directive`.

The factory function we define for a directive is only invoked *once*, when the compiler matches the directive the first time. Just like the `.controller` function, a directive's factory function is invoked using `$injector.invoke`.

When Angular encounters the named directive in the DOM, it will invoke the directive definition we've registered, using the name to look up the object we've registered. At this point the [directive lifecycle](#) begins, starting with the [compile method](#) and ending with the [link method](#). We'll dive into the specifics of this process later in this chapter.

Let's look at all the available options we can provide to a directive definition.



A JavaScript object is made up of keys and values. When the value for a given key is set to a String, Boolean, Number, Array or Object, we call the key a *property*. When we set the key to a function, we call it a *method*.

The possible options are shown below. The value of each key provides the signature of either the method or the type we can set the property to:

```

1 angular.module('myApp', [])
2 .directive('myDirective', function() {
3   return {
4     restrict: String,
5     priority: Number,
6     terminal: Boolean,
7     template: String or Template Function: function(tElement, tAttrs) {...},
8     templateUrl: String,
9     replace: Boolean or String,
10    scope: Boolean or Object,
11    transclude: Boolean,
12    controller: String or function($scope, $element, $attrs, $transclude, otherIn\
13 jectables) { ... },
14    controllerAs: String,
15    require: String,
16    link: function(scope, iElement, iAttrs) { ... },
17    compile: return an Object OR function(tElement, tAttrs, transclude) {
18      return {
19        pre: function(scope, iElement, iAttrs, controller) { ... },
20        post: function(scope, iElement, iAttrs, controller) { ... }
21      }
22      // or
23      return function postLink(...) { ... }
24    }
25  };
26 });

```

Restrict (string)

restrict is an optional argument. It is responsible for telling Angular which format our directive will be declared as in the DOM. By default a custom directive is expected to be declared as an attribute, meaning the restrict option is set to A.

The available options are as follows:

- E (an element) {lang="html"}
- A (an attribute, default) {lang="html"} <div my-directive="expression"></div>
- C (a class) {lang="html"} <div class="my-directive: expression;"></div>
- M (a comment) {lang="html"} <- directive: my-directive expression ->

These options can be used alone or in combination:

```
1 angular.module('myDirective', function() {
2   return {
3     restrict: 'EA' // either an element or an attribute
4   };
5 });
```

In this case the directive can be declared as an attribute or an element:

```
1 <-- as an attribute -->
2 <div my-directive></div>
3 <-- or as an element -->
4 <my-directive></my-directive>
```

Attributes are the default and most common form of directive because they will work across all browsers, including older versions of Internet Explorer without having to register a new tag in the head of the document. See the [chapter on Internet Explorer](#) for more information on this topic.



Avoid using comments to declare a directive. This format was originally introduced as a way to create directives that span multiple elements. This was especially useful, for example, when using `ng-repeat` inside a `<table>` element. However, as of Angular 1.2 `ng-repeat` provides `ng-repeat-start` and `ng-repeat-end` as a better solution to this problem, minimizing the need for the comment form of a directive even more so. If curious however, take a look at the Chrome Dev tools elements tab when using `ng-repeat` to see comments being used under the hood.

Element or Attribute?

Use an element when creating something new on the page that will encapsulate a self contained piece of functionality. For example, if we're creating a clock (and could care less about supporting [old versions of Internet Explorer](#)) we'd make clock directive and declare it in the DOM like so:

```
1 <my-clock></my-clock>
```

This tells users of our directive that we're specifying a whole piece of our application. Our clock is not decorating or augmenting a pre-existing clock, instead it's declaring a whole unit. While we could have used an attribute in this scenario (Angular doesn't care) we've chosen to use an element because it clarifies our intent.

Use an attribute when decorating an existing element with data or behavior. Using our clock example, lets pretend we're interested in an analog version of the clock:

```
1 <my-clock clock-display="analog"></my-clock>
```

The choice usually comes down to whether the directive being defined provides the core behavior of a component on the page, or whether the directive will be decorating or augmenting a core directive with optional behavior, state or anything else one might find while programming in the wild (like analog output for a clock).

The guiding principle here is that the format of a directive tells a story about our applications and reveals the intent of each piece, creating exemplary code that is easy to understand and share with others.

The other distinction that is important to make for a given directive is whether it creates, inherits, or isolates itself from the scope of its containing environment. This child parent relationship plays another key role in composition and re-usability of directives, a topic we'll spend more time discussing when we talk about the [scope](#) of directive.

Priority (number)

The priority option can be set to a number. Most directives omit this option in which case it defaults to 0. However, there are cases where a high priority is important. For example `ngRepeat` sets this option 1000 so that it *always* gets invoked before other directives on the same element.

If an element is decorated with two directives that have the same priority then the first directive declared on the element will be invoked first. If one of the directives has a higher priority than the other, then it doesn't matter which is declared first as the one with the higher priority will *always* run first.



`ngRepeat` has the highest priority of any built in directive. It is always invoked before other directives on the same element. Performance is a key factor here. We'll learn more about performance when we discuss the [compile](#).

Terminal (boolean)

`terminal` is a boolean option. It can be set to true or false.

The `terminal` option is used to tell angular to stop invoking any further directives on an element that have a higher priority. All directives with the same priority will be executed however.

As a result, don't further decorate an element if it's already been decorated with a directive that is `terminal` and has equal or higher priority, it won't be invoked.

Template (string|function)

template is optional. If provided, it must be set to either:

- a string of HTML
- a function which takes two arguments, tElement and tAttrs and returns a string value representing the template. The t in tElement and tAttrs stands for template as opposed to instance. We'll discuss the difference between a template element/attribute vs an instance element/attribute when we cover the [link](#) and [compile](#) options.

The template string is treated no differently than any other HTML in Angular. It has a scope which can be accessed using double curly markup like `{} expression {}`.

When a template string contains more than one DOM element or only a single text node, it must be wrapped in a parent element. In other words, a root DOM element must exist:

```

1 template: '\
2   <div> <-- single root element --> \
3     <a href="http://google.com">Click me</a> \
4     <h1>When using two elements, wrap them in a parent element</h1> \
5   </div> \
6 '

```

Furthermore, note the use of slashes, \, at the end of lines. This is so that Angular can parse multi-line string correctly. A better option in production code is to use the [templateUrl](#) option because multi-lines strings are a nightmare to look at and maintain.

One of the most important features to understand about a template string or templateUrl is how it's gets its scope. In the the [beginning_directives](#) we touched upon how scope is passed into a directive.

templateUrl (string|function)

templateUrl is optional. If provided, it must be set to the either:

- the path to a HTML file, as a string
- as a function which takes two arguments tElement and tAttrs. The function must return the path to an HTML file as a string.

In either case, the the template URL is passed through the built in [security](#) layer provided by Angular, specifically \$getTrustedResourceUrl, which protects our templates from being fetched from untrusted resources.

By default the HTML file will be requested on demand via ajax when the directive is invoked. Because of this, two important factors should be kept in mind:

- When developing locally, run a server in the background to serve up the local HTML templates from your file system. Failing to do so will raise a Cross Origin Request Script error (CORS).
- The template loading is asynchronous, which means compilation/linking is suspended until the template is loaded.

Having to wait for a large number of templates to asynchronously load via ajax can really slow down a client side application. To prevent this, it's possible to cache one or more HTML templates prior to deploying an application. This is a better option in most cases because an ajax request will not be made providing better performance by minimizing the number of requests made. For more information about caching, check out the in-depth discussion on [caching](#).

After a template has been fetched, it's cached in the `$templateCache` services provided by angular by default. In production, we can pre-cache these templates into a javascript file that defines our templates so we don't have to fetch the templates over XHR. For more information about how this works, see the [next steps](#) chapter.

replace (boolean)

`replace` is optional. If provided it must be set to `true`. It is set to `false` by default.

By default, `replace` is set to `false`. This means that the directive's template will be appended as a child node within the element where the directive was invoked:

```
1 <div some-directive></div>

1 .directive('someDirective' function() {
2   return {
3     template: '<div>some stuff here</div>'
4   }
5 })
```

The result when the directive is invoked will be (this is default behavior, when `replace` is `false`):

```
1 <div some-directive>
2   <div>some stuff here</div>
3 </div>
```

If we set `replace true`:

```

1 .directive('someDirective' function() {
2   return {
3     replace: true // MODIFIED
4     template: '<div>some stuff here</div>'
5   }
6 })

```

Then the result when the directive is invoked will be:

```
1 <div>some stuff here</div>
```

Directive Scope

In order to fully understand the rest of the options available inside a directive definition object we'll need to have an understanding of how scope works.

A special object, known as the \$rootScope is initially created when we declare the ng-app directive in the DOM:

```

1 <div ng-app="myApp"
2   ng-init="someProperty = 'some data'"></div>
3   <div ng-init="siblingProperty = 'more data'">
4     Inside Div Two
5     <div ng-init="aThirdProperty"></div>
6   </div>

```

In the above code we set three properties on the root scope of our app called `someProperty`, `siblingProperty` and `aThirdProperty`.

From here on out every directive invoked within our DOM will either:

- a) directly use the same object
- b) create a new object that inherits from the object
- c) create an object that is isolated from the object

The example above shows the first case. The second `div` is a sibling element that has get and set access to the root scope. Furthermore, inside this `div` is another `div` that also has get/set access to the exact same root scope.

Just because a directive is nested within another directive does not necessarily mean it's scope has been changed. By default child directives are given access to the **exact same** scope as their parent DOM nodes. The reason for this can be understood by learning about the `scope` directive option, which is set to `false` by default.

Scope Option (boolean|object)

The scope option is optional. It can be set to `true` or an object, `{}`. By default it is set to `false`.

When `scope` is set to `true`, a new scope object is created that prototypically inherits from its parent scope.

If multiple directives on an element provide an isolate scope, only one new scope is applied. Root elements within the template of a directive always get a new scope, thus for those objects, `scope` is set to `true` by default.

The built in `ng-controller` directive exists for sole purpose of creating a new child scope that prototypically inherits from the surrounding scope. It creates a new scope that inherits from the surrounding scope.

Let's amend our last example, with this knowledge in hand:

```

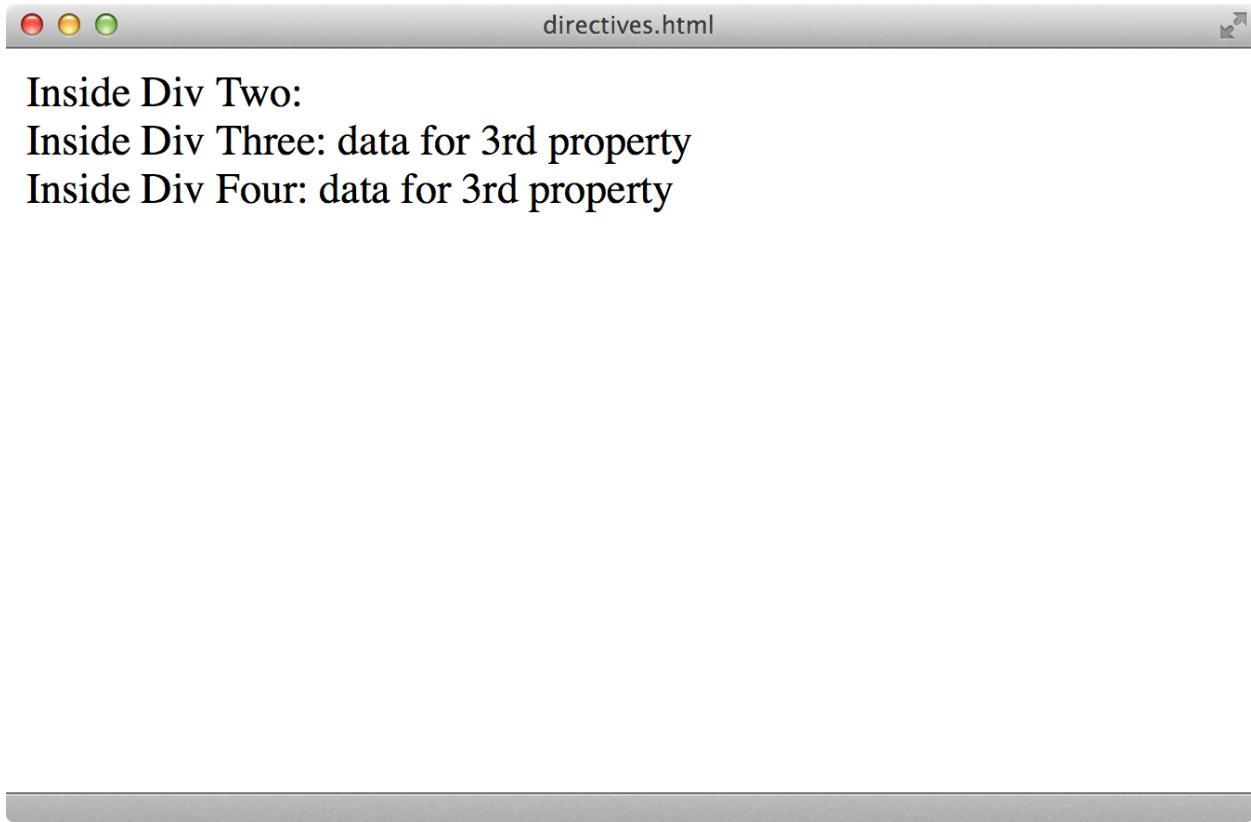
1 <div ng-app="myApp"
2     ng-init="someProperty = 'some data'"></div>
3 <div ng-init="siblingProperty = 'more data'">
4     Inside Div Two: {{ aThirdProperty }}
5     <div ng-init="aThirdProperty = 'data for 3rd property'"
6         ng-controller="SomeCtrl">
7         Inside Div Three: {{ aThirdProperty }}
8         <div ng-init="aFourthProperty">
9             Inside Div Four: {{ aThirdProperty }}
10        </div>
11    </div>
12 </div>
```

If we run the previous code by itself it fails because we haven't defined an associated controller in our JavaScript, so let's do that:

```

1 angular.module('myApp', [])
2 .controller('SomeCtrl', function($scope) {
3     // we can leave it empty, it just needs to be defined
4 })
```

If we reload the page, we can see that inside the second div `{{ aThirdProperty }}` is undefined and therefore outputs nothing. However inside the third div the value we set inside our inherited scope `data for a 3rd property` is shown.



Directives

Furthermore, to prove the point that this scope flows downwards during inheritance, but not upwards, let's make another child scope and check to see that `{} aThirdProperty {}` inherits its value from its parent.

```
1 <div ng-app="myApp"
2   ng-init="someProperty = 'some data'"></div>
3 <div ng-init="siblingProperty = 'more data'">
4   Inside Div Two: {{ aThirdProperty }}
5   <div ng-init="aThirdProperty = 'data for 3rd property'"
6     ng-controller="SomeCtrl">
7     Inside Div Three: {{ aThirdProperty }}
8     <div ng-controller="SecondCtrl">
9       Inside Div Four: {{ aThirdProperty }}
10      </div>
11    </div>
12  </div>
```

We'll need to update our JavaScript so that `SecondCtrl` is defined:

```

1 angular.module('myApp', [])
2 .controller('SomeCtrl', function($scope) {
3   // we can leave it empty, it just needs to be defined
4 })
5 .controller('SecondCtrl', function($scope) {
6   // also can be empty
7 })

```

To create our own directive that who's scope prototypically inherits from the outside world, set the scope property to true:

```

1 angular.module('myApp', [])
2 .directive('myDirective', function() {
3   return {
4     restrict: 'A',
5     scope: true
6   }
7 })

```

Now let's use our directive to alter the scope of the DOM:

```

1 <div ng-app="myApp"
2   ng-init="someProperty = 'some data'"></div>
3 <div ng-init="siblingProperty = 'more data'">
4   Inside Div Two: {{ aThirdProperty }}
5   <div ng-init="aThirdProperty = 'data for 3rd property'"
6     ng-controller="SomeCtrl">
7     Inside Div Three: {{ aThirdProperty }}
8     <div ng-controller="SecondCtrl">
9       Inside Div Four: {{ aThirdProperty }}
10      <br>
11      Outside myDirective: {{ myProperty }}
12      <div my-directive ng-init="myProperty = 'wow, this is cool'">
13        Inside myDirective: {{ myProperty }}
14        <div>
15          </div>
16        </div>
17      </div>

```

[Live JS Bin⁴⁷](#)

Now that we understand how surrounding scope and inherited scope work, there remains only one piece to the scope puzzle, isolate scope.

⁴⁷<http://jsbin.com/ITEBAF/1/edit>

Isolate Scope

Isolate scope is likely the most confusing of the three options available when setting the scope property, but also the most powerful. Isolate scope is based on the ideology present in Object Oriented programming. Languages like Small Talk and design principles like SOLID find they're into Angular via directives that use isolate scope.

The main use case for such directives is re-usable widgets that can be shared and used in unexpected contexts without polluting the scope around them or having their internal scope corrupted inadvertently.

To create a directive with isolate scope we'll need to set the scope property of the directive to an empty object, {}. Once we've done that no outer scope is available:

```
1 Outside myDirective: {{ myProperty }}  
2 <div my-directive ng-init="myProperty = 'wow, this is cool'">  
3   Inside myDirective: {{ myProperty }}  
4 </div>
```

```
1 angular.module('myApp', [])  
2 .directive('myDirective', function() {  
3   return {  
4     restrict: 'A',  
5     scope: {}  
6   };  
7 })
```

[Live JS Bin⁴⁸](#)

A look at the JS Bin will tell you that it seems almost identical to setting scope to true. There is a difference by making another directive with inherited scope and comparing the two, as we can see here:

⁴⁸<http://jsbin.com/eguDipa/1/edit>

```
1 <div ng-init="myProperty = 'wow, this is cool'"></div>
2 Surrounding scope: {{ myProperty }}
3 <div my-inherit-scope-directive="SomeCtrl">
4   Inside an directive with inherited scope: {{ myProperty }}
5 </div>
6 <div my-directive>
7   Inside myDirective, isolate scope: {{ myProperty }}
8 <div>
```

The JavaScript code:

```
1 angular.module('myApp', [])
2 .directive('myDirective', function() {
3   return {
4     restrict: 'A',
5     scope: {}
6   };
7 })
8 .directive('myInheritScopeDirective', function() {
9   return {
10    restrict: 'A',
11    scope: true
12  };
13 })
```

[Live JS Bin⁴⁹](#)

With the most important concepts about scope out of the way, we *can bind* properties on our isolated scope with the outside world, allowing us to *poke holes* through the isolate scopes.

Two Way Data Binding

Perhaps the most powerful feature in Angular, two way data binding allows us to bind lock the value of a property inside the private scope of our directive to the value of an attribute available within the DOM. In the [previous chapter on directives](#) we looked at a good example of how `ng-model` provides two way data binding with the outside world and a custom directive we created, that in many way mirrored the behavior that `ng-bind` itself provides. Review that chapter and practice the example to gain a greater understanding of this important concept.

⁴⁹<http://jsbin.com/OxAlek/1/edit>

Transclude

`transclude` is optional. If provided it must be set to `true`. It is set to `false` by default.

Transclusion is sometimes considered an advanced topic, but once it makes sense it'll fit right in with the rules we've just learned about scope. We'll also see that it's a very powerful addition to our tool set, especially when building customizable chunks HTML that can be shared among a team, between projects and with the rest of the Angular community.

Transclusion is most often used for creating re-usable widgets. A great example is a modal box or a navbar.

Transclude allows us to pass in an entire template, including its scope, to a directive. This gives us the opportunity to pass in arbitrary contents and arbitrary scope to a directive. In order for scope to be passed in the `scope` option must be isolated, `{}` or set to `true`. If the `scope` option is not set, then the scope available inside the directive will be applied to the template passed in.



Only use `transclude: true` when you want to create a directive that wraps arbitrary content.

Transclusion makes it easy to allow users of our directive to customize all these aspects at once by allowing them to provide their own HTML template that has its own state and behavior.

Let's walk through a small example where we provide a re-usable directive that others can customize.

Let's create a reusable sidebar box, similar to the sidebars that are popular with wordpress blogs. The usage is that we'll want our styling of these boxes to stay consistent, but want to reduce the amount of HTML that we have to write for each one.

For instance, let's say we want to create a sidebar box that takes a title and some HTML content, like so:

```
1 <div sidebox title="Links">
2   <ul>
3     <li>First link</li>
4     <li>Second link</li>
5   </ul>
6 </div>
```

We can create the `sidebox` directive pretty simply by creating a directive with the `transclude` option set to `true`.

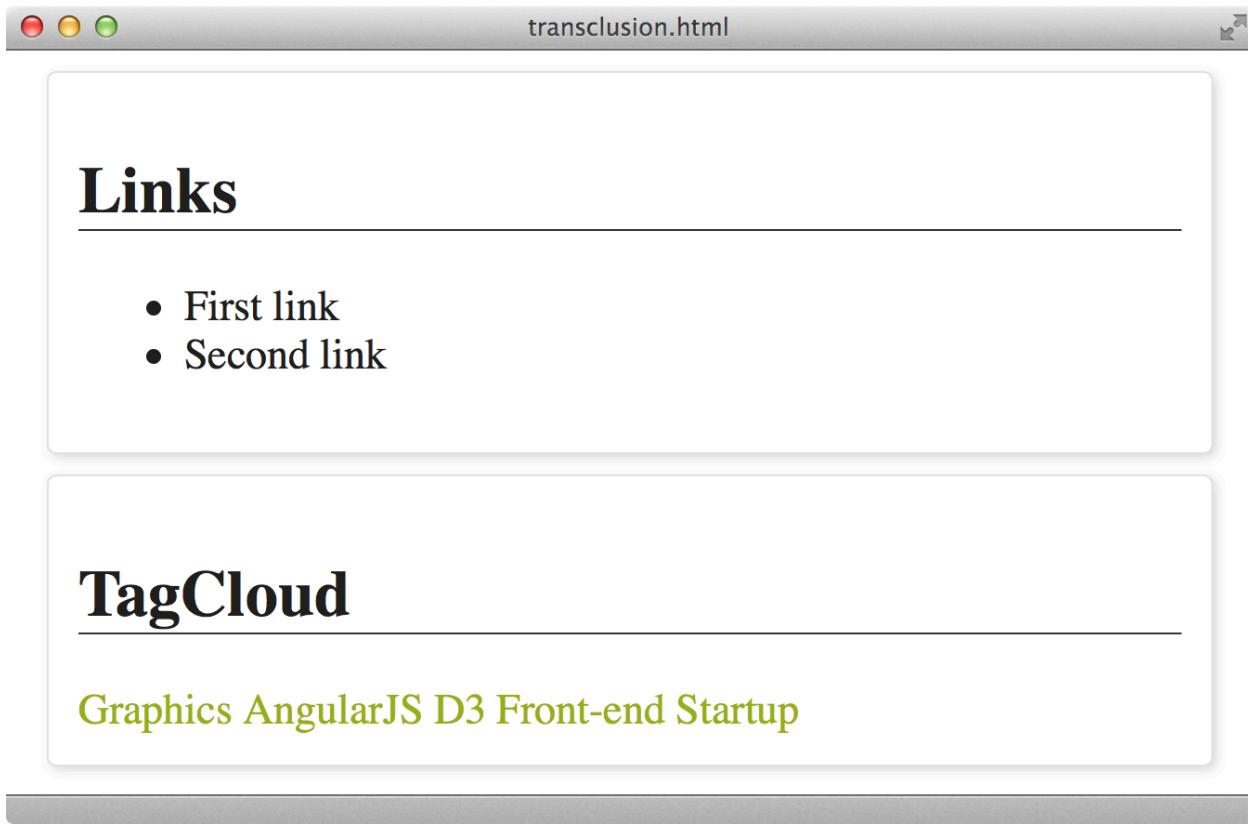
```
1 angular.module('myApp', [])
2 .directive('sidebox', function() {
3   return {
4     restrict: 'EA',
5     scope: {
6       title: '@'
7     },
8     transclude: true,
9     template: '<div class="sidebox"> \
10       <div class="content"> \
11         <h2 class="header">{{ title }}</h2> \
12         <span class="content" ng-transclude> \
13           </span> \
14         </div> \
15       </div> '
16   }
17});
```

This tells the angular compiler where it finds the `ng-transclude` directive is where it should place the content that's captured from inside the DOM element.

Now, we can reuse this directive with the `transclusion` to provide a secondary element without needing to worry about the styles and the layout.

For instance, this will yield us two boxes with consistent styles:

```
1 <div sidebox title="Links">
2   <ul>
3     <li>First link</li>
4     <li>Second link</li>
5   </ul>
6 </div>
7 <div sidebox title="TagCloud">
8   <div class="tagcloud">
9     <a href="">Graphics</a>
10    <a href="">AngularJS</a>
11    <a href="">D3</a>
12    <a href="">Front-end</a>
13    <a href="">Startup</a>
14  </div>
15 </div>
```



sidebox

Transclude will make watching for model property changes from within the controller of a directive not work properly. This is why best practice always recommends using the \$watch service from within the link function.

Controller (string|function)

The controller option takes a string or a function. When set to a string, the name of the string is used to look up a controller constructor function registered elsewhere in our application:

```
1 angular.module('myApp', [])
2 .directive('myDirective', function() {
3   restrict: 'A', // always required
4   controller: 'SomeCtrl'
5 })
6
7 // elsewhere in our application
8 // either in the same file or another
9 // one included by our index.html
10 angular.module('myApp')
```

```
11 .controller('SomeCtrl', function($scope, $element, $attrs, $transclude) {  
12   // controller logic goes here  
13 })
```

A controller can be defined inline within a directive by setting the controller function as an anonymous constructor function:

```
1 angular.module('myApp', [])  
2 .directive('myDirective', function() {  
3   restrict: 'A',  
4   controller:  
5     function($scope, $element, $attrs, $transclude) {  
6       // controller logic goes here  
7     }  
8 })
```

As the above examples suggest the arguments that we can pass into a controller are:

\$scope

The current scope that gets associated with the directive element.

\$element

The current element directive element

\$attrs

The cattributes object for the current element. For instance, the following element:

```
1 <div id="aDiv" class="box"></div>
```

Has the attribute object of:

```
1 {  
2   id: "aDiv",  
3   class: "box"  
4 }
```

\$transclude

A transclude linking function pre-bound to the correct transclusion scope.

This `transclude` linking function is the function that will get run to actually create a clone of the element and manipulates the DOM.



It goes against the *Angular Way* to manipulate the DOM inside of a controller, but it is possible through the linking function. It's a best practice to only use this `transcludeFn` inside the `compile` option.

For example, let's be lazy and say that we just want to add a linktag through the use of a directive, we can accomplish this inside the controller using the `$transclude` function like so:

```

1 angular.module('myApp')
2 .directive('link', function() {
3   return {
4     restrict: 'EA',
5     transclude: true,
6     controller:
7       function($scope, $element, $transclude) {
8         $transclude(function(clone) {
9           var a = angular.element('<a>');
10          a.attr('href', clone.text());
11          a.text(clone.text());
12          $element.append(a);
13        });
14      }
15    }
16  });

```

A directive's controller can often be interchanged with the directive's `link` function. The main use case for a controller is when we want to provide re-usable behavior between directives. As the `link` function is only available inside the current directive any behavior defined within is not shareable.



The `link` function provides **isolation** between directives, while the `controller` defines **shareable** behavior.

However, because a directive can require the controller of another directive, Controller's are a great place to place actions we may want to use in more than one directive.

The `controller` option attaches a controller to the template of the directive, acting just like `ngController` would if it was the parent scope of the directive's template.

Using the `controller` option is good when we want to expose an api to other directives, otherwise we can rely on `link` to provide us local functionality for the directive element. It's better to use `link` when we use `scope.$watch()` or when we're doing any interaction with the live scope of the DOM.

Technically, the `$scope` passed into a controller is passed in before the DOM is actually rendered to the screen. In certain situations, for example when working with transcludes, the scope inside a controller may reflect a different scope than we might expect, and the `$scope` object in such cases is not always guaranteed to update.



Use the `scope` argument passed into the `link` function when expecting to interact with the instance of the scope on screen.

ControllerAs (string)

The `controllerAs` option enables us to set a controller alias. This allows our controller to be published under this name. This gives the scope access to the `controllerAs` name.

require (string|array)

The `require` option can be set to a string or an array of strings. The string(s) contain the name of another directive. `require` is used to inject the controller of the required directive as the fourth parameter of the the current directive's linking function.

The `require` option takes a string or an array of string. The string or strings (if it is an array) provided are the names of directives that reside in the current scope of the current directive.

The `scope` setting will affect what the surrounding scope refers to, be it an isolate scope, an independent scope, or no scope at all. In all cases, the template of the current directive will be consulted when looking for controllers for where the template declares `ng-controller`.

Without using the `^` prefix, the directive only looks for the controller on it's own element.

```
1 // ...
2 restrict: 'EA',
3 require: 'ngModel'
4 // ...
```

This directive definition will *only* look for the `ng-model=""` definition in the local scope of the directive.

```
1 <!-- Our directive will find the ng-model on the local scope -->
2 <div my-directive ng-model="object"></div>
```

The string(s) provided to the `require` option may optionally be prefixed with the following options, which change the behavior when looking up a controller:

?

If the required controller is not found on the directive provided, pass `null` to the 4th argument of the `link` function.

^

If we provide the prefix `^`, then the directive will look upwards on it's parent chain for the controller in the `require` option.

?^

Combine the behavior of the prior two options where we optionally require the directive and look up the parent chain for the directive.

no prefix

This tells the directive to locate the required controller on the named directive provided and throw an error if no controller (or directive by that name) is found.

Technically, *anything* we put in the `require` option will necessitate that we have an associated controller with it.

Angular JS Life Cycle

Before our Angular application boots, it's sits in our text editor as raw HTML. Once we've booted the app and the compile and link stages have taken place however, we're left with a life data bound application that responds on the fly to changes made by the user or the scope to which our HTML is bound. How does this magic take place and what do we need to know in order to build effective applications?

There are two main phases that take place.

compile phase

The first is called the **compile** phase. During the compile phase, Angular slurps up our initial HTML page and begins processing the directives we declared according to the directive definitions we've defined within our application's JavaScript.

Most directives, although not all, have their own templates. Each template of each directive is pieced together, finally producing a large DOM tree, often many times larger than what meets the eye of our initial HTML page. Remember, each directive can have a template, that may contain directives, which themselves may have their own templates.

Once a directive and it's child templates are have been *walked* or compiled, a function is returned for the compiled template known as the `template` function. Before the template function for a directive is returned however, we have the opportunity to modify the existing functionality to the compiled DOM tree.

At this point the DOM tree is not data bound, meaning we've got just plain HTML we can manipulate with little-to-no performance costs. During this phase built-in directives, such as `ng-repeat` and `ng-transclude` advantage of this and manipulate the DOM before it has been bound to any scope data.

`ng-repeat`, for example, will loop over the array or object it has been given building out the full representation of the DOM before passing the result off for data binding. If we're building an unordered list using `ng-repeat`, where each `` was decorated with the `ng-click` directive, this process provides us with performance that is orders of magnitude faster *especially* as our list approaches 100 elements.

The difference is that instead of cloning an ``, linking it with data, and then repeating that for each item in the loop, we're simply building out the unordered list first, and then passing the new version of the DOM (the compiled DOM) to the linking phase, the next phase in the directive life cycle.

Once a complete representation of the final DOM tree, including it's attributes, known as template attributes(`tAttr`) and it's elements(`tElement`) has been built for the directive and it's child directives, a function is returned. This compiled tree of DOM nodes is referred to as a template function hence the `t` prefix for the arguments in it's method signature.

This template function is then passed to the link function, where scope, determined by the directive definition rules of each directive in the compiled DOM tree, is applied all at once.

Once scope has been applied, we can have full access to the live representation of the DOM within the `link` and `controller` options available to us within our directive definition.

Compile (object|function)

The `compile` option can return an object or a function.

Understanding the `compile` vs `link` option is one of the more advanced topics we'll run across in Angular but does provide us with considerable context about how Angular really works.

The `compile` option by itself is not explicitly used very often. However, the `link` function is used very often and under the hood, setting the `link` option, we're actually creating a function is actually provided to the `link` function.

Often times, when we set the `compile` function, we're interested in manipulating the DOM before the directive gets placed on it with live data. Here, it is safe to manipulate HTML, add and remove elements, etc.



The `compile` option and the `link` option are mutually exclusive. If both are set, then the `compile` option will be set and be expected to return the `link` function, while the `link` option will simply be ignored.

```

1 // ...
2 compile: function(tEle, tAttrs) {
3   var tplEl = angular.element('<div>' +
4     '<h2></h2>' +
5     '</div>');
6   var h2 = tplEl.find('h2');
7   h2.attr('type', tAttrs.type);
8   h2.attr('ng-model', tAttrs.ngModel);
9   h2.val("hello");
10  tEle.replaceWith(tplEl);
11  return function(scope, ele, attrs) {
12    // The link function
13  }
14 }
15 // ...

```

TODO: cleanup - attach specified behavior to that DOM element - transform the DOM element and its children - traverses the DOM matching directives against the DOM elements - what does it mean to compile an HTML template? - attaching event listeners to the HTML to make it interactive - recursive process of attaching directives mirrors the process of compiling source code

- transforms the template DOM
- not used often since most directives don't do template transformation
- examples are `ng-repeat`(transforms template DOM)
- `ng-view` - loads the contents asynchronously
 - `tElement` - template element

- `tAttrs` - template attributes - attrs defined on this element, shared between all directive compile functions
- `transclude` - A transclude linking function: `function(scope, cloneLinkFn)`
- template instance and link instance may be different objects if the template has been cloned. Thus only do DOM transformation that are safe to do to all cloned DOM nodes within the compile function. Don't do DOM listener registration, that should be done in the linking function.

Has a return value that can either be a function or an object

- returning a `postLink` function:
 - * same thing as registering the the linking function via the `link` property when the compile function is empty.
- returning an object:
 - * with functions registered via `pre` and `post` properties - allows you to control when a linking function should be called during the linking phase(more info below)

The `compile` function deals with transforming the template DOM. ##### The `link` function deals with linking scope to the DOM.

Before scope is linked to the DOM, we may manually manipulate the DOM. In practice this is rather rare when writing custom directives but there are a few built in directives that take advantage of this functionality. Understanding the process will make help us understand how Angular actually works.

Link

Use the `link` option to create a directive that manipulates the DOM.

The `link` function is optional. If the `compile` function is defined it returns the `link` function, therefore the `compile` function will overwrite the `link` function when both are defined. If our directive is simple and doesn't require setting any additional options, we may return a function from the factory function(callback function) instead of an object. When doing so, this function is the will be the `link` function.

These two definitions of the directive are **functionally equal**:

```
1 angular.module('myApp' [])
2 .directive('myDirective', function() {
3   return {
4     pre: function(tElement, tAttrs, transclude) {
5       // executed before child elements are linked
6       // NOT safe to DOM transformations here b/c the `link` function
7       // called afterward will fail to locate the elements for linking
8     },
9     post: function(scope, iElement, iAttrs, controller) {
10      // executed after the child elements are linked
11      // IS safe to do DOM transformations here
12      // same as the link function, if the compile option here we're
13      // omitted
14    }
15  }
16});
```



```
1 angular.module('myApp' [])
2 .directive('myDirective', function() {
3   return {
4     link: function(scope, ele, attrs) {
5       return {
6         pre: function(tElement, tAttrs, transclude) {
7           // executed before child elements are linked
8           // NOT safe to DOM transformations here b/c the `link` function
9           // called afterward will fail to locate the elements for linking
10        },
11        post: function(scope, iElement, iAttrs, controller) {
12          // executed after the child elements are linked
13          // IS safe to do DOM transformations here
14          // same as the link function, if the compile option here we're
15          // omitted
16        }
17      }
18    }
19  });
20});
```

When defining the compile function instead of the link function, the link function is the second method we can provide to the object returned, known as the postLink function. In essence this describes precisely what the link function is responsible for. It is invoked after the compiled template has been linked to the scope, and therefore is responsible for setting up event listeners, watching for data changes and manipulating the live DOM.

The `link` function has control over the live data bound DOM and as such performance considerations should be taken into account. Review the section on the [life cycle of a directive](#) for more information on performance concerns when choosing to implement something in the `compile` function versus the `link` function.

The `link` function has the following signature:

```
1 link: function (scope, element, attrs) {  
2   // manipulate the DOM in here  
3 }
```

If the directive definition has been provided with the `require` option, the method signature will gain a 4th argument representing the controller or controllers of the required directive:

```
1 require 'SomeCtrl',  
2 link: function (scope, element, attrs, SomeCtrl) {  
3   // manipulate the DOM in here, with access to the controller of the  
4   // required directive  
5 }
```

If the `require` option was given an array of directives, the 4th argument will be an array representing the controller's for each of the required directives.

Let's go over each of the arguments available to the `link` function:

scope

The scope to be used by the directive for registering watches from within the directive.

element

The *instance element* is the element where directive is to be used. We should only manipulate children of this element in the `postLink` function since the children have already been linked.

attrs

The *instance attributes* are normalized (camelCased) list of attributes declared on this element shared between all directive linking functions. These are passed as a javascript object.

controller

A controller instance if at least one directive on the element defines a controller. The controller is shared among all directives, which allows the directives to use the controllers as a communication channel (public api). If multiple requires are set, then this will be an array of controller instances, rather than just a single controller.

Angular module loading

Angular modules themselves have the opportunity to configure themselves *before* the module actually bootstraps and starts to run. We can apply different sets of logic during the bootstrap phase of the app.

Configuration

Blocks of configuration get executed during the provider registration and configuration phases in the bootstrapping of the module and is the only part of the angular flow that can get modified before the app starts up.

```
1 angular.module('myApp', [])
2   .config(function($provide) {
3     // Configuration goes here
4   });

```

Throughout this book, we use methods that are simply syntactic sugar around the `.config()` function and get executed at configuration-time. For instance, when we create a factory or a directive on top of the module:

```
1 angular.module('myApp', [])
2   .factory('myFactory', function() {
3     var service = {};
4     return service;
5   })
6   .directive('myDirective', function() {
7     return {
8       template: '<button>Click me</button>'
9     }
10 })

```

These helper functions are executed at the compile-time and are *functionally equivalent* to:

```
1 angular.module('myApp', [])
2 .config(function($provide, $compileProvider) {
3   $provide.factory('myFactory', function() {
4     var service = {};
5     return service;
6   });
7   $compileProvider.directive('myDirective',
8     function() {
9       return {
10         template: '<button>Click me</button>'
11       }
12     })
13 });
```

In particular, it's also important to note that these functions are ran in-order in which they are written and registered. That is to say that we cannot inject a provider that has not yet been defined.



The *only* exception to the rule of in-order definitions is the `constant()` method. These are always placed at the beginning of all configuration blocks.

When writing configuration for a module, it's important to note that there are only a few types of objects that can get *injected* into the `.config()` function. These are only providers and constants.



If we inject any ole' service into a `.config()` function then we might *accidentally* instantiate one before we actually configure it.

The by-product of this strict requirement for configurable services is that we can only inject custom services that are build with the `provider()` syntax and cannot inject other services.

For more information on how to build with the provider syntax, head over to the [services](#) chapter.

These `.config()` blocks are how we'll custom configure our own services, such as setting API keys and custom urls.

We can also define multiple configuration blocks, which are executed in-order and allow us to focus our configuration in the different phases of the app.

```

1 angular.module('myApp', [])
2 .config(function($routeProvider) {
3   $routeProvider.when('/', {
4     controller: 'WelcomeCtrl',
5     template: 'views/welcome.html'
6   });
7 })
8 .config(function(ConnectionProvider) {
9   ConnectionProvider.setApiKey('SOME_API_KEY');
10 })

```

The `config()` function takes a single argument:

configFunction (function) The function that will be executed on module load.

Run blocks

Unlike the configuration blocks, run blocks are executed *after* the injector is created and are the first methods that are executed in any angular app.

The run blocks are the *closest thing in Angular to the main method*. The run block is code that is typically hard to unit test and is related to the general app.

Typically, these run blocks are places that we'll set up event listeners that should happen at the global scale of the app. For example, we'll use the `.run()` block to set up listeners for routing events or unauthenticated requests.

Let's say that we want to run a function that validates we have an authenticated user every time that we change our route. The only logical place to set this functionality is in the `run` method:

```

1 angular.module('myApp', [])
2 .run(function($rootScope, AuthService) {
3   $rootScope.$on('$routeChangeStart',
4     function(evt, next, current) {
5       // If the user is NOT logged in
6       if (!AuthService.userLoggedIn()) {
7         if (next.templateUrl === "login.html") {
8           // Already heading to the login route
9           // so no need to redirect
10        } else {
11          $location.path('/login');
12        }
13      }

```

```
14      })
15  });


```

The `run()` function takes a single argument:

initializeFn (function) The function that will be executed after the injector is created.

Multiple views and routing

In a single-page app, navigating from one page view to another is crucial. When apps grow more and more complex, we'll need a way to manage the screens that a user will see as they navigate their way through the app.

We can already support such management by including template code in line in the main HTML, but not only will this in-line code grow large and unmanageable, it will also make it difficult to allow other developers to join in development.

Rather than including multiple templates in the view (which we could do with the `ng-include` directive), we can break out the view into a layout and template views and only show the view we want to show based upon the current URL the user is currently accessing.

We'll break these partial templates into views to be composed inside of a layout template. AngularJS allows us to do that by declaring routes on the `$routeProvider`, a provider of the `$route` service.

Using the `$routeProvider`, we can take advantage of the browser's history navigation and enable users to bookmark and share specific pages, as it uses the current URL location in the browser.

Installation

Since 1.2.0-rc.2, `ngRoutes` was been pulled out of the core of angular into it's own module. In order to use routes in our angular app, we'll need to install and reference it in our app.

We can download it from code.angularjs.org⁵⁰ and save it in a place that we can reference it from our HTML, like `js/vendor/angular-route.js`.

We can also install it using `bower`, which will place it in our usual bower directory. For more information about bower, see the [bower chapter](#).

```
1 $ bower install --save angular-route
```

We'll need to reference this in our HTML *after* we reference angular itself.

```
1 <script src="js/vendor/angular.js"></script>
2 <script src="js/vendor/angular-route.js"></script>
```

Lastly, we'll need to reference the `ngRoute` module as a dependency in our app module:

⁵⁰<http://code.angularjs.org/>

```
1 angular.module('myApp', ['ngRoute']);
```

Layout template

To make a *layout* template, we'll need to change the HTML to tell AngularJS where to render the template. Using the `ng-view` directive in combination with the router, we can specify exactly where the rendered template of the current route should be placed in the DOM.

For instance, a layout template might look something like:

```
1 <header>
2   <h1>Header</h1>
3 </header>
4 <div class="content">
5   <div ng-view></div>
6 </div>
7 <footer>
8   <h5>Footer</h5>
9 </footer>
```

In this case, all of the rendered content will be placed in the `<div class="content">` whereas the `<header>` and `<footer>` elements will be left intact on route changes.

The `ng-view` directive is a special directive that's included with the `ngRoute` module. It's specific responsibility is to stand as a placeholder for `$route` view content.

It creates its own scope and nests the template inside of it.



The `ng-view` directive is a terminal directive at a 1000 priority. Any directives at a lower priority (most all) will **not** get run, which means all other directives on the `<div ng-view></div>` element are meaningless.

The `ngView` directive follows these specific steps:

- Anytime the `$routeChangeSuccess` event is fired, the view will update.
- If there is a template associated with the current route:
 - Create a new scope
 - Remove the last view, which cleans up the last scope
 - link the new scope and the new template
 - link the controller to the scope, if specified in the routes
 - emit the `$viewContentLoaded` event
 - run the `onload` attribute function, if provided

Routes

All application routes in AngularJS are declared using one of two methods: the `when` method and the `otherwise` method.

To create a route on a specific module/app, we'll use the `config` function.

```
1 angular.module('myApp', []).  
2   config(['$routeProvider', function($routeProvider) {  
3     }]);
```

We're using special dependency injection syntax here. For more information on this syntax, check out the dependency injection chapter.

Now, to add a specific route, the `when` method can be used. This method takes two parameters (`when(path, route)`).

A single route can be created like this:

```
1 angular.module('myApp', []).  
2   config(['$routeProvider', function($routeProvider) {  
3     $routeProvider  
4       .when('/', {  
5         templateUrl: 'views/home.html',  
6         controller: 'HomeCtrl'  
7       });  
8   }]);
```

The first parameter is the route path that is matched against the `$location.path`, the path of the current URL. Trailing or double slashes will still work. You can store parameters in the URL by starting off the name with a colon (for instance, `:name`). We'll talk about how to retrieve these parameters using the `$routeParams`.

The second parameter is the configuration object to determine exactly what to do if the route in the first parameter is matched. The configuration object properties that can be set are `controller`, `template`, `templateURL`, `resolve`, `redirectTo`, and `reloadOnSearch`.

A more complex routing scenario needs multiple routes and a catch-all that redirects a route.

```

1 angular.module('myApp', []).
2   config(['$routeProvider', function($routeProvider) {
3     $routeProvider
4       .when('/', {
5         templateUrl: 'views/home.html',
6         controller: 'HomeCtrl'
7       })
8       .when('/login', {
9         templateUrl: 'views/login.html',
10        controller: 'LoginCtrl'
11      })
12      .when('/dashboard', {
13        templateUrl: 'views/dashboard.html',
14        controller: 'DashboardCtrl',
15        resolve: {
16          user: function(SessionService) {
17            return SessionService.getCurrentUser();
18          }
19        }
20      })
21      .otherwise({
22        redirectTo: '/'
23      });
24    }]);

```

controller

```

1 controller: 'MyCtrl'
2 // or
3 controller: function($scope) {}

```

If the controller property is set on the configuration object, the controller given will be associated with the new scope of the route. If you pass a string, it associates the registered controller on the module with the new route. If you pass a function, this function will be associated with the template as the controller for the DOM element.

template

```

1 template: '<div><h2>Route</h2></div>'

```

If the template property is set in the configuration object, Angular will render the HTML template in the `ng-view` DOM element.

templateUrl

```
1 templateUrl: 'views/template_name.html'
```

If the templateUrl property is set, then your app will attempt to fetch the view using XHR (utilizing the \$templateCache). If it finds the template and can load it, Angular will render the template's contents in the ng-view DOM element.

resolve

```
1 resolve: {
2   'data': ['$http', function($http) {
3     return $http.get('/api').then(
4       function success(resp) { return response.data; }
5       function error(reason) { return false; }
6     );
7   }]
8 }
```

If the resolve property is set, then Angular will inject the elements of the *map* into the controller. If these dependencies are promises, then they will be resolved and set as a value before the controller is loaded *and* before the \$routeChangeSuccess event is fired.

The map object can be:

- key where the key is the string name of a dependency that will be injected into the controller
- factory where the factory can either be a string of the name of a service, a function whose result is injected into the controller, or a promise that is to be resolved and the value is injected into the controller.

In the example above, resolve will send the \$http request off and fill the value of 'data' as the result of the request. The data key of the map above will be injected into our controller, so it can be retrieved in the controller.

redirectTo

```
1 redirectTo: '/home'
2 // or
3 redirectTo: function(route, path, search)
```

If the redirectTo property is set with a string, then the value will change the path and trigger a route change to the new path.

If the redirectTo property is set with a function, the result of the function will be set as the value of the new path and a route change request will be triggered.

If the redirectTo property is a function, Angular will call it with one of the following parameters:

1: The route parameters extracted from the current path 2: The current path 3: The current search

reloadOnSearch

If the `reloadOnSearch` option is set to true (by default), then reload the route when `$location.search()` is changed. If you set this option to false, then the page won't reload the route if the search part of the URL changes. This is useful for nested routing or in-place pagination, etc.

Now we can set up our routes using the `when` function.

In this example, we're going to set up two routes: a home route and an inbox route. We'll also set the home route as the default route.

```

1 angular.module('MyApp', []);
2   config(['$routeProvider', function($routeProvider) {
3     $routeProvider
4       .when('/', {
5         controller: 'HomeCtrl',
6         templateUrl: 'views/home.html'
7       })
8       .when('/inbox/:name', {
9         controller: 'InboxCtrl',
10        templateUrl: 'views/inbox.html'
11      })
12      .otherwise({redirectTo: '/'});
13  }]);

```

Above, we've set up these three routes with the `when` method. If no route matches, then the `otherwise` method will be called. Using the `otherwise` method, we've set up a default route of '/'.

When the browser loads the AngularJS app, it will default to the URL set as the default route. Unless the browser is loaded with a different URL, the default is the '/' route.

\$routeParams

As mentioned above, if we start a route param with a colon (:), AngularJS will parse it out and pass it into the `$routeParams`. For instance, if we set up a route like so:

```

1 $routeProvider
2   .when('/inbox/:name', {
3     controller: 'InboxCtrl',
4     templateUrl: 'views/inbox.html'
5   })

```

then the `$routeParams` will be populated with the key of `:name` and the value of `key` will be populated with the value of the loaded URL.

If the browser loads the URL /inbox/all, then the `$routeParams` object will look like:

```
1 { name: 'all' }
```

As a reminder, to get access to these variables in the controller, we need to inject the `$routeProvider` in the controller:

```
1 app.controller('InboxCtrl', function($scope, $routeParams) {
2   // We now have access to the $routeParams here
3 });
```

\$location service

AngularJS provides a service that parses the URL in the address bar and gives you access to the route of the current path of your applications. It also gives you the ability to change paths and deal with any sort of navigation.

The `$location` service provides a nicer interface to the `window.location` JavaScript object and integrates directly with your AngularJS apps.

We'll use the `$location` service whenever we need to provide a redirect internal to our app, such as redirecting after a user signs up, changes settings, or logs in.

The `$location` service does *not* provide access to refreshing the entire page. If you need to refresh the entire page, you'll need to use the `$window.location` object (an interface to `window.location`).

path()

To get the current path, we can run the `path()` method:

```
1 $location.path(); // returns the current path
```

To change the current path and redirect to another URL in the app:

```
1 $location.path('/'); // change the path to the '/' route
```

The `path` method interacts directly with the HTML5 history API, so the user will be able to press the back button and be returned to the previous page.

replace()

If you want to redirect completely without giving the user the ability to return using the back button (it's useful for times when a redirect will occur after they are redirected), then AngularJS provides the `replace()` method:

```
1 $location.path('/home');
2 $location.replace();
3 // or
4 $location.path('/home').replace();
```

absUrl()

If you want to get full URL representation with all segments encoded, use the `absUrl()` method:

```
1 $location.absUrl()
```

hash()

To get the hash fragment in the url, we can use the `hash()` method:

```
1 $location.hash(); // return the current hash fragment
```

To change the current hash fragment, we can pass a string parameter into the method. This will return the location object.

```
1 $location.hash('movies'); // returns $location
```

host()

We can get the current host of the current url by using the `host()` method:

```
1 $location.host(); // host of the current url
```

port()

The port of the current url is available through the `port()` method:

```
1 $location.port(); // port of the current url
```

protocol()

The protocol of the current url is available through the `protocol()` method:

```
1 $location.protocol(); // protocol of the current url
```

search()

To get the search part of the current url, we can use the `search()` function:

```
1 $location.search();
```

To modify the search parameters, we can pass in parameters. This will modify the search portion of the url:

```
1 // Setting search with an object
2 $location.search({name: 'Ari', username: 'auser'});
3 // Setting search with a string
4 $location.search('name=Ari&username=auser');
```

The `search()` method takes two parameters:

- `search` (optional string or object)

The `search` parameter represents the new search params. A hash object might contain an array of values as well

- `paramValue` (optional string)

If the `search` is a string, then the `paramValue` will override a single search parameter. If the value is null, then the parameter will be removed.

url()

To get the url of the current page, we can use the `url()` method:

```
1 $location.url(); // String of the url
```

We can set and change the url using the `url()` method with parameters. This will change the path, search, and hash when called with parameters and will return `$location`.

```
1 // Set the new url
2 $location.url("/home?name=Ari#hashthing")
```

The `url()` can take two parameters:

- `url` (optional string)

This is the new url without the base prefix.

- `replace` (optional string)

The path that will be changed

Routing modes

The routing mode refers specifically to the format of the URL in the browser address bar. The default behavior of the \$location service is to route using the hashbang mode.

The routing mode determines what the URL of your site will look like.

hashbang mode

Hashbang mode is a trick that AngularJS uses to provide deep-linking capabilities to your Angular apps. In hashbang mode (the fallback for html5 mode), url paths are prepended with the # character. They do not rewrite tags and also do not require any server-side support. Hashbang mode is the default mode that AngularJS uses if it's not told otherwise.

A hashbang URL looks like:

```
1 http://yoursite.com/#!inbox/all
```

To be explicit and configure hashbang mode, it needs to be configured in the config function on an app module.

```
1 angular.module('myApp', ['ngRoute'])
2   .config(['$locationProvider', function($locationProvider) {
3     $locationProvider.html5Mode(false);
4   }]);

```

We can also configure the hashPrefix, which is, in hashbang mode the ! prefix. This is part of the fallback mechanism that Angular uses for older browsers. This character can also be configured. To configure the hashPrefix:

```
1 angular.module('myApp', ['ngRoute'])
2   .config(['$locationProvider', function($locationProvider) {
3     $locationProvider.html5Mode(false);
4     $locationProvider.hashPrefix('!');
5   }]);

```

html5 mode

The other routing mode that is supported with AngularJS is the html5Mode. The html5Mode will make your URLs look like regular URLs (except, in older browsers they will look like the hashbang URL). For instance, the same route above in html5 mode would look like:

```
1 http://yoursite.com/inbox/all
```

Inside AngularJS, the \$location service uses the HTML5's history API. This allows for our app to use the regular URL path. The \$location service will automatically fallback to using the hashbang URLs if the browser doesn't support the HTML5 history API.

One feature of the \$location service is that if a hashbang URL is loaded by a modern browser that does support the HTML5 history API then it will rewrite the URL for our users.

In html5 mode, angular will take care of rewriting links when specified in the `` tags. That is to say that as Angular compiles your app, it will rewrite the `href=""` portion depending upon the browser's capabilities.

For example with the tag: `Person`, a legacy browser's URL will be rewritten to the hashbang url equivalent: `/index.html#/person/42?all=true`. In a modern browser, it will see the URL as it was intended.

The backend server will have to support the URL rewriting on the server-side. To support html5 mode, the server will have to make sure to deliver the `index.html` page for all apps. This will ensure that our angular app will handle the route.

When writing links inside of our angular app in html5mode, we'll never want to use relative links. If you are serving your app using the root, this won't be a problem, however if you are serving in any other base route, then the angular app won't be able to handle it.

Alternatively, you can set the base URL of your app using the `<base>` tag in the HEAD section of the HTML document:

```
1 <base href="/base/url" />
```

Routing events

The \$route service fires events at different stages of the routing flow. It's possible to set up event listeners for these different routing events and react.

This functionality is useful particularly when you want to manipulate events based upon routes and are particularly useful for detecting when users are logged in and authenticated.

To listen for routing events, we'll need to setup an event listener for the events. To do this, we'll use the \$rootScope to listen for the event.

\$routeChangeStart

The \$routeChangeStart is broadcasted before the route changes. This is where the route services start resolving all of the dependencies needed for the route change to happen. This is where templates and the resolve keys are resolved.

```

1 angular.module('myApp', [])
2   .run(['$rootScope', '$location', function($rootScope, $location) {
3     $rootScope.$on('$routeChangeStart',
4       function(evt, next, current) {
5     })
6   }])

```

The `$routeChangeStart` event is fired with two parameters:

- The next url that we are attempting to navigate
- The current url that we are on before the route change

\$routeChangeSuccess

The `$routeChangeSuccess` event is broadcasted after the route dependencies have been resolved.

```

1 angular.module('myApp', [])
2   .run(['$rootScope', '$location', function($rootScope, $location) {
3     $rootScope.$on('$routeChangeSuccess', function(evt, next, previous) {
4     })
5   }])

```

The `$routeChangeSuccess` event is fired with three parameters:

- The raw angular evt object
- The current route that the user is on
- The previous route (or undefined if the current route is the first route)

\$routeChangeError

The `$routeChangeError` is broadcasted if any of the promises are rejected or fail.

```

1 angular.module('myApp', [])
2   .run(['$rootScope', '$location', function($rootScope, $location) {
3     $rootScope.$on('$routeChangeError', function(current, previous, rejection) {
4     })
5   }])

```

The `$routeChangeError` is fired with three parameters:

- The current route information
- The previous route information
- The rejection promise error

\$routeUpdate

The \$routeUpdate is broadcasted if the reloadOnSearch property has been set to false and we're reusing the same instance of a controller.

Note about indexing

Web crawlers traditionally have a hard time with fat javascript client-side apps. To support web crawlers running through the app, we will have to add a meta tag in the head. This meta tag will cause the crawler to request links with an empty *escaped_fragment* parameter so that the backend will serve back snippets of HTML.

```
1 <meta name="fragment" content="!" />
```

Other advanced routing topics

Page reloading

The \$location service will **not** reload the entire page, it will simply change the URL. If you need to cause a full page reload, then you'll have to set the location using the \$window service:

```
1 $window.location.href = "/reload/page";
```

Async location changes

If you need to use the \$location service outside of the scope life-cycle, then you'll have to use the \$apply function to propagate the changes through the app. This is because the \$location service uses the \$digest phase as the impetus to start the browser route change (this is how why routing events work).

This includes in testing the \$location service as this happens outside of the scope of the \$digest loop cycle.

Form validation

When taking input from our users, it's important to show visual feedback of their input. In the context of human relationships, form validation is just as much about giving feedback as well as getting the "right" input.

Not only does it provide positive feedback for our user, it will also semi-protect our webapp from bad or invalid input. We can only protect our backend as much as is possible with our web front-end.

Out of the box, AngularJS supports form validation with a mix of the HTML5 form validation inputs as well as with its own validation directives.

There are many form validation directives available in AngularJS. We'll talk about a few of the core validations and then we'll get into how to build your own validations. Later, in the [Our App](#) chapter, we'll touch on validations again when we build them into our app.

```
1 <form name="form">
2   <label name="email">Your email</label>
3   <input type="email" name="email" ng-model="email" placeholder="Email Address" />
4 </form>
```

AngularJS makes it pretty easy for us to handle client-side form validations without adding a lot of extra effort. Although we can't depend on client-side validations keeping our web application secure, they provide instant feedback of the state of the form.

To use form validations, we first must ensure that the form has a name associated with it, like in the above example.

All input fields can validate against some basic validations, like minimum length, maximum length, etc. These are all available on the new HTML5 attributes of a form.

It is usually a great idea to use the novalidate flag on the form element. This will prevent the browser from submitting the form.

Let's look at all the validation options we have that we can place on an input field:

Required

To validate that a form input has been filled out, simply add the html5 tag: required to the input field:

```
1 <input type="text" required />
```

Minimum length

To validate that a form input input is at least a {number} of characters, add the AngularJS directive ng-minlength="{number}" to the input field:

```
1 <input type="text" ng-minlength=5 />
```

Maximum length

To validate that a form input is equal to or less than a number of characters, add the AngularJS directive ng-maxlength="{number}":

```
1 <input type="text" ng-maxlength=20 />
```

Matches a pattern

To ensure that an input matches a regex pattern, use the AngularJS directive: `ng-pattern="/PATTERN/":`

```
1 <input type="text" ng-pattern="/a-zA-Z/" />
```

Email

To validate an email address in an input field, simply set the `input` type to `email`, like so:

```
1 <input type="email" name="email" ng-model="user.email" />
```

Number

To validate an input field has a number, set the `input` type to `number`:

```
1 <input type="number" name="email" ng-model="user.age" />
```

Url

To validate that an input represents a url, set the `input` type to `url`:

```
1 <input type="url" name="homepage" ng-model="user.facebook_url" />
```

Custom validations

AngularJS makes it very easy to add your own validations as well by using directives. For instance, let's say that we want to validate that our username is available in the database. To do this, we'll implement a directive that fires an ajax request whenever the form changes.

```

1 var app = angular.module('validationExample', []);
2
3 app.directive('ensureUnique', ['$http', function($http) {
4   return {
5     require: 'ngModel',
6     link: function(scope, ele, attrs, c) {
7       scope.$watch(attrs.ngModel, function() {
8         $http({
9           method: 'POST',
10          url: '/api/check/' + attrs.ensureUnique,
11          data: {'field': attrs.ensureUnique}
12        }).success(function(data, status, headers, cfg) {
13          c.$setValidity('unique', data.isUnique);
14        }).error(function(data, status, headers, cfg) {
15          c.$setValidity('unique', false);
16        });
17      });
18    }
19  }
20 }]);

```

Control variables in forms

AngularJS makes properties available on the containing \$scope object available to us as a result of setting a form inside the DOM. These enable us to react to the form in *realtime* (just like everything else in AngularJS). The properties that are available to us are:

Note that these properties are made available to us in the format:

```
1 formName.inputFieldName.property
```

Unmodified form

A boolean property that tells us if the user has modified the form. This is true if the user hasn't touched the form, and false if they have:

```
1 formName.inputFieldName.$pristine;
```

Modified form

A boolean property if and only if the user has actually modified the form. This is set regardless of validations on the form:

```
1  formName.inputFieldName.$dirty
```

Valid form

A boolean property that tells us that the form is valid or not. If the form is currently *valid*, then this will be true:

```
1  formName.inputFieldName.$valid
```

Invalid form

A boolean property that tells us that the form is invalid. If the form is currently *invalid*, then this will be true:

```
1  formName.inputFieldName.$invalid
```

The last two properties are particularly useful for showing or hiding DOM elements. They are also very useful when setting a class on a particular form.

Errors

Another useful property that AngularJS makes available to us is the `$error` object. This object contains all of the validations on a particular form and if they are valid or invalid. To get access to this property, use the following syntax:

```
1  formName.inputfieldName.$error
```

If a validation *fails* then this property will be true, while if it is false, then the value has passed the input field.

A little style never hurts

When AngularJS is handling a form, it adds specific classes to the form based upon their state. These classes are named similar to the properties that we can check as well.

These classes are:

```
1 .ng-pristine {}
2 .ng-dirty {}
3 .ng-valid {}
4 .ng-invalid {}
```

The correspond to their counterpart on the particular input field.

When a field is invalid, the `.ng-invalid` class will be applied to the field. This particular site sets the css class as:

```
1 input.ng-invalid {
2   border: 1px solid red;
3 }
4 input.ng-valid {
5   border: 1px solid green;
6 }
```

Putting it all together

Let's build a signup form. This signup form will include the person's name, their email, and a desired username.

Let's start by looking at what the form will look like when it's done:

The form is titled "Signup form". It contains three input fields: "Name" (labeled "Your name"), "Email" (labeled "Your email"), and "Desired username" (labeled "Username"). A "Submit" button is positioned to the right of the "Desired username" field. The entire form is contained within a light gray rectangular box.

Signup form

Play with it⁵¹

Let's start with defining the form:

⁵¹<http://jsbin.com/ePomUnI/5/edit>

```

1 <form name="signup_form" novalidate ng-submit="signupForm()">
2   <fieldset>
3     <legend>Signup</legend>
4
5     <button type="submit" class="button radius">Submit</button>
6   </fieldset>
7 </form>

```

This form's name is `signup_form` and we are going to call `signupForm()` when the form is submitted.

Now, let's add the name of the user:

```

1 <div class="row">
2   <div class="large-12 columns">
3     <label>Your name</label>
4     <input type="text"
5       placeholder="Name"
6       name="name"
7       ng-model="signup.name"
8       ng-minlength=3
9       ng-maxlength=20 required />
10    </div>
11 </div>

```

We'll discuss styling in a future chapter, but we'll include styles in this chapter as an introduction. We're using the [Foundation⁵²](#) framework in this chapter for css layouts

Next, we've added a form that has an input called `name` that is bound (by `ng-model`) to the object `signup.name` on the `$scope` object.



Don't forget to add a name to the input field. Adding a name to the input is important as that's how we'll reference the form input when showing validation messages to the user.

We also setup a few validations. These validations say we have to have a `minlength` of our name of 3 or more characters. We also impose a maximum limit of characters of 20 characters (this will be invalid at 21 characters and higher). Lastly, we're `require` that the name be filled out for the form to be valid.

Let's use the properties to show and/or hide a list of errors if the form is invalid. We'll also use the `$dirty` attribute to make sure the errors don't show up if the user hasn't touched the field:

⁵²<http://foundation.zurb.com>

```
1 <div class="row">
2   <div class="large-12 columns">
3     <label>Your name</label>
4     <input type="text"
5       placeholder="Name"
6       name="name"
7       ng-model="signup.name"
8       ng-minlength=3
9       ng-maxlength=20 required />
10    <div class="error"
11      ng-show="signup_form.name.$dirty && signup_form.name.$invalid">
12      <small class="error"
13        ng-show="signup_form.name.$error.required">
14          Your name is required.
15      </small>
16      <small class="error"
17        ng-show="signup_form.name.$error.minlength">
18          Your name is required to be at least 3 characters
19      </small>
20      <small class="error"
21        ng-show="signup_form.name.$errormaxlength">
22          Your name cannot be longer than 20 characters
23      </small>
24    </div>
25  </div>
26 </div>
```

Breaking this down, we're only going to show our errors if the form is invalid and changed, just as before. This time, we'll look through each of the validations and only show a particular DOM element if the particular validation property is invalid.

Let's look at the next set of validations, the email validation:

```
1 <div class="row">
2   <div class="large-12 columns">
3     <label>Your email</label>
4     <input type="email"
5       placeholder="Email"
6       name="email"
7       ng-model="signup.email"
8       ng-minlength=3 ng-maxlength=20 required />
9     <div class="error"
10       ng-show="signup_form.email.$dirty && signup_form.email.$invalid">
```

```
11   <small class="error"
12     ng-show="signup_form.email.$error.required">
13     Your email is required.
14   </small>
15   <small class="error"
16     ng-show="signup_form.email.$error.minlength">
17     Your email is required to be at least 3 characters
18   </small>
19   <small class="error"
20     ng-show="signup_form.email.$error.email">
21     That is not a valid email. Please input a valid email.
22   </small>
23   <small class="error"
24     ng-show="signup_form.email.$errormaxlength">
25     Your email cannot be longer than 20 characters
26   </small>
27 </div>
28 </div>
29 </div>
```

This time (with the entire form included), we're looking at the email field. Note that we set the type of the input field to *email* and added a validation error on `$error.email`. This is based off the AngularJS email validation (and the HTML5 attribute).

Finally, let's look at our last input field, the username:

```
1  <div class="large-12 columns">
2    <label>Username</label>
3    <input type="text"
4      placeholder="Desired username"
5      name="username"
6      ng-model="signup.username"
7      ng-minlength=3
8      ng-maxlength=20
9      ensure-unique="username" required />
10   <div class="error"
11     ng-show="signup_form.username.$dirty && signup_form.username.$invalid">
12     <small class="error"
13       ng-show="signup_form.username.$error.required">
14         Please input a username
15     </small>
16     <small class="error"
17       ng-show="signup_form.username.$error.minlength">
```

```

18      Your username is required to be at least 3 characters
19      </small>
20      <small class="error"
21          ng-show="signup_form.username.$errormaxlength">
22          Your username cannot be longer than 20 characters
23      </small>
24      <small class="error"
25          ng-show="signup_form.username.$error.unique">
26          That username is taken, please try another
27      </small>
28  </div>
29 </div>
```

In our last field, we're using all the same previous validations, with the exception that we've added our custom validation. This custom validation is defined using an AngularJS directive:

```

1 app.directive('ensureUnique', ['$http', function($http) {
2     return {
3         require: 'ngModel',
4         link: function(scope, ele, attrs, c) {
5             scope.$watch(attrs.ngModel, function() {
6                 $http({
7                     method: 'POST',
8                     url: '/api/check/' + attrs.ensureUnique,
9                     data: {'field': attrs.ensureUnique}
10                }).success(function(data, status, headers, cfg) {
11                    c.$setValidity('unique', data.isUnique);
12                }).error(function(data, status, headers, cfg) {
13                    c.$setValidity('unique', false);
14                });
15            });
16        }
17    }
18 }]);
```

When the form input is valid, this will make a *POST* request check to the server at /api/check/username to check if the username is available. Now, obviously since we're only talking about front-end code here, we don't have a backend to test this on, although it could easily be written.

Lastly, putting our button together, we can use the angular directive `ng-disabled` to disable and re-enable the button when the form is valid:

```

1 <button type="submit" ng-disabled="signup_form.$invalid" class="button radius">Su\
2 bmit</button>

```

Play with the example so far⁵³

As we said above, the form itself will have a `$invalid` and `valid` attributes given to us for free.

Although live validation is great, it can be abrasive to the user when they see errors pop-up while they are typing, long before they have put in a *valid* value. You can be *nicer* to your users if you show the validations either only after they have submitted the form *or* after they have moved off of the input. Let's look at both ways to do that.

Show validations after submit

To show validations only after the user has attempted to submit the form, you can capture a 'submitted' value on the scope and check for that scope to show the error.

For instance, let's take a look at the first example and only show the errors when the form has been submitted. On the `ng-show` directive on for the form input, we can add a check to see if the form has been submitted (which we will implement shortly):

```

1 <form name="signup_form" novalidate ng-submit="signupForm()" ng-controller="signu\
2 pController">
3   <fieldset>
4     <legend>Signup</legend>
5     <div class="row">
6       <div class="large-12 columns">
7         <label>Your name</label>
8         <input type="text"
9            placeholder="Name"
10           name="name"
11           ng-model="signup.name"
12           ng-minlength=3
13           ng-maxlength=20 required />
14         <div class="error"
15           ng-show="signup_form.name.$dirty && signup_form.name.$invalid && sign\
16 up_form.submitted">
17           <small class="error"
18             ng-show="signup_form.name.$error.required">
19               Your name is required.
20           </small>
21           <small class="error">

```

⁵³<http://jsbin.com/ePomUnI/5/edit>

```

22          ng-show="signup_form.name.$error minlength">
23              Your name is required to be at least 3 characters
24      </small>
25      <small class="error"
26          ng-show="signup_form.name.$error maxlength">
27              Your name cannot be longer than 20 characters
28      </small>
29  </div>
30  </div>
31 </div>
32  <button type="submit" class="button radius">Submit</button>
33 </fieldset>
34 </form>
```

Now, the error div will only show up if the `signup_form.submitted` variable has been set to true. We can implement this in the `signupForm` action, like so:

```
app.controller('signupCtrl', ['$scope', function($scope) { $scope.submitted = false; $scope.signupForm = function() { if ($scope.signup_form.$valid) { // Submit as normal } else { $scope.signup_form.submitted = true; } } }]);
```

Now, when your users try to submit the form while there is an invalid element, you can catch it and show them the appropriate errors.

[Try it out⁵⁴](#)

Show validations only after blur

If you want to retain the *real-time* nature of the error input, you can show your users the errors after they have blurred off of the input form. To do this, we like to add a small directive that will attach a new variable to the form.

The directive we like to use is the `ngFocus` directive and it looks like:

```

1 app.directive('ngFocus', [function() {
2     var FOCUS_CLASS = "ng-focused";
3     return {
4         restrict: 'A',
5         require: 'ngModel',
6         link: function(scope, element, attrs, ctrl) {
7             ctrl.$focused = false;
8             element.bind('focus', function(evt) {
9                 element.addClass(FOCUS_CLASS);
```

⁵⁴<http://jsbin.com/ePomUnI/6/edit>

```
10      scope.$apply(function() {ctrl.$focused = true;});
11  }).bind('blur', function(evt) {
12    element.removeClass(FOCUS_CLASS);
13    scope.$apply(function() {ctrl.$focused = false;});
14  });
15 }
16 ]
17 ]);
```

To implement the `ngFocus` directive, we can simply attach this directive to the input element, like so:

```
1 <input ng-class="{error: signup_form.name.$dirty && signup_form.name.$invalid}" t\
2  ype="text" placeholder="Name" name="name" ng-model="signup.name" ng-minlength=3 n\
3  g-maxlength=20 required ng-focus />
```

The `ngFocus` directive simply attaches an action to the `blur` and `focus` events on the form input and adds a class (`ng-focused`) and sets the form input field `$focused` as true. Then you can show your individual error messages depending if the form is focused or not. For instance:

```
1 <div class="error" ng-show="signup_form.name.$dirty && signup_form.name.$invalid \
2 && !signup_form.name.$focused">
```

Play with the full example⁵⁵

⁵⁵<http://jsbin.com/ePomUnI/7/edit>

Dependency injection

In general, there are only 3 ways an object can get a hold of it's dependencies:

1. It can be created internally to the dependent
2. It can be looked up or referred to as a global variable
3. It can be passed in where it's needed.

Dependency injection tackles the third way (as the other two present other difficult challenges, such as dirtying the global scope and making isolation near-impossible). Dependency injection is a design pattern that allows for the removal of hard-coded dependencies, which makes it possible to remove or change them at run-time.

This ability to modify dependencies at run-time allows us to create isolated environments ideal for testing. We can replace *real* objects in production environments with mocked ones for testing environments.

Functionally, the pattern *injects* depended-upon resources into the destination when needed by automatically looking up the dependency in advance and providing the destination for the dependency.

Simply, as we write components dependent upon other objects or libraries, we will describe it's dependencies. At runtime, an injector will create instances of the dependencies and pass them along to the *dependent* consumer.

```
1 // Great example from the angular docs
2 function SomeClass(greeter) {
3   this.greeter = greeter;
4 }
5 SomeClass.prototype.greetName = function(name) {
6   this.greeter.greet(name)
7 }
```

At runtime, the `SomeClass` doesn't care *how* it gets the `greeter` dependency, so long as it is handed to it at run-time. In order to get that `greeter` instance into `SomeClass`, the creator of `SomeClass` is responsible for passing the dependency to it when it's created.

Angular uses the `$injector` for managing look-ups and instantiation of dependencies for this reason. In fact, the `$injector` is responsible for handling all instantiations of our angular components, including our app modules, directives, controllers, etc.

When *any* of our modules boot up at runtime, the `injector` is responsible for actually instantiating the instance of the object and passing in any of its required dependencies.

For instance, this simple app declares a single module and a single controller, like so:

```
1 angular.module('myApp', [])
2 .controller('MyCtrl', [
3   function($scope, greeter) {
4     $scope.sayHello = function() {
5       greeter.greet("Hello!");
6     };
7   });
8 
```

At runtime, when angular instantiates the instance of our module, it looks up the greeter and simply passes it in naturally:

```
1 <div ng-app="myApp">
2   <div ng-controller="MyController">
3     <button ng-click="sayHello()">Hello</button>
4   </div>
5 </div>
```

Behind the scenes, the Angular process looks like:

```
1 // Load the app with the injector
2 var injector = angular.injector(['myApp']);
3 // Load the controller
4 injector.instantiate(MyController);
```

Nowhere in the above example did we describe how to find the greeter, it simply *works* as the injector takes care of finding and loading them for us.

AngularJS uses an `annotate` function to pull the properties off of the passed-in array during instantiation. You can view this function by typing the following in the chrome developer tools:

```
1 > angular.injector().annotate(GithubController);
2 ["$scope", "$http"]
```

In every angular app, the `$injector` has been at work, whether we know it or not. When we write a controller without the `[]` bracket notation or through *explicitly* setting them, the `$injector` will *infer* the dependencies based on the name of the arguments.

Annotation by inference

Angular will assume the function parameter names are the names of the dependencies, if not otherwise specified. Thus, it will call `toString()` on the function, parse and extract the function arguments and then use the `$injector` to *inject* these arguments into the instantiation of the object.

The injection process looks like:

```
1 $injector.invoke(function($scope, greeter) {});
```

Note, that this will *only* work with non-minified/non-obfuscated code as angular needs to parse the arguments in-tact.

With this javascript inference, order is **not** important as Angular will *figure* it out for us and inject the right properties in the “right” order.



JavaScript minifiers generally change function arguments to the minimum amount of characters (along with changing whitespaces, removing newlines, comments, etc) so as to reduce the ultimate filesize of the javascript files. If we do not explicitly describe the arguments, Angular will not be able to infer the arguments and thus the required injectable.

Explicit annotation

Angular provides a method for us to explicitly define the dependencies that a function needs upon invocation. This method allows for minifiers to rename the function parameters and still be able to inject the proper services into the function.

The injection process uses the `$inject` property to annotation the function. The `$inject` property of a function is an array of service names to inject as dependencies.

To use the `$inject` property method, set it on the function or name.

```
1 var aControllerFactory =
2   function aController($scope, greeter) {
3     // ... Controller
4   };
5 aControllerFactory.$inject = ['$scope', 'greeter'];
6 // Our app controller
7 angular.module('myApp')
8   .controller('MyCtrl', aControllerFactory);
9 // Invoke the controller
10 $injector.invoke(MyController);
```

With this annotation style, order is important as the `$inject` array must match the ordering of the arguments to inject. This method of injection *does* work with minification as the annotation information will be packaged with the function.

Inline annotation

The last method of annotation provided by angular out of the box is the inline annotation. This syntactic sugar works the same was as the `$inject` method of annotation from above, but allows us to inline the arguments in the function definition. Additionally it affords the ability to not use a temporary variable in the definition.

Inline annotation allows us to pass an array of arguments instead of a function when defining an angular object. The elements inside this array are the list of injectable dependencies as strings, with the last argument being the function definition of the object.

For instance:

```
1 angular.module('myApp')
2   .controller('MyCtrl', ['$scope', 'greeter',
3     function($scope, greeter) {
4
5   }]);
```

The inline annotation method *works* with minifiers as we are passing a list of strings. We often refer this method as the bracket or array notation [].

\$inject api

Although it's relatively rare that we'll need to work directly with the `$injector` knowing about the api will give us some good insight into how exactly it works.

annotate()

The `annotate()` function returns an array of service names which are to be injected into the function when instantiated. The `annotate()` function is used by the injector to determine which services will be injected into the function at invocation time.

The `annotate()` function takes a single argument:

- fn (function or array)

The `fn` argument is either given a function or an array in the bracket notation of a function definition.

The `annotate()` method returns a single array of the names of services that will be injected into the function at the time of invocation.

```
1 angular.module('myApp')
2 .controller('MyCtrl', ['$scope', 'greeter',
3   function($scope, greeter) {
4 }]);
5 var injector = angular.injector(['myApp']);
6 // ['$scope', 'greeter']
7 injector.annotate('MyCtrl');
```

Try it in your chrome debugger.

get()

The `get()` method returns an instance of the service. The `get()` method takes a single argument.

- name (string)

The `name` argument is the name of the instance to get.

`get()` returns an instance of the service by `name`.

has()

The `has()` method returns a boolean if the injector knows if a service exists or not in its registry. It takes a single argument:

- name (string)

The string is the name of the service to query the injector's registry.

It returns true if the injector knows about a particular service and false if it does not.

instantiate()

The `instantiate()` method creates a new instance of the javascript type. It takes a constructor and invokes the `new` operator with all of the arguments specified. It takes two arguments:

- Type (function)

The annotation constructor function to invoke.

- locals (object – optional)

This optional argument provides another way to pass argument names into the function when it is invoked.

The `instantiate()` method returns a new instance of `Type`.

invoke()

The `invoke()` method invokes the method and adds the method arguments from the `$injector`.

This `invoke()` method takes three arguments:

- `fn` (function)

This is the function to invoke. The arguments for the function are set with the function annotation

- `self` (object – optional)

The `self` argument allows for us to set the `this` argument for the `invoke` method

- `locals` (object – optional)

This optional argument provides another way to pass argument names into the function when it is invoked.

The `invoke()` method returns the return value by the `fn` function.

ngMin

With the three methods of defining annotations from above, it's important to note that these options all exist when defining a function. In production, however it is often less convenient to explicitly concern ourselves with order of arguments and code-bloat.

The `ngMin` tool allows us to alleviate the responsibility to need to define our dependencies explicitly. `ngMin` is a *pre-minifier* for angular apps. It walks through our Angular apps and sets up dependency injection for us.

For instance, it will turn the following:

```
1 angular.module('myApp', [])
2 .directive('myDirective',
3   function($http) {
4 })
5 .controller('IndexCtrl',
6   function($scope, $q) {
7 });
```

Into the following:

```
1 angular.module('myApp', [])
2 .directive('myDirective', [
3   '$http',
4   function ($http) {
5   }
6 ]).controller('IndexCtrl', [
7   '$scope',
8   '$q',
9   function ($scope, $q) {
10  }
11]);
```

ngMin saves us a lot of typing and cleans our source files significantly.

Installation

To install ngmin, we'll use the `npm` package manager:

```
1 $ npm install -g ngmin
```



If we're using [Grunt](#), we can install the `grunt-ngmin` grunt task. If we are using rails, we can use the ruby gem `ngmin-rails`.

Using ngmin

We can use ngMin in standalone mode at the CLI by passing two arguments: the `input.js` and the `output.js` files or via `stdio/stdout`, like so:

```
1 $ ngmin input.js output.js
2 # or
3 $ ngmin < input.js > output.js
```

Where `input.js` is our source file and `output.js` is the annotated output file.

How it works

At its core, it uses an Abstract Syntax Tree, or AST for short as it walks through the javascript source. In short, it walks through the source building an AST. With the help of `astral`, an AST tooling framework it will rebuild the source with the annotations and then dump the updated source using `escodegen`.

`ngmin` expects our angular source code to be logical declarations. If our code uses syntax similar to the code used in this book, `ngmin` will be able to parse the source and pre-minify it.

Services

Up until now, we've only concerned ourselves with how the view is tied to \$scope and how the controller manages the data. For memory and performance purposes, controllers are instantiated only when they are needed and discarded when they are not. That means that every time we switch a route or reload a view, the current controller gets garbage collected.

Services provide a method for us to keep data around for the lifetime of the app and communicate across controllers in a consistent manner.

Services are singletons objects that are instantiated only once per app (by the \$injector) and lazy-loaded (only created when necessary). They provide an interface to keep together methods that relate to a specific function.

The \$http service, for instance, is an example of an AngularJS service. It provides low-level access to the browser's XMLHttpRequest object. Rather than needing to dirty the application with low-level calls to the XMLHttpRequest object, we can simply interact with the \$http API.

```
1 // Example service that holds on to the
2 // current_user for the lifetime of the app
3 angular.module('myApp', [])
4 .factory('UserService', function($http) {
5   var current_user;
6
7   return {
8     getCurrentUser: function() {
9       return current_user;
10    },
11     setCurrentUser: function(user) {
12       current_user = user;
13     }
14   }
15 });


```

Angular comes with several built-in services with which we'll interact consistently. It will also be useful to make our own services for any decently complex application.

AngularJS makes it very easy to create our own services, simply by registering the service. Once a service is registered, the Angular compiler can reference it and load it as a dependency for runtime use. In this manner, with the name registry makes it easy to isolate application dependencies for mocks and stubbing in our tests.

Registering a service

There are several different ways to create and register a service with the `$injector`. We'll explore those later in this chapter.

The most common and flexible way to create a service uses the `angular.module` API factory:

```
1 angular.module('myApp.services', [])
  .factory('githubService', function() {
  3   var serviceInstance = {};
  4   // Our first service
  5   return serviceInstance;
 6 });


```

Although this `githubService` doesn't do anything very much interesting, it is now registered with the AngularJS app using the name `githubService` as its name.

This service factory function is responsible for generating a single object or function that becomes this service that will exist for the lifetime of the app. When the service is loaded by our angular app, it will execute this function and hold on to the returned value as the singleton service object.

The service factory function can be either a function or an array, just like how we create controllers:

```
1 // Creating the factory through using the
2 // bracket notation
3 angular.module('myApp.services', [])
  .factory('githubService', [function($http) {
 5 }]);
```

For instance, this `githubService` will require access to the `$http` service, so we'll list the `$http` service as a dependency for Angular to inject into the function.

```
1 angular.module('myApp.services', [])
  .factory('githubService', ['$http',
 3   function($http) {
 4     // Our serviceInstance now has access to
 5     // the $http service in it's function definition
 6     var serviceInstance = {};
 7     return serviceInstance;
 8 }]);
```

Now, anywhere that we need to access the Github API, we no longer need to call it through `$http`; we can call the `githubService` instead and let it handle the complexities of dealing with the remote service.

The github API exposes a list of recent events that a user has made on Github, the activity stream for the user on Github. In our service, we can create a method that accesses this API and exposes the resulting set to our API.

To expose a method on our service, we can place it as an attribute on the service object.

```

1 angular.module('myApp.services', [])
2   .factory('githubService', ['$http', function($http) {
3     var githubUrl = 'https://api.github.com';
4
5     var runUserRequest = function(username, path) {
6       // Return the promise from the $http service
7       // that calls the Github API using JSONP
8       return $http({
9         method: 'JSONP',
10        url: githubUrl + '/users/' +
11          username + '/' +
12          path + '?callback=JSON_CALLBACK'
13      });
14    }
15    // Return the service object with a single function
16    // events
17    return {
18      events: function(username) {
19        return runUserRequest(username, 'events');
20      }
21    };
22  }]);

```

The `githubService` has a single method that the components in our application can call.

Using services

To use services, we'll need to identify it as a dependency for the component where we're using it: a controller, directive, filter or another service. At runtime, angular will take care of instantiating it and resolving dependencies like normal.

To *inject* the service in the controller, we pass the name as an argument to the controller function. With the dependency listed in the controller, we can execute any of the methods we define on the service object.

```
1 angular.module('myApp', ['myApp.services'])
2 .controller('ServiceCtrl', ['$scope', 'githubService',
3   function($scope, githubService) {
4     // We can call the events function on the object
5     $scope.events =
6       githubService.events('auser');
7 }]);
```

With the new `githubService` injected into our `ServiceController`, it is now available for use just like any other service.

Let's set up our example flow to call the GitHub API for a GitHub username that we define in our view. Just as we saw in the [data binding](#) section, we'll *bind* the `username` property to the view.

```
1 <div ng-controller="ServiceController">
2   <label for="username">Type in a GitHub username</label>
3   <input type="text"
4     ng-model="username"
5     placeholder="Enter a GitHub username" />
6   <ul>
7     <li ng-repeat="event in events">
8       <!--
9         event.actor and event.repo are returned
10        by the github API. To view the raw
11        API, uncomment the next line:
12        -->
13       <!-- {{ event | json }} -->
14       {{ event.actor.login }} {{ event.repo.name }}
15     </li>
16   </ul>
17 </div>
```

Now we can *watch* for the `$scope.username` property to react to how we've changed the view, based on our bi-directional data binding.

```
1 .controller('ServiceCtrl', ['$scope', 'githubService',
2   function($scope, githubService) {
3     // Watch for changes on the username property.
4     // If there is a change, run the function
5     $scope.$watch('username', function(newUsername) {
6       // uses the $http service to call the
7       // GitHub API and returns the resulting promise
8       githubService.events(newUsername)
9         .success(function(data, status, headers) {
10           // the success function wraps
11           // the response in data
12           // so we need to call data.data to
13           // fetch the raw data
14           $scope.events = data.data;
15         })
16     });
17 }]);
```

Since we are returning the `$http` promise, we can call the `.success` method on the return as though we are calling `$http` directly.

Using `$watch` in the controller like this is not recommended. We're setting this as an example for simplicity. In production, we would wrap this into a directive and set the `$watch` function inside there instead.

In this example, you'll notice that there is a delay before the input field changes. If we don't include this delay, we'll end up calling the GitHub API for every key that is entered into the input, which is obviously not what we want.

To introduce this delay, we're using the built-in `$timeout` service. To use the `$timeout` service, we inject it into our controller just like we injected the `githubService` into the controller:

```
1 app.controller('ServiceCtrl', [
2   '$scope', '$timeout', 'githubService',
3   function($scope, $timeout, githubService) {
4 }]);
```

It's conventional to inject any Angular services before our own custom services.

Now we can use the `$timeout` service in the controller. The `$timeout` service, in this case, cancels any network requests that would otherwise be running and gives us a 350 millisecond delay between changes in the input field. In other words, if there is a delay of 350 milliseconds between keyboard strokes, we'll assume the user is done typing and we can start the GitHub request:

```
1 app.controller('ServiceCtrl', [
2   '$scope', '$timeout', 'githubService',
3   function($scope, $timeout, githubService) {
4     // The same example as above, plus
5     // the $timeout service
6     var timeout;
7     $scope.$watch('username', function(newUserName) {
8       if (newUserName) {
9         // If there is a timeout already
10        // in progress
11        if (timeout) $timeout.cancel(timeout);
12        timeout = $timeout(function() {
13          githubService.events(newUserName)
14            .success(function(data, status) {
15              $scope.events = data.data;
16            });
17        }, 350);
18      }
19    });
20  }]);
```

Since we began this app, we've only looked at how services can bundle similar functionality together. Services also are the canonical way to share data across several controllers.

For instance, if our application requires authentication from a backend service, we might want to create a `SessionsService` that handles user authentication and holds on to a token passed by the backend service. When any part of our application wants to make an authenticated request, they can use the `SessionsService` to get access to the access token.

If our application has a settings page where we set the user's GitHub username, we'll want to share the username to the other controllers in our application.

To share data across controllers, we need to add a method to our service that stores the username. Remember, the service is a singleton service that lives for the lifetime of the app, so we can store the username safely inside of it.

```
1 angular.module('myApp.services', [])
2   .factory('githubService', ['$http', function($http) {
3     var githubUrl = 'https://api.github.com',
4       githubUsername;
5
6     var runUserRequest = function(path) {
7       // Return the promise from the $http service
8       // that calls the Github API using JSONP
9       return $http({
10         method: 'JSONP',
11         url: githubUrl + '/users/' +
12             githubUsername + '/' +
13             path + '?callback=JSON_CALLBACK'
14       });
15     }
16     // Return the service object with two methods
17     // events
18     // and setUsername
19     return {
20       events: function() {
21         return runUserRequest('events');
22       },
23       setUsername: function(username) {
24         githubUsername = username;
25       }
26     };
27   }]);

```

Now, we have a `setUsername` method in our service that enables us to set the username for the current GitHub user.

In any controller in our application, we can inject the `githubService` and call `events()` without concerning ourselves with whether or not we have the right username on our scope object.

```
1 angular.module('myApp', ['myApp.services'])
2 .controller('ServiceCtrl', ['$scope', 'githubService',
3   function($scope, githubService) {
4     $scope.setUsername =
5   }]);

```

Options for creating services

While the most common method for registering a service with our angular app is through the `factory()` method, there are some other APIs we can take advantage of in situations to shorten our code or make it more terse.

The five different methods for creating services are:

- `factory()`
- `service()`
- `constant()`
- `value()`
- `provider()`

factory()

As we've seen the `factory()` method is a shorthand for creating and configuring a service.

The `factory()` function takes two arguments:

- name (string)

The name of the service to register

- getFn (function)

This is the function that will be run when angular creates this service.

```
1 angular.module('myApp')
2 .factory('myService', function() {
3   return {
4     'username': 'auser'
5   }
6 });


```

The `getFn` will be invoked **once** for the duration of the app lifecycle as it's a singleton object. It can take the same form as defining other angular services. This means it can take an array or a function that will take other injectable objects.

The `getFn` function can return anything from a primitive value, a function, or an object (similar to the `value()` function).

```
1 angular.module('myApp')
2 .factory('githubService', [
3   '$http', function($http) {
4     return {
5       getUserEvents: function(username) {
6         // ...
7       }
8     }
9   }]);

```

service()

If we want to register an instance of a service using a constructor function, we can use `service()`, which will enable us to register a constructor function for our service object.

The `service()` method takes two arguments:

- name (string)

The name to register the service instance.

- constructor (function)

The constructor function that will be called to instantiate the instance.

The `service()` function will instantiate the instance using the `new` keyword when creating the instance.

```
1 var Person = function($http) {
2   this.getName = function() {
3     return $http({
4       method: 'GET',
5       url: '/api/user'
6     });
7   };
8 };
9 angular.module('personService', Person);
```

provider

These factories are all created through the `$provide` service, which is responsible for instantiating these providers at run-time.

A provider is an object with a `$get()` method. The `$injector` calls the `$get` method to create a new instance of the service. The `$provider` exposes several different API methods for creating a service, each with a different intended use.

At the root of all of the methods for creating a service is the `provider` method. The `provider()` method is responsible for registering services in the `$providerCache`.

Technically, the `factory()` function is shorthand for creating a service through the `provider()` method by assuming the `$get()` function is the function passed in.

The two method calls are functionally equivalent and will create the same service.

```
1 angular.module('myApp')
2   .factory('myService', function() {
3     return {
4       'username': 'auser'
5     }
6   })
7 // This is equivalent to the
8 // above use of factory
9 .provider('myService', [
10   $get: function() {
11     return {
12       'username': 'auser'
13     }
14   }
15 ]);
```

Why would we ever need to use the `.provider()` method when we have the `.factory()` method created for us?

The answer lies in the ability to externally configure a service returned by the `.provider()` method using the `angular.config()` function (used as `angular.module('myApp').config()`). Unlike the other methods of creating a service we can *inject* a special attribute into the `config()` method.

Let's say we want to configure our `githubService` with our URL in advance of the application starting up:

```
1 // register the service using `provider`  
2 angular.module('myApp', [])  
3 .provider('githubService', function($http) {  
4   // default, private state  
5   var githubUrl = 'https://github.com'  
6  
7   setGithubUrl: function(url) {  
8     // change default via .config  
9     if (url) { githubUrl = url }  
10    }  
11    method: JSONP, // override me, if you want  
12  
13    $get: function($http) {  
14      self = this;  
15      return $http({  
16        method: self.method,  
17        url: githubUrl +  
18          '/events'  
19      });  
20    }  
21  });
});
```

The idea here is that by using the `.provider()` method, we have more flexibility when using our service in more than one app, when sharing our service across applications, or when sharing with the community.

With the example above, the `provider()` method creates an additional provider with the string ‘Provider’ appended on the end of it that *can* be injected into the `config()` function.

```
1 angular.module('myApp', [])  
2 .config(function(githubServiceProvider) {  
3   githubServiceProvider  
4     .setGithubUrl("git@github.com");  
5 })
```

If we want to be able to configure the service in the `config()` function, we must use `provider()` to define our service.

The `provider()` method registers a provider for a service. It takes two arguments:

- name (string)

The `name` argument is a string that will be used as the key in the `providerCache`. This will make the `name + Provider` available as the provider for the service. The `name` will be used as the name of an instance of the service.

For instance, if we define a service as `githubService`, then the provider will be available as `githubServiceProvider`.

- `aProvider` (object/function/array)

The `aProvider` argument can be a few different types.

If the `aProvider` argument is a function, then the function is called through dependency injection and is responsible for returning an object with the `$get` method.

If the `aProvider` argument is an array, then it's treated just like a function with inline dependency injection annotation and will expect the last argument to be a function which will be expected to return an object with the `$get` method.

If the `aProvider` argument is an object, then it is expected to have a `$get` method.

The `provider()` function returns an object which is a registered provider instance.

The most *raw* method of creating a service is by using the `provider()` api directly:

```
1 // Example of creating a provider directly on the
2 // module object.
3 angular.module('myApp', [])
4 .provider('UserService', [
5   favoriteColor: null,
6   setFavoriteColor: function(newColor) {
7     this.favoriteColor = newColor;
8   },
9   // the $get function can take injectables
10 $get: function($http) {
11   return {
12     'name': 'Ari',
13     getFavoriteColor: function() {
14       return this.favoriteColor || 'unknown';
15     }
16   }
17 }
18]);
```

Creating a service in this way, we must return an object that has the `$get()` function defined, otherwise it will result in an error.

We can instantiate it with the `injector` (although, it's unlikely that we'll ever do this directly as angular apps do this by [under the hood](#)):

```
1 // Get the injector
2 var injector = angular.module('myApp').injector();
3 // Invoke our service
4 injector.invoke(
5   ['$UserService', function(UserService) {
6     // UserService returns
7     // {
8     //   'name': 'Ari',
9     //   getFavoriteColor: function() {}
10    // }
11  }]);
12]);
```

Using `.provide()` is very powerful and gives us access to use our services across our applications with the ability to share.

Two other methods that are important to know when creating services are the ability to create constants in our app.

constant()

It's possible to register an existing value as a service that we can later *inject* into other parts of our app as services. For instance, let's say we want to set an `apiKey` for a backend service. We can store that constant value using `constant()`.

The `constant()` function takes two arguments:

- name (string)

This is the name to register this constant.

- value (constant value)

This is the constant value to register as the constant.

The `constant()` method returns a registered service instance.

```
1 angular.module('myApp')
2 .constant('apiKey', '123123123')
```

Now, we can *inject* this value into a configuration function just like any other service.

```
1 angular.module('myApp')
2 .controller('MyCtrl', [
3   '$scope', 'apiKey',
4   function($scope, apiKey) {
5     // We can use apiKey as a constant
6     // string as 123123123 set from above
7     $scope.apiKey = apiKey;
8 });
});
```



This is not *interceptable* by the `decorator`.

value()

If the return value of the `$get` method in our service is a constant, we don't need to define a full-blown service with a more complex method. We can simply use the `value()` function to register the service.

The `value()` method accepts two arguments:

- name (string)

This is the name that we want to register this value

- value (value)

This is the value of which we'll return as the injectable instance.

The `value()` method returns a registered service instance for the name given.

```
1 angular.module('myApp')
2 .value('apiKey', '123123123');
```

When to use value or constant

The major difference between the `value()` method and the `constant()` method is that you can *inject* a *constant* in to a config function, whereas you cannot inject a *value*.

Conversely, with *constants*, we're unable to register services objects or functions as the value.

Typically, a good rule of thumb to follow is that we should use `value()` to register a service object or function, while we should use `constant()` for configuration data.

```
1 angular.module('myApp', [])
2   .constant('apiKey', '123123123')
3   .config(function(apiKey) {
4     // The apiKey here will resolve to 123123123
5     // as we set above
6   })
7   .value('FBid', '231231231')
8   .config(function(FBid) {
9     // This will throw an error with
10    // Unknown provider: FBid
11    // because the value is not accessible by
12  });
```

decorator()

The `$provide` service provides for a way to intercept the service instance creation. Decorating our services enable us to extend services or replace it with something else entirely.

Use-cases for decorating services might include extending a service to cache external data requests to `localStorage` or can wrap a service in debugging or tracing wrappers for development purposes.

For instance, let's say that we want to provide logging calls to our previously-defined `githubService`. Rather than modifying the original service, we can *decorate* the service using a `decorator()` function.

The `decorator()` function takes two arguments:

- name (string)

The name of the service to decorate.

- `decoratorFn` (function)

The function that will be invoked at the time of the service instantiation. The function is called with `injector.invoke`, which allows us to *inject* services into it.

The `decoratorFn` will be called with a single argument: `$delegate`, which is the original service instance that we can decorate.

```
1 var githubDecorator = function($delegate) {
2   var events = function(path) {
3     var startedAt = new Date();
4     var events = $delegate.events(path);
5     // Events is a promise
6     events.always(function() {
7       console.log("Fetching events" +
8         " took " +
9         (new Date() - startedAt) + "ms");
10    });
11    return result;
12  }
13
14  return {
15    events: events
16  };
17 }
18 angular.module('myApp')
19 .config(function($provide) {
20   $provide.decorator('githubService',
21     githubDecorator);
22 }));
```

Communicating with the outside world: XHR and server-side communication

AngularJS webapps are entirely client-side applications. As you've seen, we can write AngularJS applications without integrating with a backend at all and still have a dynamic, responsive web app.

Without a backend, we are limited to only showing information that we have at load-time. What if we want to integrate our AngularJS app with information from a remote server?

AngularJS provides several methods to accomplish this.

Using \$http

We can directly call out using the built-in `$http` service. The `$http` service is simply a wrapper around the browser's raw XMLHttpRequest object.

The `$http` service is a function that takes a single argument, a configuration object that is used to generate a HTTP request. The function returns a `promise` that has two helper methods: the `success` and `error` methods.



See the [\\$http config object](#) in this chapter for details on the available options.

The most basic usage of the method looks like:

```
1 $http({  
2   method: 'GET',  
3   url: '/api/users.json'  
4 }).success(function(data, status, headers, config) {  
5   // This is called when the response is  
6   // ready  
7 }).error(function(data, status, headers, config) {  
8   // This is called when the response  
9   // comes back with an error status  
10});
```

One point to notice about the `$http` object is that it looks like we are passing in a callback method to call when the response comes back. This is not accurate as the method actually returns a promise.

Since this promise is returned, we can return the result of the `$http` method as a variable and chain other promises atop it to resolve when the HTTP has resolved.

We'll use this technique quite often when we build services so that our services can return a promise instead of requiring a callback.

```
1 var promise = $http({
2   method: 'GET',
3   url: '/api/users.json'
4});
```

Since the `$http` method returns a promise object, we can use the `then` method to handle the callback when the response is ready. If we use the `then` method, we'll get a special argument that represents the **response object**. Otherwise, we can use the `success` and `error` callbacks instead.

```
1 promise.then(function(resp) {
2   // resp is a response object
3 });
4 // OR we can use the success/error methods
5 promise.success(function(data, status, headers, config) {
6   // Handle successful responses
7 });
8 // error handling
9 promise.error(function(data, status, headers, config) {
10   // Handle non-successful responses
11});
```

If the response status code is between 200 and 299, then the response is considered to be successful and the `success` callback will be called. Otherwise, the `error` callback will be invoked.



Note that if the response results in a redirect, then the XMLHttpRequest will follow it and the `error` callback will not be called.

Note that we have the ability to use the `then()` method or the `success()` and `error()` methods on the `HttpPromise`. The difference between using the `then()` method and the convenience helpers is that the `success()` and `error()` functions contain a destructured representation of the response object, which the `then()` method receives in whole.

When we call the `$http` method, it won't actually execute until the next `$digest` loop runs. Although most of the time we'll be calling `$http` inside of a `$apply` block, we can execute the method outside of the Angular digest loop.

To execute an `$http` function outside of the `$digest` loop, we'll need to wrap it up in an `$apply` block. This will force the digest loop to run and our promises will be resolved as we expect they will.

```
1 $scope.$apply(function() {
2   $http({
3     method: 'GET',
4     url: '/api/users.json'
5   });
6 });
```

Shortcut methods

The `$http` service also provides handy methods that allow us to shorten our method calls that don't require more customization than a URL and a method name; or data with POST/PUT requests.

These shortcut methods allow us to modify the above `$http` GET call to:

```
1 // Shortened GET request
2 $http.get('/api/users.json');
```

get()

This is the shortcut method for sending a GET request.

The `get()` function accepts two parameters:

- `url` (string)

A relative or absolute URL specifying the destination of the request.

- `config` (optional object)

This optional object is an optional configuration object.

The `get()` method returns a `HttpPromise` object.

delete()

This is the shortcut method for sending a DELETE request.

The `delete()` function accepts two parameters:

- `url` (string)

A relative or absolute URL specifying the destination of the request.

- config (optional object)

This optional object is an optional configuration object.

The `delete()` method returns a `HttpPromise` object.

head()

This is the shortcut method for sending a HEAD request.

The `head()` function accepts two parameters:

- url (string)

A relative or absolute URL specifying the destination of the request.

- config (optional object)

This optional object is an optional configuration object.

The `head()` method returns a `HttpPromise` object.

jsonp()

This is the shortcut method for sending a JSONP request.

The `jsonp()` function accepts two parameters:

- url (string)

A relative or absolute URL specifying the destination of the request. In order to send the `jsonp` request, it must contain the string `JSON_CALLBACK`. For instance:

```
1 $http
2 .jsonp( "/api/users.json?callback=JSON_CALLBACK");
```

- config (optional object)

This optional object is an optional configuration object.

The `jsonp()` method returns a `HttpPromise` object.

post()

This is the shortcut method for sending a POST request.

The `post()` function accepts three parameters:

- `url` (string)

A relative or absolute URL specifying the destination of the request.

- `data` (object or string)

This is the request data content.

- `config` (optional object)

This optional object is an optional configuration object.

The `post()` method returns a `HttpPromise` object.

put()

This is the shortcut method for sending a PUT request.

The `put()` function accepts three parameters:

- `url` (string)

A relative or absolute URL specifying the destination of the request.

- `data` (object or string)

This is the request data content.

- `config` (optional object)

This optional object is an optional configuration object.

The `put()` method returns a `HttpPromise` object.

Configuration object

When we call the `$http` service as a method, we pass it a configuration object. This configuration object is used to describe how to craft the XMLHttpRequest object.

It can contain the following keys:

method (string)

The HTTP method to use to make the request. This should be one of the following: ‘GET’, ‘DELETE’, ‘HEAD’, ‘JSONP’, ‘POST’, ‘PUT’.

url (string)

The absolute *or* relative URL of the resource that is being requested.

params (map of string/object)

The map of strings or objects that will be turned into the query string after the URL. If the value is **not** a string, it will be JSONified.

```
1 // Will params into ?name=ari
2 $http({
3   params: {"name": "ari"}
4 })
```

data (string/object)

The data that will be sent as the message data.

headers (object)

This is the map of strings for functions that return strings that represent HTTP headers to send with the request. If the return value of the function is `null`, the header will **not** be sent.

xsrftokenName (string)

The name of the HTTP header to populate with the CSRF token.

xsrftokenCookieName (string)

The name of the cookie that holds the CSRF token.

transformRequest (function/array of functions)

This function or array of functions take the HTTP request body and headers and returns its transformed version. This is generally used to serialize data before it is sent to the server.

The function looks like:

```
1 function(data, headers)
```

transformResponse (function/array of functions)

The function or array of functions takes the HTTP response body and headers and returns its transformed version. This is generally used to deserialize data after it has received the data back.

The function looks like:

```
1 function(data, headers)
```

cache (boolean/Cache object)

If this is true, then the default \$http cache will cache the GET request. If this is a cache object built with \$cacheFactory, then this new cache will be used to cache the response.

timeout (number/promise)

This is a timeout in milliseconds *or* a promise that should abort the request when the promise is resolved.

withCredentials (boolean)

If this is true, then the `withCredentials` flag on the XHR request object will be set.

Normally, CORS requests will not set any cookies by default. The `withCredentials` flag will set the `Access-Control-Allow-Credentials` header which will make the request with any cookies from the remote domain in the request.

responseType (string)

The `responseType` option sets the `XMLHttpRequestResponseType` property on the request. This can be set to one of the different types of available HTTP response types:

- "" (string – default)
- "arraybuffer" (ArrayBuffer)
- "blob" (blob object)
- "document" (HTTP Document)
- "json" (JSON object parsed from a JSON string)
- "text" (string)
- "moz-blob" (Firefox to receive progress events)
- "moz-chunked-text" (streaming text)
- "moz-chunked-arraybuffer" (streaming arraybuffer)

Response object

The response object that is passed in to the `then()` method contains 4 properties:

- `data` (string/object)

The `data` represents the transformed response body (if any transformations are defined).

- `status` (number)

The HTTP status code of the response.

- `headers` (function)

This is the header getter function that takes a single parameter to get the header value for the header name. For instance, to get the header of `X-Auth-ID`, we can call the function, like so:

```
1 $http({  
2   method: 'GET',  
3   url: '/api/users.json'  
4 }).then(resp) {  
5   // Fetch the X-Auth-ID  
6   resp.headers('X-Auth-ID');  
7 }
```

- `config` (object)

This is the full, generated configuration object that was used to generate the original request.

Caching http requests

By default, the `$http` service does not cache requests in a local cache. We can enable caching per-request by passing either a boolean or a cache instance in our `$http` requests.

```
1 $http.get('/api/users.json', { cache: true })  
2 .success(function(data) {})  
3 .error(function(data) {});
```

The first time that we send this `$http` request, the `$http` service will send a GET request to `/api/users.json`. The *next* time that we send this GET request, instead of making the HTTP GET request, it will pull the request out of the cache.

By passing `true`, for this particular request, Angular will use the default cache using the `$cacheFactory`. This is created automatically for us by Angular at bootstrap time.



For more information on working directly with Angular caching, check out the [caching](#) chapter.

For more custom control of the cache that angular uses, we can pass a custom cache instead of `true` in the request.

For instance, for a Least recently used (or LRU) cache, like so:

```

1 var lru = $cacheFactory('lru', {
2   capacity: 20
3 });
4 // $http request
5 $http.get('/api/users.json', { cache: lru })
6 .success(function(data) {})
7 .error(function(data) {});
```

Now, the latest 20 unique requests will be cached. The 21st unique request will cause the least recently used request to be removed from the cache.

Now it gets a tad cumbersome to pass a custom cache every single time (even in a [service](#)). We can set a default cache for all `$http` requests across our application in the `.config()` function of our app:

```

1 angular.module('myApp')
2 .config(function($httpProvider, $cacheFactory) {
3   $httpProvider.defaults.cache =
4     $cacheFactory('lru', {
5       capacity: 20
6     });
7 });
```

Every single request will now use our custom LRU cache.

interceptors

Anytime that we want to provide global functionality on all of our requests, such as through authentication, error handling, etc., it's useful to be able to provide the ability to intercept all requests before they are both passed to the server and back from the server.

For instance, in authentication, if the server returns a response code with `401`, we likely would want to kick the user out to a login page.

Angular provides a way for us to handle responses at the global level using *interceptors*.

Interceptors, although the term sounds scary are basically *middleware* for the `$http` service that leverages the promise flow to handle injecting logic flow in the app.

Interceptors, at their core are service factories (See [services](#) for more information about services) that are registered through the `$httpProvider` by adding them to the `$httpProvider.interceptors` array.

There are four types of interceptors, two interceptors and two rejection interceptors:

- request

The request interceptor will get called with the `$http` config object. It can modify the config object or create a new one and it's responsible for returning the updated config object or a promise that resolves a new config object.

- response

The response interceptor will get called with the `$http` response object. The function can modify the response or created a new one. It's responsible for returning the response or a promise that resolves a new response object.

- requestError

This gets called when the previous interceptor throws an error or is resolved with a rejection.

- responseError

This gets called when the previous interceptor throws an error or is resolved with a rejection.

To create an interceptor, we'll use the `.factory()` method on our module and add one or more of the four methods in our service.

```
1 angular.module('myApp')
2 .factory('myInterceptor',
3   function($q) {
4     var interceptor = {
5       'request': function(config) {
6         // Successful request method
7         return config; // or $q.when(config);
8       },
9       'response': function(response) {
10        // successful response
11        return response; // or $q.when(config);
12      },
13      'requestError': function(rejection) {
14        // an error happened on the request
15        // if we can recover from the error
16        // we can return a new request
17        // or promise
18        return response; // or new promise
19        // Otherwise, we can reject the next
20        // by returning a rejection
21        // return $q.reject(rejection);
22      },
23      'responseError': function(rejection) {
24        // an error happened on the request
25        // if we can recover from the error
26        // we can return a new response
27        // or promise
28        return rejection; // or new promise
29        // Otherwise, we can reject the next
30        // by returning a rejection
31        // return $q.reject(rejection);
32      }
33    };
34
35    return interceptor;
36  });

```

We'll then need to **register** it with the `$httpProvider`:

```
1 $httpProvider.interceptors.push('myInterceptor');
```

Configuring the \$httpProvider

Using the `.config()` option, we can add certain HTTP headers to every request. This is useful for times when we want to send authentication headers along with every request or set a response type, etc.

The default headers that are sent for *every* single request are in the `$httpProvider.defaults.headers.common` object. The default headers that are sent are:

```
1 Accept: application/json, text/plain, * / *
```

We can change or augment these headers using our `.config()` function for *every* request, like so:

```
1 angular.module('myApp')
2   .config(function($httpProvider) {
3     $httpProvider.defaults.headers
4       .common['X-Requested-By'] = 'MyAngularApp';
5   });

```

We can also manipulate these defaults at the runtime using the `defaults` property of the `$http` object. For instance, to add a property for a dynamic headers, we can set the `header` property like so:

```
1 $http.defaults
2   .common['X-Auth'] = "RandomString";
```



This functionality can be achieved in either using a request transformer or for a single request by setting the `headers` option in the `$http` request.

It's also possible to manipulate the requests that are sent only on either POST or PUT requests.

The default headers that are sent for all POST requests are:

```
1 Content-Type: application/json
```

We can change or augment the POST headers in our `config()` function, like so:

```

1 angular.module('myApp')
2 .config(function($httpProvider) {
3   $httpProvider.defaults.headers
4     .post['X-Posted-By'] = 'MyAngularApp';
5 });

```

Similarly, we can do the same for all PUT requests. The default header sent for PUT requests are:

```
1 Content-Type: application/json
```

We can change or augment the PUT headers in our config() function, like so:

```

1 angular.module('myApp')
2 .config(function($httpProvider) {
3   $httpProvider.defaults.headers
4     .put['X-Posted-By'] = 'MyAngularApp';
5 });

```

Using \$resource

Angular comes with another very handy optional service called the \$resource service. This service creates a resource object that allows us to intelligently work with RESTful server-side data sources. This comes in handy when dealing with backends that support the RESTful data model out of the box.



Representational state transfer, or REST for short is a method of serving data from a backend web service intelligently. For more information about REST, check out the [Wikipedia article⁵⁶](#) on it.

The \$resource service is incredibly useful and will abstract away a lot of complexities that come with setting up a meaningful interaction with a backend server, provided it supports the RESTful data model.

The \$resource service will allow us to turn our \$http requests into simple methods like save or update. Rather than needing to be repetitive or require us to write tedious code, we can use the \$resource method to handle it for us.

⁵⁶http://en.wikipedia.org/wiki/Representational_State_Transfer

Installation

The `ngResource` module is an optional Angular module that adds support for interacting with RESTful back-end data sources. Since the `ngResource` module is not built-in to angular by default, we'll need to install it and reference it inside of our app.

We can download it from code.angularjs.org⁵⁷ and save it in a place that we can reference it from our HTML, like `js/vendor/angular-resource.js`.

We can also install it using `bower`, which will place it in our usual bower directory. For more information about bower, see the [bower chapter](#).

```
1 $ bower install --save angular-resource
```

We'll need to reference this in our HTML *after* we reference angular itself.

```
1 <script src="js/vendor/angular.js"></script>
2 <script src="js/vendor/angular-resource.js"></script>
```

Lastly, we'll need to reference the `ngResource` module as a dependency in our app module:

```
1 angular.module('myApp', ['ngResource']);
```

Now we're ready to use the `$resource` service.

Using `$resource`

The `$resource` service itself is a factory that creates a resource object. The returned `$resource` object has methods that provide a high-level API to interact with back-end servers.

```
1 var User = $resource('/api/users/:userId.json',
2   {
3     userId: '@id'
4   }
5 );
```

`$resource` returns a resource class object with a few methods for default actions. We can think of the `User` object as an interface to our RESTful back-end services.

This resource class object itself contains methods that allow us to interact indirectly with our back-end services.

By default, this object generates five methods that allow us to interact with a collection of resources or to generate an instance of a resource object. It creates 2 methods that are HTTP GET-based methods and 3 that are non-GET methods.

⁵⁷<http://code.angularjs.org/>

HTTP GET methods

The two HTTP GET methods it creates expects the following three parameters:

- params (object)

These are the parameters which will be sent with the request. These can be named parameters in the url, or they can be query parameters.

- successFn (function)

This is the callback function that will be called upon a successful HTTP response.

- errorFn (function)

This is the callback function that will be called upon a non-successful HTTP response.

get(params, successFn, errorFn)

The get method sends a GET request to the URL and expects a JSON response.

Without specifying a named parameter, such as above, the get() request is generally used to get a single resource.

```
1 // Issues a request to:  
2 // GET /api/users  
3 User.get(function(resp) {  
4     // Handle successful response here  
5 }, function(err) {  
6     // Handle error here  
7});
```

If the named parameter *is* passed into the parameters (in our example, this would be `id`), then the `get()` method will send a request to the url with the `id` in the URL:

```

1 // Issues a request to:
2 // GET /api/users/123
3 User.get({
4   id: '123'
5 }, function(resp) {
6   // Handle successful response here
7 }, function(err) {
8   // Handle error here
9 });

```

query(params, successFn, errorFn)

The query method sends a GET request to the URL and expects a collection of resource objects as a JSON response array.

```

1 // Issues a request to:
2 // GET /api/users
3 User.query(function(users) {
4   // The first user in the collection
5   var user = users[0];
6 });

```

The only major difference between the query() method and the get() method is that angular expects the query() method to return an array.

HTTP non-GET methods

The three HTTP non-GET methods it creates expects the following four parameters:

- params (object)

These are the parameters which will be sent with the request. These can be named parameters in the url, or they can be query parameters.

- postData (object)

This is the payload that is sent with the request.

- successFn (function)

This is the callback function that will be called upon a successful HTTP response.

- errorFn (function)

This is the callback function that will be called upon a non-successful HTTP response.

save(params, payload, successFn, errorFn)

The `save` method sends a POST request to the URL and uses the payload to generate the request body. The `save()` method is used to create a new resource on the server.

```
1 // Issues a request to:  
2 // POST /api/users  
3 // with the body {name: 'Ari' }  
4 User.save({}, {  
5   name: 'Ari'  
6 }, function(response) {  
7   // Handle a successful response  
8 }, function(response) {  
9   // Handle a non-successful response  
10});
```

delete(params, payload, successFn, errorFn)

The `delete` method sends a DELETE request to the URL and *can* use the payload to generate the request body. This is used to remove an instance from the server.

```
1 // Issues a request to:  
2 // DELETE /api/users  
3 User.delete({}, {  
4   id: '123'  
5 }, function(response) {  
6   // Handle a successful delete response  
7 }, function(response) {  
8   // Handle a non-successful delete response  
9 });
```

remove(params, payload, successFn, errorFn)

The `remove` method is a synonym for the `delete()` method and primarily exists because `delete` is a reserved word in javascript and can cause problems when using in Internet Explorer.

```

1 // Issues a request to:
2 // DELETE /api/users
3 User.remove({}, {
4   id: '123'
5 }, function(response) {
6   // Handle a successful remove response
7 }, function(response) {
8   // Handle a non-successful remove response
9 });

```

\$resource instances

When the methods above return data, they wrap the response in a prototype class that adds convenience methods on the instances.

The three instance methods that are available on the instance objects are:

- \$save()
- \$remove()
- \$delete()

These methods are the same as the resource counterpart except that they are called on a single resource, instead of a collection.

The three methods can be called on the instances themselves. For instance:

```

1 // Using the $save() instance methods
2 User.get({id: '123'}, function(user) {
3   user.name = 'Ari';
4   user.$save(); // Save the user
5 });
6 // This is equivalent to the collection-level
7 // resource call
8 User.save({id: '123'}, {name: 'Ari'});

```

\$resource instances are asynchronous

With all these methods, it's important to note that when they are invoked, the \$resource object immediately returns an empty reference to the data. This data is an empty *reference*, not the actual data as all these methods are executed asynchronously.

Therefore, a call to get an instance might *look* synchronous, but is actually not. In fact, it's simply a reference to data that *will* be filled in automatically by Angular when it arrives back from the server.

```
1 // $scope.user will be empty
2 $scope.user = User.get({id: '123'});
```

We can wait for the data to come back as expected using the callback method that the methods provide:

```
1 User.get({id: '123'}, function(user) {
2   $scope.user = user;
3 });
```

Additional properties

The `$resource` collections and instances have two special properties that enable us to interact with the underlying resource definitions.

- `$promise` (promise)

The `$promise` property is the original promise that is created for the `$resource`. This is particularly useful for using in conjunction with the `$routeProvider.when()` `resolve` property.

If the request is successful, the promise is resolved with the resource instance or collection object. If the request is unsuccessful, then the promise is resolved with the HTTP response object, without the `resource` property.

- `$resolved` (boolean)

The `$resolved` property is a boolean that is turned true upon the *first* server interaction (regardless if it's successful or not).

Custom `$resource` methods

Although the `$resource` service provides five methods, it is extensible enough for us to add our own custom methods to the resource object.

To create a custom method on our `$resource` class object, we can pass a third argument to the resource class that contains an object of a modified `$http` configuration object as methods.

The keys are the *name* of the method and the value is the `$http` configuration object.

```

1 var User = $resource('/api/users/:userId.json',
2 {
3   userId: '@id'
4 },
5 {
6   sendEmail: {
7     method: 'POST'
8   },
9   allInboxes: {
10    method: 'JSONP',
11    isArray: true
12  }
13 }
14 );

```

With this `User` resource, we'll now have the methods `sendEmail()` and `update()` available on the collection (`User` resource object) as well as the individual instances (as `user.$sendEmail()` and `user.$update()`).

\$resource configuration object

The `$resource` configuration object is very similar to the `$http` configuration object with a few changes.

The *value* of the object, the *action* is the name of the method on the resource object.

It can contain the following keys:

method (string)

The HTTP method to use to make the request. This should be one of the following: ‘GET’, ‘DELETE’, ‘JSONP’, ‘POST’, and ‘PUT’.

url (string)

The action-specific url override for the route of this method.

params (map of string/object)

The optional set of pre-bound parameters for this action. If any of the values are functions, they will be executed every time when a parameter values needs to be fetched for a request.

isArray (boolean)

If this isArray is set to true, then the returned object for this action will be returned as an array.

transformRequest (function/array of functions)

This function or array of functions is a transform function that takes the HTTP request body and headers and returns its transformed version. This is usually used for serialization.

```

1 var User = $resource('/api/users/:id', {
2   id: '@'
3 }, {
4   sendEmail: {
5     method: 'PUT',
6     transformRequest: function(data, headerFn) {
7       // Return modified data for the request
8       return JSON.stringify(data);
9     }
10   }
11 });

```

transformResponse (function/array of functions)

This function or array of functions is a transform function that takes an HTTP response body and headers and returns its transformed version. This is usually used for deserialization, for instance.

```

1 var User = $resource('/api/users/:id', {
2   id: '@'
3 }, {
4   sendEmail: {
5     method: 'PUT',
6     transformResponse: function(data, headerFn)
7     {
8       // Return modified data for the response
9       return JSON.parse(data);
10    }
11  }
12 });

```

cache (boolean/cache)

If the cache property is set to true, then the default \$http cache will be used to cache GET requests. If the cache property is set to an instance of a \$cacheFactory object, then it will be used to cache GET requests.

If this property is set to false, then no caching will be applied to the \$resource requests.

timeout (number/promise)

If `timeout` is set to a number, then the timeout for this request will be set to the number as milliseconds. If it's set to a promise, then it will abort the request when the promise is resolved.

withCredentials (boolean)

If this is true, then the `withCredentials` flag on the XHR request object will be set.

Normally, CORS requests will not set any cookies by default. The `withCredentials` flag will set the `Access-Control-Allow-Credentials` header which will make the request with any cookies from the remote domain in the request.

responseType (string)

The `responseType` option sets the `XMLHttpRequestResponseType` property on the request. This can be set to one of the different types of available HTTP response types:

- `""` (string – default)
- `"arraybuffer"` (`ArrayBuffer`)
- `"blob"` (`blob` object)
- `"document"` (HTTP Document)
- `"json"` (JSON object parsed from a JSON string)
- `"text"` (string)
- `"moz-blob"` (Firefox to receive progress events)
- `"moz-chunked-text"` (streaming text)
- `"moz-chunked-arraybuffer"` (streaming arraybuffer)

interceptor (object)

The `interceptor` property has two optional methods, either a `response` or a `responseError`. These *interceptors* get called with the `$http` response object, just like normal `$http` interceptors.

\$resource services

Using the `$resource` as the basis for our own custom services. Building custom services enables us greater customization and the ability to abstract the responsibility of communicating to remote services away from our controllers and views.

Finally, we *highly* recommend using `$resource` from inside a custom service object. Not only does it enable us to abstract away the responsibility of fetching remote services into a single, controllable

angular service, it disconnects this logic from our controllers, enabling us to keep them clean. Additionally, it enables us to not worry about *how* we are getting data in our controllers.

This disconnection from inside Angular objects also helps with making testing a breeze as we can stub and mock back-end calls without worrying about actually making the calls to our back end during tests.

To create a service that wraps \$resource, we will need to *inject* the \$resource service in our service object and call the methods like normal. For instance:

```

1 angular.module('myApp', ['ngResource'])
2 .factory('UserService', [
3   '$resource', function($resource) {
4
5     return $resource('/api/users/:id', {
6       id: '@'
7     }, {
8       update: {
9         method: 'PUT'
10      }
11    });
12  }]);

```

\$resource API

The \$resource service is available through the use of the \$resource() method. The method itself takes up to three parameters:

- url (string)

This is the parametrized URL string template with all of the parameters that we'll use to identify the resource prefixed by the : character. That is to say for any parameters we'll be passing in the URL, we can pass any *named* arguments in it.

```

1 $resource('/api/users/:id.:format', {
2   format: 'json',
3   id: '123'
4 })

```

Note that if the parameter before the suffix is empty (:id in our example above), then the URL will collapse into a single . character.



If we use a server that requires a port as part of the URL, for instance: `http://localhost:3000`, we **must** escape the URL pattern using `\\"`. The URL pattern for this backend server would look similar to:

```
$resource('http://localhost\\:3000/api/users/:id.json')
```

- `paramDefaults` (optional object)

The second parameter is the default values for the URL parameters that will be sent with each request. The keys in the object are matched up with the named parameters. If a key is passed that is not set as a named parameter, then it is passed as a regular query parameter.

For instance, if the url string passed has the signature of `/api/users/:id` and we set the default parameters to: `{id: '123', name: 'Ari'}`, then the resulting URL would become: `/api/users/123?name=Ari`.

We can either pass a static value here, such as we've done above by hardcoding it in the default parameter OR we can set it to pull a dynamic parameter out of the data object.

To set a dynamic parameter, we only need to prefix the value with a '@' character.

- `actions` (optional object)

The actions object is a hash with declarations of custom actions that can extend the default set of resource actions.

The keys in the object are the *names* of the custom actions while the values the [\\$http configuration object](#).

For instance, we can declare a new update action on our resource like so:

```
1 $resource('/api/users/:id.:format', {
2   format: 'json',
3   id: '123'
4 }, {
5   update: {
6     method: 'PUT'
7   }
8 })
```

Using Restangular

Although Angular on its own is powerful enough to build standalone applications where we pack all important data inside the application, in doing so we would be missing out on one of the nicest features of the framework: its ability to talk with the outside world.

In this section, we're going to talk specifically about an incredibly well-developed and well-thought-out library: **Restangular**.

The what and the why

Restangular is an Angular service specifically designed simply to fetch data from the rest of the world.

Why not use `$http` or `$resource`? Although `$http` and `$resource` are built into the framework, they carry limitations with them. Restangular takes a completely different approach to XHR and makes it a pleasant experience.

The complete list of benefits to using Restangular is available on the [README⁵⁸](#), but we'll cover a few benefits here:

Promises

Using promises makes Restangular *feel* more Angular-esque as it uses and resolves on promises. This enables us to chain together responses just as though we're using the raw `$http` method.

Explicit

Restangular has little to no magic included in the library. We don't have to guess how it works or dig up documentation on how to use it.

All HTTP methods

All *HTTP* methods are supported.

Forget URLs

While `$resource` requires us to specify the URLs we want to fetch, Restangular doesn't require us to *know* the URLs in advance, nor do we have to specify them all upfront, other than the base URL.

Nested resources

Want to use nested resources? No need to create another instance of the Restangular instance, it'll handle it for us.

One resource, not many

Unlike `$resource`, we only ever need to create one instance of the Restangular resource object.

And there is much, much more.

⁵⁸<https://github.com/mgonto/restangular>

Installation

Installing Restangular is easy – we have options. We can download the files manually (from [GitHub⁵⁹](#)) and include the file locally. If we download the file to our `js/vendor` directory, then we can include it in our HTML, like so:

```
1 <script type="text/javascript" src="js/vendor/restangular.min.js"></script>
```

We can include the JavaScript libraries hosted on [jsDelivr⁶⁰](#) in our page:

```
1 <script type="text/javascript" src="http://cdn.jsdelivr.net/restangular/latest/re\
2 stangular.js"></script>
3 <script type="text/javascript" src="http://cdn.jsdelivr.net/restangular/latest/re\
4 stangular.min.js"></script>
```

Alternatively, we can use `npm` to install Restangular if we're using `npm` in our project:

```
1 $ npm install restangular
```

Or we can use Bower to install Restangular, if we've set up Bower for our project.

```
1 $ bower install restangular
```

Note: Restangular depends on either lodash or underscore, so in order support restangular, we'll need to make sure we include one of the two.

We can either use [jsDelivr⁶¹](#) to include lodash:

```
1 <script type="text/javascript"
2   src="//cdn.jsdelivr.net/lodash/2.1.0/lodash.compat.min.js">
3 </script>
```

Or download lodash [here⁶²](#) or use bower to install.

```
1 bower install --save lodash
```

And include it as a script on the page:

⁵⁹<https://raw.github.com/jimaek/jsdelivr/master/files/restangular/latest/restangular.min.js>

⁶⁰<http://www.jsdelivr.com/>

⁶¹<http://www.jsdelivr.com/#!lodash>

⁶²<http://lodash.com/>

```
1 <script type="text/javascript" src="/js/vendor/lodash/dist/lodash.min.js"></script>
2 <
```

Also, just like any other AngularJS library, we'll need to include the `restangular` resource as a dependency for our app module object:

```
1 angular.module('myApp', ['restangular']);
```

Once we've done that, we'll be able to inject the `Restangular` service in our Angular objects:

```
1 angular.module('myApp')
2 .factory('UserService',
3   ['Restangular',
4    function(Restangular) {
5      // Now we have access to the Restangular
6      // service our service
7    }]);
8 
```

Intro to the Restangular object

To use Restangular, there are two ways we can create an object to fetch services. We can either set the base route to fetch objects from:

```
1 var User = Restangular.all('users');
```

This will set all HTTP requests that come from the Restangular service to pull from `/users`. For instance, calling `getList()` on the above object will fetch from `/users`:

```
1 var allUsers = User.getList(); // GET /users
```

It's also possible to fetch nested requests for a single object. Instead of passing only a route, we can pass a unique ID to pull from as well:

```
1 var oneUser = Restangular.one('users', 'abc123');
```

This will generate the request `/users/abc123` when calling `getList()` on it.

```
1 oneUser.get().then(function(user) {  
2   // GET /users/abc123/inboxes  
3   user.getList('inboxes');  
4 });
```

As you can see from above, Restangular is smart enough to figure out how to construct URLs based upon the methods that we are calling on the Restangular source object. Sometimes, however, it is convenient to set the URL that we're fetching, especially if our back end doesn't support pure RESTful APIs.

To set the URL for a specific request, we can pass a separate argument using the `allUrl` method:

```
1 // All URLs on searches will use  
2 // `http://google.com/` as the baseUrl  
3 var searches =  
4   Restangular.allUrl('one', 'http://google.com/');  
5 // Will send a request to GET http://google.com/  
6 searches.getList();
```

Additionally, we can set the base URL for one particular request, rather than manipulating the entire request using `oneUrl`:

```
1 var singleSearch =  
2   Restangular.oneUrl('betaSearch', 'http://beta.google.com/1');  
3  
4 // Trigger a request to GET http://beta.google.com/1  
5 singleSearch.get();
```

Using Restangular

With a good handle on the Restangular object, we can now get down to using it to make requests.

With the initial object that's returned from Restangular, we have a lot of methods that we can use to interact with our back-end API.

Let's say we've created a Restangular object that represents public discussions:

```
1 var messages = Restangular.all('messages');
```

With this object, we can get a list of all of the messages with the `getList()` method. This `getList()` method returns a collection containing methods we can call to work with the specific collection.

```

1 // allMessages is a promise that will resolve
2 // into the list of all messages
3 var allMessages = messages.getList();

```

We can also create messages using the Restangular object. To create an object, we'll use the `post()` method.

The `post` method requires a single object as a parameter and will send a (you guessed it) POST request to the *URL* we specified. We can also add `queryParameters` and `headers` to this request.

```

1 // POST to /messages
2 var newMessage = {
3   body: "Hello world"
4 };
5 messages.post(newMessage);
6 // OR we can call this on an element
7 // to create a nested resource
8 var message = Restangular.one('messages', 'abc123');
9 message.post('replies', newMessage);

```

Because Restangular returns promises, we can then call methods on the returned data on promises so that we can run a function after the promise has completed. For instance, after we update a collection, we can then refresh the collection on our scope:

```

1 messages.post(newMessage).then(function(newMsg) {
2   $scope.messages = messages.getList();
3 }, function error(reason) {
4   // An error has occurred
5 });

```

We can also remove an object from the collection. Using the `remove()` method, we can send a DELETE HTTP request to our back end. To send a delete request, we can call the `remove()` method on an object inside the collection (an element).

```

1 var message = messages.get(123);
2 message.remove(); // Send a DELETE to /messages

```

Updating and saving objects is something we'll do quite often. Traditionally, this operation is done with the HTTP method PUT. Restangular supports this out of the box through the method `put()`.

To update an object, we'll need to query the object, set our new attributes on the instance, and then call `put()` on the object to save the updated attributes in the back end.

Note, that before modifying an object, it's good practice to copy it and then modify the copied object before we save it. Restangular has its own version of copy such that it won't rebind `this` in the bundled functions. It's good practice to use `Restangular.copy()` when updating an object.

Note that before modifying an object, it's good practice to copy it and then modify the copied object before we save it. Restangular has its own version of copy such that it won't rebind `this` in the bundled functions. It's good practice to use `Restangular.copy()` when updating an object.

Now that we have experience working on instances of our collection, let's dig into nested components. Nested components are those that live underneath other components. For instance, for all of the books written by a certain author.

Restangular supports nested resources by default. In fact, we can query a particular instance from our collection for their nested resources.

```
1 var author = Restangular.one('users', 'abc123');
2 // Builds a GET to /authors/abc123/books
3 var books = author.getList('books');
```

But what about my HTTP methods?

Restangular supports, out of the box, *all* HTTP methods. It can support calling GET, PUT, POST, DELETE, HEAD, TRACE, OPTIONS, and PATCH.

```
1 author.get(); // GET /authors/abc123
2 author.getList('books'); // GET /authors/abc123/books
3 author.put(); // PUT /authors/abc123
4 author.post(); // POST /authors/abc123
5 author.remove(); // DELETE /authors/abc123
6 author.head(); // HEAD /authors/abc123
7 author.trace(); // TRACE /authors/abc123
8 author.options(); // OPTIONS /authors/abc123
9 author.patch(); // PATCH /author/abc123
```

Restangular also makes it possible to create custom HTTP methods for cases when our back-end server maps resources differently than we expect.

For instance, if we want to get the author's biography (not a RESTful resource), then we can set the URL through the `customMETHOD()` function (where METHOD is replaced by any of the following: GET, GETLIST, DELETE, POST, PUT, HEAD, OPTIONS, PATCH, TRACE).

```
1 // Maps to GET /users/abc123/biography
2 author.customGET("biography");
3 // Or customPOST with a new bio object
4 // as {body: "Ari's bio"}
5 // The two empty fields in between are the
6 // params field and any custom headers
7 author.customPOST({body: "Ari's Bio"}, // post body
8   "biography", // route
9   {},          // custom params
10  {});         // custom headers
```

Custom query parameters and headers

With all of these methods, we can send custom query parameters or custom headers.

To add custom query parameters, we'll add a JavaScript object as the second parameter to our method call. We can also add a second JavaScript object as a third parameter. Most all of the individual methods that we can call on an element take these two parameters as optional parameters.

With custom query parameters, a post method might look something like:

```
1 var queryParamObj = { role: 'admin' },
2     headerObj = { 'x-user': 'admin' };
3
4 messages.getList('accounts', queryParamObj, headerObj);
```

Restangular is incredibly simple to use and gets out of the way so that we can focus on building our app, rather than wrestling with the API.

Configuring Restangular

Restangular is highly configurable and expects us to configure it for our apps. It does come with defaults for every single property, so we don't have to configure it if we don't need to do so.

There are a few different places where we can configure a Restangular service. We can configure it globally or using a custom service.

To configure Restangular for all Restangular usages, regardless of the location it's used in, we can inject the `RestangularProvider` in a `config()` function or inject Restangular in a `run()` function.

A good rule of thumb to determine where we should configure our Restangular instances: If we need to use any other service in configuring Restangular, then we should configure it in the `run()` method, otherwise we'll keep it in the `config()` method.

Setting the baseUrl

To set the `baseUrl` for all calls to our backend API, we can use the `setBaseUrl()` method. For instance, if our API is located at `/api/v1`, rather than at the root of our server.

```

1 angular.module('myApp')
2   .config(function(RestangularProvider) {
3     RestangularProvider.setBaseUrl('/api/v1');
4   });

```

Adding element transformations

We can add any element transformations after an element has been loaded by Restangular.

Using these `elementTransformers`, we can add *custom* methods to our Restangular objects, such as when the object instance was fetched, for example.

This will get called as a callback that will enable us to update or modify the element we fetched after it's been loaded, but before we use it in our Angular objects.

```

1 angular.module('myApp')
2   .config(function(RestangularProvider) {
3     // Three parameters:
4     // the route
5     // if it's a collection - boolean (true/false)
6     // and the transformer
7     RestangularProvider.addElementTransformer('authors',
8       false, function(element) {
9         element.fetchedAt = new Date();
10        return element;
11      });
12    });

```

Setting responseInterceptors

Restangular can set `responseInterceptors`. `responseInterceptors` are useful for when we want to translate the response we get back from the server. For instance, if our server comes back with the data tucked away in a nested object, we can use a `responseInterceptor` to dig it out.

This is called after every response we get back from the backend. It will get called with the following parameters:

- `data` - data retrieved from the server
- `operation` - the HTTP method used

- what - the model that's requested
- url - the relative URL that's being requested
- response - the full server response, including headers
- deferred - the promise for the request

```
1 angular.module('myApp')
2   .config(function(RestangularProvider) {
3     RestangularProvider.setResponseInterceptor(
4       function(data, operation, what) {
5         if (operation == 'getList') {
6           return data[what];
7         }
8         return data;
9       });
10    });

```

Using requestInterceptors

Restangular supports the other side of the operation as well: We can work with the data we are going to send to the server before we ever actually send any data back to the server in the first place.

requestInterceptors are useful for times when we'll need to run manipulations on the object before sending it to the server. For instance, we can't call directly to mongo with an `_id` field, so we have to remove it before it gets sent to the backend if we're in a PUT operation.

To set a requestInterceptor, we can use the method `setRequestInterceptor()`. The `setRequestInterceptor()` method will be called with the following parameters:

- element - the element we're sending to the server
- operation - the HTTP method used
- what - the model that's being requested
- url - the relative URL that's being requested

```
1 angular.module('myApp')
2   .config(function(RestangularProvider) {
3     RestangularProvider.setRequestInterceptor(
4       function(elem, operation, what) {
5         if (operation === 'put') {
6           elem._id = undefined;
7           return elem;
8         }
9         return elem;
10      });
11    });

```

Custom fields

Restangular also supports setting custom Restangular fields. This is important for times when we are connecting not to a back-end server, but to a back-end database, such as MongoDB where the id field doesn't map to an id.

For instance, when connecting to MongoDB, the id field actually maps to _id.\$oid.

```
1 angular.module('myApp')
2   .config(function(RestangularProvider) {
3     RestangularProvider.setRestangularFields({
4       id: '_id.$oid'
5     });
6   });
7 });


```

Catching errors with errorInterceptors

It's also possible to set error Interceptors, for those times when we want to catch an error from within Restangular. Using the errorInterceptor gives us the ability to halt the flow of the error down to our app.

If we return false from the errorInterceptor, then the flow of the promise will end and our app will never need to deal with handling errors.

This is a good time to handle dealing with authentication failures, for instance. If any request comes back with a 401, we can use the errorInterceptor to catch it and handle redirecting the user to the login page.

```

1 angular.module('myApp')
2   .config(function(RestangularProvider) {
3     RestangularProvider.setErrorInterceptor(
4       function(resp) {
5         displayError();
6         return false; // stop the promise chain
7     });
8   });
9 });

```

Setting parentless

If we're fetching a resource that is **not** nested underneath other nested resources, we can tell Restangular to not build the nested URL structure using the `setParentless` configuration property field.

```

1 angular.module('myApp')
2   .config(function(RestangularProvider) {
3     RestangularProvider.setParentless([
4       'cars'
5     ]);
6   });

```

The `setParentless()` configuration function can take two different types of parameters:

boolean

If set to true, then all resources will be considered 'parentless' and no URL will be nested

array

Only the resources identified by the string in the array will be considered parentless.

Custom Restangular services

Finally, we *highly* recommend using Restangular from inside a custom service object. This is particularly useful, as we can configure Restangular on a per-service level using a service as well as disconnecting the logic to talk to our back end from within our controllers/directives and enabling our services to handle talking to them directly.

This disconnection from inside Angular objects also helps with making testing a breeze as we can stub and mock back-end calls without worrying about actually making the calls to our back end during tests.

To create a service that wraps Restangular, we simply need to *inject* the Restangular service in our factory and call the methods like normal. Inside this factory, we can create custom configurations by using the `withConfig()` function.

For instance:

```
1 angular.module('myApp', ['restangular'])
2 .factory('MessageService', [
3   'Restangular', function(Restangular) {
4     var restAngular =
5       Restangular.withConfig(function(Configurer) {
6         Configurer.setBaseUrl('/api/v2/messages');
7       });
8
9     var _messageService = restAngular.all('messages');
10
11    return {
12      getMessages: function() {
13        return _messageService.getList();
14      }
15    }
16  }]);
17});
```

XHR in practice

Cross-origin and same-origin policy

Web browsers nearly universally prevent web pages from fetching and executing scripts on foreign domains.

The *same-origin* policy specifically permits scripts running on pages which are originating from the same site, including their scheme, hostname, and port number. Any other interactions with scripts originating from off-site has heavy restrictions placed on them to run.

The Cross Origin Resource Sharing (or CORS, for short) is often a source of headaches for fetching data over XHR and dealing with foreign sources.

Fortunately, there are several ways for us to get data exposed by external data sources into our app. We'll look at two of these methods and mention a third (that requires a bit more backend support):

- JSONP
- CORS
- Server proxies

JSONP

JSONP is a trick around getting past the browser security issues that are present when we're trying to request data from a foreign domain. In order to work, with JSONP, the server must be able to support the method.

JSONP works by issuing a GET request using a `<script>` tag, instead of using XHR requests. The JSONP technique creates a `<script>` tag and place it in the DOM. When it shows up in the DOM, the browser takes over and requests the script referenced in the `src` tag.

When the server returns the request, it surrounds the response with a javascript function invocation that corresponds to a request our javascript knows about.

Angular provides a helper for JSONP requests using the `$http` service. The `jsonp` method of request through the `$http` service looks like:

```
1 $http
2 .jsonp("https://api.github.com?callback=JSON_CALLBACK")
3 .success(function(data) {
4   // Data
5 });
```

When we make this call, Angular will place a `<script>` tag on the DOM that might look something like:

```
1 <script src="https://api.github.com?callback=angular.callbacks._0"
2 type="text/javascript"></script>
```

Notice that the `JSON_CALLBACK` is replaced with a custom angular function that's created by angular specifically for this request.

When the data comes back from the JSONP-enabled server, it will be wrapped in the function `angular.callbacks._0`.

In this case, the github server will return some JSON wrapped in the callback and it's response might look like:

```
1 // shortened for brevity
2 angular.callbacks._0({
3   "meta": {
4     "X-RateLimit-Limit": "60",
5     "status": 200,
6   },
7   "data": {
8     "current_user_url": "https://api.github.com/user"
9   }
10 })
```

When the special angular function is called, the `$http` promise will be resolved as angular takes care of the rest.



When we write our own backend servers to support JSONP, we'll need to ensure that when we respond, we wrap the data inside the function that's given by the request with `callback`.

When using JSONP, we need to be aware of the potential security risks. First, we're opening up our server to allow a back-end server to be able to call any javascript in our app.

A foreign site that we do not control can change their script anytime (or a malicious cracker could) exposing our site for vulnerabilities. The server or a middleman could potentially send extra javascript logic back into our page that could expose private user data.

Additionally, JSONP can only be used to send GET requests since we're setting a GET request in the `<script>` tag. Additionally, it's tough to manage errors on a script tag. We should use JSONP sparingly and only with servers we trust and control.

Using CORS

In recent years, the W3C has created the CORS specification, or Cross Origin Resource Sharing policy. CORS was written to replace the JSONP *hack* in a standard way.

The CORS specification is simply an extension to the standard XMLHttpRequest object which allows javascript to make cross-domain XHR calls. It does this by *preflighting* a request to the server to effectively ask for permission to send the request.

This *preflight* gives the receiving server the ability to accept or reject any request from all or a select server or set of servers. This being said, this means that both the client app and the server app need to coordinate to provide data to the client server.

The CORS specification was written with the intention of abstracting away a lot of the details from the client-side developer so that it appears as though the request is made the same as a same-origin request.

Configuration

To use CORS within angular, we need to tell Angular that we're using CORS. We'll use the `.config()` method on our angular app module to set two options.

First, we need to tell Angular to use the `XDomain` as well as we must remove the `X-Requested-With` header from all of our requests.



The `X-Requested-With` header has been removed from the common header defaults, but it's a good idea to ensure it's been removed anyway.

```
1 angular.module('myApp')
2   .config(function($httpProvider) {
3     $httpProvider.defaults.useXDomain = true;
4     delete $httpProvider.defaults.headers
5       .common['X-Requested-With'];
6   });

```

Now we're ready to make CORS requests.

Server CORS requirements

Although we will not dive into server-side CORS set-up in this chapter (we do in the [server communication](#) chapter), it's important that the server we're working with supports CORS.

A server supporting CORS must respond to requests with several access control headers:

- Access-Control-Allow-Origin

The value of this header can either echo the Origin request header or it must be a * to allow any and all requests from any origin.

- Access-Control-Allow-Credentials (optional)

By default, CORS requests are **not** made with cookies. If the server includes this header, then we can send cookies along with our request. We do this in angular by setting the `withCredentials` option to true.

If we set the `withCredentials` option in our `$http` request to true, but the server does not respond with this header, then the request will fail and visa versa.

The back-end server must also be able to handle OPTIONS request methods.

There are two types of CORS requests, simple and not-simple requests.

Simple requests

Requests are considered simple if they match one of the HTTP methods:

- HEAD
- GET
- POST

And if they are made with one or many of the following HTTP headers, and no others:

- Accept
- Accept-Language
- Content-Language
- Last-Event-ID
- Content-Type
 - application/x-www-form-urlencoded
 - multipart/form-data
 - text/plain

These are categorized as *simple* requests because the browser can make these types of requests without the use of CORS. Simple requests do NOT require any special type of communication between the server and the client.

A simple CORS request using the \$http service looks like any other request:

```
1 $http
2 .get("https://api.github.com")
3 .success(function(data) {
4   // Data
5 });
```

Non-simple requests

None-simple requests are considered any requests that are made that violate the requirements for the simple requests. If we want to support PUT or DELETE methods, or if we want to set the specific type of content-type in our requests, then we're going to call a non-simple request.

Although this doesn't look any different to us as client-side developers, the request is handled differently by the browser.

The browser actually sends two requests, a preflight and then the request. First, the browser issues a preflight request where the server requests permission to make the request. If the permissions have been granted, then the browser can make the actual request.

The browser takes care of handling the CORS request transparently.

Similar to the simple request, the browser will add the Origin header to both of the requests (preflight and the actual request).

Preflight request

The preflight request is made as an OPTIONS request. It will contain a few headers in the request:

- Access-Control-Request-Method

This is the HTTP method of the actual request. This is always included in the request.

- Access-Control-Request-Headers (optional)

This is a comma delimited list of non-simple headers that are included in the request.

The server should accept the request, check if the HTTP method and the headers are valid. If they are, then the server should respond with the following headers:

- Access-Control-Allow-Origin

The value of this header can either echo the Origin request header or it must be a * to allow any and all requests from any origin.

- Access-Control-Allow-Methods

The list of allowed HTTP methods. This is helpful as we can cache this request in the client and we don't have to constantly ask for preflights in future requests.

- Access-Control-Allow-Headers

If the Access-Control-Request-Headers is set, then the server should respond with this header.

The server is expected to respond with a 200 if the request is acceptable. If it is, then the second request will be made.



CORS is **not** a security mechanism, it is simply a standard that modern browsers implement. It's still our responsibility to set up security in our app.

Non-simple requests look exactly like regular requests inside Angular:

```
1 $http
2 .delete("https://api.github.com/api/users/1")
3 .success(function(data) {
4   // Data
5 });


```

Server-side proxies

The simplest method for making requests to **any** server, however is to simply use a backend server on the same domain (or on a remote server with CORS setup) as a proxy for remote resources.

Rather than our client-side app making requests to foreign resources, we can simply use our own local server to make requests for our client-side app and responding.

This enables older browsers to be able to make requests (CORS is implemented in modern browsers only), doesn't require a second request for non-simple CORS requests and uses the standard browser-level security as it was intended on being used.

In order to use a server-side proxy, we'll need to set up a local server to handle our requests, which takes care of sending the actual requests.

For more information about setting up a server-side component, read the [Server communication](#) chapter.

Working with json

JSON, or the JavaScript Object Notation is a data-interchange format that looks a lot like a javascript object. In fact, it resolves to one when loaded by JavaScript and Angular will resolve any requests that respond with a JavaScript object in JSON format to a corresponding object for our angular app.

For instance, if we have the following JSON returned by our server:

```
1 [  
2     {"msg": "This is the first msg", state: 1},  
3     {"msg": "This is the second msg", state: 2},  
4     {"msg": "This is the third msg", state: 1},  
5     {"msg": "This is the fourth msg", state: 3}  
6 ]
```

When our angular app receives this data over \$http, we can simply reference the data as a JavaScript object:

```
1 $http.get('/v1/messages.json')  
2     .success(function(data, status) {  
3         $scope.first_msg = data[0].msg;  
4         $scope.first_state = data[0].state;  
5     });
```

Working with xml

Although Angular transparently handles JSON objects that are handed back from the server, we can handle other data types as well.

For instance, if our server hands us back XML instead of JSON, we'll need to massage the data into a JavaScript object.

Luckily, there are some great open-source libraries available as well as some built-in browser parsers to parse XML into JavaScript objects for us.

We'll use the X2JS library, a fantastic open-source library available [here⁶³](#).

First, we'll need to make sure we *install* the X2JS library. We can use bower to install the library for us:

⁶³<https://code.google.com/p/x2js/>

```
1 $ bower install x2js
```

And then reference it from the googlecode.com or from our bower components:

```
1 <script type="text/javascript" src="https://x2js.googlecode.com/hg/xml2json.js"><\n2 /script>\n3 <!-- OR -->\n4 <script type="text/javascript" src="bower_components/xml2json/xml2json.js"></script>\n5 
```

Starting off with our lightweight XML parser, we'll create a factory that simply will parse the XML in the DOM for us.

```
1 angular.factory('xmlParser', function() {\n2     var x2js = new X2JS();\n3     return {\n4         xml2json: x2js.xml2json,\n5         json2xml: x2js.json2xml_str\n6     }\n7 });\n\n
```

With this lightweight parsing factory, we can create an `transformResponse` to parse our XML within our `$http` requests, such as:

```
1 angular.factory('Data', ['$http, 'xmlParser',\n2     function($http, xmlParser) {\n3         $http.get('/api/msgs.xml', {\n4             transformResponse:\n5                 function(data) {\n6                     return xmlParser.xml2json(data);\n7                 }\n8             })\n9     });\n\n
```

And now our response will come back as a JSON object and we can use the response just as though the server returned JSON.

Authentication with AngularJS

In most serious web applications, there usually are protected resources that we want to keep secret from the general public and only give access to authenticated users whom we know and trust. These resources can be anything from paid material to administration ability.

Regardless of what we are protecting, the methods that we can use to protect our resources will be similar.

In this section, describing *how* to implement server-side authentication is out of scope. However, instead we'll focus on describing *what* our server-side back-end needs to do to feature our front-end view.

Then we'll dive right into discussing how to provide client-side authentication protection and discuss potential edge-cases for the process.

Server-side requirements

First and foremost, we must take the time to secure our server-side API. As we are dealing with uncompiled code being sent by a potentially untrusted source, we cannot count on our uses to all be genuine users.

There are generally *two* ways we can handle securing our client-side app:

Server-side rendered views

If we're serving our site through a server-side server that controls all of the HTML, we can use traditional authentication methods and only send the HTML that our client-side needs and is authenticated for from the server.

Pure client-side authentication

If we want to be able to build our client-side and server-side as different components and allow the deployment of these components to be naturally separated in production deployment. We'll need to secure our client-side authentication using the server-side API, but not reliant on it's authentication.

We're going to implement client-side authentication through token authentication. Our server-side needs to be able to provide our client app with a auth token.

The token itself should be a random string of numbers and letters that the server-side can associate to a particular user session.



`uuid` libraries are generally good candidates for generating tokens.

That is, when a user logs into our site, instead of sending a user id or using any identifiable information, our server-side will need to generate this random token and create an association with the user session to this token.

This token will be expected to be sent with every single client-side request so that we can look up the user by this random string of characters on each one.

Our server-side will then need to send the proper status codes for the particular events that we get back, if they are valid or not so that our client-side can react based on the received status code.

For instance, for all unauthenticated requests, we'll want our server-side to send back a 401 response status code.

The following table is a short list of response status codes that we'll deal with in this section to describe how to handle authentication.

Code	Meaning
200	Everything is good
401	Unauthenticated request
403	Forbidden request
404	Page not found
500	Server error

When we encounter these these status codes, we can react accordingly.

The data flow looks like this:

1. Unauthenticated user visits our site. 2a. The user tries to access a protected resource and is redirected to the login page. 2b. The user visits the login page manually.
2. The user enters their login id (username or email) and their password and our angular app makes a POST request to our server with the user's data.
3. Our server looks at the login id and password and determines if they are a match. 5a. If the login id matches the password, a generated unique token is sent back with the request with a 200 response code. 5b. If the login id does not match the password, the request is responded to with a status code of 401.

For authenticated users (who takes the 5a request path from above):

1. A protected resource path (such as their own account page) will be requested. 2a. If the user has not logged in, our app will redirect the user to the login page. 2b. If the user is logged in, the request will be made with the unique user token for the session.
2. The server will validate this token and return the appropriate data back based on the request.

Client-side authentication

From the above section, we've outlined a few behaviors that our authentication scheme will need to handle:

- Redirection on unauthorized page requests
- Capture non-200 responses back and act accordingly on any XHR request
- Keeping track of the user through the page session

To handle redirection on unauthorized page requests, such as trying to access a protected resource we'll need to determine how to define a protected resource vs. a public one.

There are several ways to handle defining routes as public vs. non-public routes.

Protected resources from API

If we're protecting routes that need to operate on protected API calls, meaning that in order to load the page we make a protected resource API call where the server can respond with a `401` response code, then we can simply rely on `$http` interceptors to handle the work for us.

To create an `$http` interceptor who's responsibility it is to respond to unauthenticated API requests, we'll need to create one that handles responses.

We'll set up our `$http` response interceptor inside of a `.config()` block inside our app where we *inject* the `$httpProvider`:

```
1 angular.module('myApp', [])
2 .config(function($httpProvider) {
3   // Build our interceptor here
4 });


```

This interceptor will handle both responses and responseErrors and will be called on all requests.

```
1 angular.module('myApp', [])
2 .config(function($httpProvider) {
3   // Build our interceptor here
4   var interceptor =
5     function($q, $rootScope, Auth) {
6       return {
7         'response': function(resp) {
8           if (resp.config.url == '/api/login') {
9             // Assuming our API server response
10            // with the following data:
```

```
11      // { token: "AUTH_TOKEN" }
12      Auth.setToken(resp.data.token);
13    }
14  },
15  'responseError': function(rejection) {
16    // Handle errors
17    switch(rejection.status) {
18      case 401:
19        if (rejection.config.url!=='api/login')
20          // If we're not on the login page
21          $rootScope
22            .$broadcast('auth:loginRequired');
23        break;
24
25      case 403:
26        $rootScope
27          .$broadcast('auth:forbidden');
28        break;
29
30      case 404:
31        $rootScope
32          .$broadcast('page:notFound');
33        break;
34
35      case 500:
36        $rootScope
37          .$broadcast('server:error');
38        break;
39    }
40
41    return $q.reject(rejection);
42  }
43}
44}
45});
```

This *auth* interceptor handles a few of the server-side response codes that we can possibly receive from our server on any given request. The response interceptor takes any 401 response and \$broadcasts an event down the app from the \$rootScope so that any child scope (all scopes) can handle the event.

Additionally, this interceptor will *save* the token for any successful 200 request to our /api/login route.

To actually implement this interceptor for our requests, we'll need to tell the `$httpProvider` to include it in its interceptor chain:

```
1 angular.module('myApp', [])
2 .config(function($httpProvider) {
3   // Build our interceptor here
4   var interceptor =
5     function($q, $rootScope, Auth) {
6       // ...
7     }
8   // Integrate the interceptor in the
9   // request/response chain for $http
10  $httpProvider
11    .interceptors.push(interceptor);
12 });

});
```



For more information on `$http` interceptors, check out the [\\$http interceptors section](#).

Protected resources by route definition

If we always want paths protected and/or no API calls are being made that need to protect the route, then we will need monitor our routes and ensure that we have a logged in user for the routes we are interested in protecting.

In order to monitor our routes, we'll need to set up an event listener on the `$routeChangeStart` event. This event is fired when we the route properties start to resolve, but before we've actually changed the route.



Combined with the previous method, this approach will be more secure. If we don't check for status code, our users can still make requests

We'll set our listener to focus on this event and check to see if the route itself is defined to be exposed to the current user.

First, let's define some access roles for our application. We can do this by setting a constant in our app so that we can check against these roles on each route.

```
1 angular.module('myApp', ['ngRoute'])
2 .constant('ACCESS_LEVELS', {
3   pub: 1,
4   user: 2
5});
```

By setting the ACCESS_LEVELS as a constant, we can *inject* it into both `.config()` and `.run()` blocks and can use it all over our application.

Now, let's use these constants to define access levels for each of our defined routes:

```
1 angular.module('myApp', ['ngRoute'])
2 .config(function($routeProvider, ACCESS_LEVELS) {
3   $routeProvider
4     .when('/', {
5       controller: 'MainCtrl',
6       templateUrl: 'views/main.html',
7       access_level: ACCESS_LEVELS.pub
8     })
9     .when('/account', {
10       controller: 'AccountCtrl',
11       templateUrl: 'views/account.html',
12       access_level: ACCESS_LEVELS.user
13     })
14     .otherwise({
15       redirectTo: '/'
16     })
17 });
```

Each of our routes above defines their own `access_level` which we can check for both user authorization (if necessary) and that the current user has the appropriate user level for the route.

At this point, there will be a user with one of two states:

- Unauthenticated anonymous user
- Authenticated known user

For us to authenticate a user, we'll need to create a service that holds on to user level. We'll also let our service work with the local browser cookie store so we can expect that our user when logged in, we can expect that they'll stay logged in while the session is still good.

This small service simply includes some helper functions on top of the user object:

```
1 angular.module('myApp.services', [])
2 .factory('Auth',
3   function($cookieStore, ACCESS_LEVELS) {
4     var _user = $cookieStore.get('user');
5
6     var setUser = function(user) {
7       if (!user.role || user.role < 0) {
8         user.role = ACCESS_LEVELS.pub;
9       }
10    _user = user;
11    $cookieStore.put('user', _user);
12  }
13
14  return {
15    isAuthorized: function(lvl) {
16      return _user.role >= lvl;
17    },
18    setUser: setUser,
19    isLoggedIn: function() {
20      return _user ? true : false;
21    },
22    getUser: function() {
23      return _user;
24    },
25    getId: function() {
26      return _user ? _user._id : null;
27    },
28    getToken: function() {
29      return _user ? _user.token : '';
30    },
31    logout: function() {
32      $cookieStore.remove('user');
33      _user = null;
34    }
35  }
36});
```

Now we simply can check on our \$routeChangeStart event if the user is authenticated and logged in.

```
1 angular.module('myApp')
2 .run(function($rootScope, $location, Auth) {
3   // Set a watch on the $routeChangeStart
4   $rootScope.$on('$routeChangeStart',
5     function(evt, next, curr) {
6
7     if (!Auth.isAuthorized(next.access_level)) {
8       if (Auth.isLoggedIn()) {
9         // The user is logged in, but does not
10        // have permissions to view the view
11        $location.path('/');
12      } else {
13        $location.path('/login');
14      }
15    }
16  })
17});
```

Talking to MongoDB

If we don't have a custom back-end, it's also possible to talk directly to a database that exposes a RESTful interface.

Instead of having to build a backend, we can talk directly to mongo.



In this example, we're using [mongolab](#)⁶⁴, a SaaS service that offers managed mongoDB instances.

In order to talk to MongoDB, we'll need to set up a few custom configurations for our Restangular objects.



Note that these configurations will change the global Restangular objects. If we want to nest this configuration for a single database, then we'll need to create a factory to nest the custom Restangular object.

First, we'll need to set our API key. Since this won't change across the entire app, we suggest creating this as a constant.

⁶⁴<https://mongolab.com>

```
1 angular.module('myApp', ['restangular'])
2 .constant('apiKey', 'YOUR_API_KEY');
```

Setting this as a constant, we can then inject it into other parts of our application. We'll set up our configuration in `config()` block on our module.

Using mongolab, we'll set our `baseUrl` to the API endpoint through mongolab:

```
1 // ...
2 .config(function(RestangularProvider, apiKey) {
3   RestangularProvider
4     .setBaseUrl('https://api.mongolab.com/api/1/databases/YOURDB/collections');
5});
```

Next, **every** single request we make to our backend database will require our api key. Restangular makes it easy to add using the `setDefaultRequestParams()` method:

```
1 // ...
2 .config(function(RestangularProvider, apiKey) {
3   // ...
4   RestangularProvider
5     .setDefaultRequestParams({
6       apiKey: apiKey
7     });
8});
```

Next, we'll need to update the Restangular field to map the custom id field provided by mongoDB as `_id.$oid` to the Restangular `id` field. This is simple to do using the `setRestangularFields()` function

```
1 // ...
2 .config(function(RestangularProvider, apiKey) {
3   // ...
4   RestangularProvider.setRestangularFields({
5     id: '_id.$oid'
6   });
7});
```

Lastly, we'll need to overwrite the `_id` field set by mongo when we're updating a record. This is because mongo won't let us 'rewrite' the `_id` field, so we can use Restangular to 'fake' setting the `_id` field. Since Restangular will call the route to update the element, we don't need to worry about the object not getting rewritten.

```
1 // ...
2 .config(function(RestangularProvider, apiKey) {
3     // ...
4     RestangularProvider.setRestangularFields({
5         id: '_id.$oid'
6     });
7 });
```

For completeness-sake, the entire config block is here:

```
1 angular.module('myApp', ['restangular'])
2 .constant('apiKey', 'API_KEY')
3 .config(function(RestangularProvider, apiKey) {
4     RestangularProvider.setBaseUrl(
5         'https://api.mongolab.com/api/1/databases/YOURDB/collections');
6     RestangularProvider.setDefaultRequestParams({
7         apiKey: apiKey
8     })
9     RestangularProvider.setRestangularFields({
10        id: '_id.$oid'
11    });
12
13     RestangularProvider.setRequestInterceptor(
14         function(elem, operation, what) {
15
16             if (operation === 'put') {
17                 elem._id = undefined;
18                 return elem;
19             }
20             return elem;
21         })
22     });
});
```

Promises

Angular's event system (we talk about this in-depth in the [under the hood](#)) provides a lot of power to our Angular apps. One of the most powerful features that it enables is automatic resolution of promises.

What's a promise?

Promises are a method of resolving a value or not in an asynchronous manner. Promises are objects that represent the return value or a thrown exception that a function may eventually provide. Promises are incredibly useful in dealing with remote objects and we can think of them as a proxy for them.

Traditionally, javascript uses closures, or callbacks to respond with meaningful data that is not available in a synchronous time, such as XHR requests after a page has loaded. Rather than depending upon a callback to be fired, we can interact with the data instead as though it has already returned.

Callbacks have worked for a long time, but the developer suffers a lot when using them. They have no consistency in callbacks, no guaranteed call, steal code flow when callbacks depend upon other callbacks, and in general make debugging incredibly difficult. At every step of the way, we have to deal with *explicitly* handling errors.

Instead of fire-and-hopefully-get-a-callback-run when executing asynchronous methods, promises offer a different abstraction. They return a promise object.

For example, in traditional callback code we might have a method that sends a message from one user to one of their friends.

```
1 // Sample callback code
2 User.get(fromId, {
3   success: function(err, user) {
4     if (err) return {error: err};
5     user.friends.find(toId, function(err, friend) {
6       if (err) return {error: err};
7       user.sendMessage(friend, message, callback);
8     });
9   },
10  failure: function(err) {
11    return {error: err}
```

```
12    }
13});
```

This callback-pyramid is already getting out of hand and we haven't included any robust error handling code either. Additionally, we'll need to know the order in which the arguments are called from within our callback.

The promised-based version of the previous code might look something closer to:

```
1 User.get(fromId)
2 .then(function(user) {
3   return user.friends.find(toId);
4 }, function(err) {
5   // We couldn't find the user
6 })
7 .then(function(friend) {
8   return user.sendMessage(friend, message);
9 }, function(err) {
10  // The user's friend resulted in an error
11 })
12 .then(function(success) {
13  // user was sent the message
14 }, function(err) {
15  // An error occurred
16});
```

Not only is this more readable, but it is also much easier to grok. We can also guarantee that the callback will resolve to a single value, rather than dealing with the callback interface.

Notice, in the first example we have to handle errors differently to non-errors and we'll need to make sure that using callbacks to handle errors will all need to implement the same API (usually with `(err, data)` being the usual method signature).

In the second example, we handle the success and error in the same way. Our resultant object will receive the error in the usual manner. Additionally, since the promise API is specific about resolving or rejecting promises, we don't have to worry about our methods implementing a different method signature.

Why promises?

Although a bi-product of promises is the fact that we can get away from *callback hell*, the point of promises is to make asynchronous functions look more like synchronous ones. With synchronous functions, we can capture both return values and exception values as expected.

We can capture errors at any point of the process and bypass future code that relies upon the error of that process. We do all this without thinking about the benefits of this synchronous code as it's the nature of it.

Thus, the point of using promises is to regain the ability to do functional composition and error bubbling while maintaining the ability of the code to run asynchronously.

Promises are *first-class* objects and carry with them a few guarantees:

- Only one resolve or reject will ever be called
 - resolve will be called with a single fulfillment value
 - reject will only be called with a single rejection reason
- If the promise has been resolved or rejected, any handlers depending upon them will still be called
- Handlers will always be called asynchronously

Additionally, we can also chain promises and allow the code to process as it will normally run. Exceptions from one promise bubble up through the entire promise chain.

They are always asynchronous, so we can use them in the flow of our code without worry that they will block the rest of the app.

Promises in Angular

Angular's event-loop gives angular the unique ability to resolve promises in its `$rootScope.$evalAsync` stage (see [under the hood](#) for more detail on the run loop). The promises will sit inert until the `$digest` run loop finishes.

This allows for Angular to turn the results of a promise into the view without any extra work. It enables us to assign the result of an XHR call directly to a property on a `$scope` object and think nothing of it.

Let's build an example that will return a list of open pull requests for Angular JS from github.

[Play with it⁶⁵](#)

⁶⁵<http://jsbin.com/UfotanA/3/edit>

```
1 <h1>Open Pull Requests for Angular JS</h1>
2
3 <ul ng-controller="DashboardController">
4   <li ng-repeat="pr in pullRequests">
5     {{ pr.title }}
6   </li>
7 </ul>
```

If we have a service that returns a promise (covered in-depth in the [services](#) chapter), we can simply place the promise in the view and expect that Angular will resolve it for us:

```
1 angular.module('myApp', [])
2 .controller('DashboardCtrl', [
3   '$scope', 'UserService',
4   function($scope, UserService) {
5     // UserService's getFriends() method
6     // returns a promise
7     User.getFriends(123)
8     .then(function(data) {
9       $scope.friends = data.data;
10    });
11  }]);
12]);
```

Note that automatically unwrapped promises is no longer the default in angular. We can re-enable this form if we really want to by setting the option to true in a `.config()` function:

```
1 .config(function($parseProvider) {
2   $parseProvider.unwrapPromises(true) ;
3 });
```

When the asynchronous call to `getPullRequests` returns, the `$scope.pullRequests` value will automatically update the view.

How to create a promise

In order to create a promise in Angular, we'll use the built-in `$q` service. The `$q` service provides a few methods in its deferred API.

First, we'll need to inject the `$q` service into our object where we want to use it.

```
1 angular.module('myApp', [])
2 .factory('GithubService', ['$q', function($q) {
3   // Now we have access to the $q library
4 }]);
```

To create a deferred object, we'll call the method `defer()`:

```
1 var deferred = $q.defer();
```

The `deferred` object exposes three methods and the single `promise` property that can be used in dealing with the promise.

- `resolve(value)`

The `resolve` function will resolve the deferred promise with the value.

```
1 deferred.resolve({name: "Ari", username: "@auser"});
```

- `reject(reason)`

This will *reject* the deferred promise with a reason. This is equivalent to resolving a promise with a rejection

```
1 deferred.reject("Can't update user");
2 // Equivalent to
3 deferred.resolve($q.reject("Can't update user"));
```

- `notify(value)`

This will respond with the status of a promises execution.

For example, if we want to provide a status back from the promise, we can use the `notify()` function to deliver it back.

Let's say that we have several long-running requests that are to be made from a single promise. We can call the `notify` function to send back a notification of progress:

```
1 .factory('GithubService', function($q, $http) {
2   // get events from repo
3   var getEventsFromRepo = function() {
4     // task
5   }
6   var service = {
7     makeMultipleRequests: function(repos) {
8       var d = $q.defer(),
9         percentComplete = 0,
10        output = [];
11       for (var i = 0; i < repos.length; i++) {
12         output.push(getEventsFromRepo(repos[i]));
13         percentComplete = (i+1)/repos.length * 100;
14         d.notify(percentComplete);
15       }
16     }
17     d.resolve(output);
18   }
19   return d.promise;
20 }
21 }
22 return service;
23});
```

With this `makeMultipleRequests()` function on our `GithubService` object, we will get a progress notification every time that a repo has been fetched and processed.

We can use this notification in our usage of the promise by adding a third function call to the promise usage. For instance:

```
1 .controller('HomeCtrl',
2 function($scope, GithubService) {
3   GithubService.makeMultipleRequests([
4     'auser/beehive', 'angular/angular.js'
5   ])
6   .then(function(result) {
7     // Handle the result
8   }, function(err) {
9     // Error occurred
10  }, function(percentComplete) {
11    $scope.progress = percentComplete;
12  });
13});
```

We can get access to the promise as a property on the deferred object:

```
1 deferred.promise
```

A full example of creating a function that responds with a promise might look similar to the following method on the GithubService as mentioned above.

```
1 angular.module('myApp', [])
2 .factory('GithubService', [
3   '$q', '$http',
4   function($q, $http) {
5     var getPullRequests = function() {
6       var deferred = $q.defer();
7       // Get list of open angular js pull requests from github
8       $http.get('https://api.github.com/repos/angular/angular.js/pulls')
9       .success(function(data) {
10         deferred.resolve(data);
11       })
12       .error(function(reason) {
13         deferred.reject(reason);
14       })
15       return deferred.promise;
16     }
17
18     return { // return factory object
19       getPullRequests: getPullRequests
20     };
21   }]);

```

Now we can use the promise API to interact with the getPullRequests() promise.

[View full example⁶⁶](#)

In the case of the above service, we can interact with the promise in two different ways.

- then(successFn, errFn, notifyFn)

Regardless of the success or failure of the promise, then will call either the successFn or the errFn asynchronously as soon as the result is available. The callbacks are **always** called with a single argument: the result or the rejection reason.

⁶⁶<http://jsbin.com/UfotanA/3/edit>

The `notifyFn` callback may be called zero or more times to provide a progress status indication *before* the promise is resolved or rejected.

The `then()` method always returns a new promise which is either resolved or rejected through the return value of the `successFn` or the `errFn`. It also notifies through the `notifyFn`.

- `catch(errFn)`

This is simply a helper function that allows for us to replace the `err` callback with `.catch(function(reason) {})`:

```
1 $http.get('/user/' + id + '/friends')
2 .catch(function(reason) {
3   deferred.reject(reason);
4 });


```

- `finally(callback)`

This allows you to observe the fulfillment or rejection of a promise, but without modifying the result value. This is useful for when we need to release a resource or run some clean-up regardless of the success/error of the promise.

We cannot call this directly due to `finally` being a reserved word in IE javascript. To use `finally`, we have to call it like:

```
1 promise['finally'](function() {});
```

Angular's `$q` deferred objects are chainable in that even `then` returns a promise. As soon as the promise is resolved, the promise returned by `then` is resolved or rejected.



These promise chains are how Angular can support `$http`'s interceptors.

The `$q` service is similar to the original Kris Kowal's `Q` library:

1. `$q` is integrated with the angular `$rootScope` model, so resolutions and rejections happen quickly inside of angular
2. `$q` promises are integrated with angular's templating engine which means that any promises that are found in the views will be resolved/rejected in the view
3. `$q` is tiny and doesn't contain the full functionality of the `Q` library

chaining requests

The `then` method returns a new derived promise after the initial promise is resolved. This give us the unique ability to attach yet another `then` on the result of the initial `then` method.

```
1 // A service that responds with a promise
2 GithubService.then(function(data) {
3   var events = [];
4   for (var i = 0; i < data.length; i++) {
5     events.push(data[i].events);
6   }
7   return events;
8 }).then(function(events) {
9   $scope.events = events;
10});
```

In this example we can create this *chain* of execution that allows us to interrupt the flow of the application based upon more functionality we can attach to different results.

This allows us to pause or defer resolutions of promises at any point during the chain of execution.



This is how the \$http service implements request/response interceptors.

The \$q library comes with several different useful methods.

all(promises)

If we have multiple promises that we want to combine into a single promise, then we can use the \$q.all(promises) method to combine them all into a single promise. This method takes a single argument:

- promises (array or object of promises)

Promises as an array or hash of promises

The all() method returns a single promise that will be resolved with an array or hash of values. Each value will correspond to the promises at the same index/key in the promises hash. If **any** of the promises are resolved with a rejection, then the resulting promise will be rejected as well.

defer()

The defer() method creates a deferred object. It takes no parameters. It returns a new instance of a single deferred object.

reject(reason)

This will create a promise that is resolved as rejected with a specific reason. This is specifically designed to give us access to forwarding rejection in a chain of promises. This is akin to throw in javascript. In the same sense that we can *catch* an exception in javascript and we can forward the rejection, we'll need to rethrow the error. We can do this with \$q.reject(reason).

This method takes a single parameter:

- reason (constant, string, exception, object)

The reasons for the rejection.

This `reject()` method returns a promise that has already been resolved as rejected with the reason.

when(value)

The `when()` function wraps an object that might be a value then-able promise into a \$q promise. This allows for us to deal with an object that may or may not be a promise.

The `when()` function takes a single parameter:

- value

This is the value or a promise

The `when()` function returns a promise that can be then used like any other promise.

Server communication

One of the most power components of Angular is it's ability to communicate with a server-side back-end. Regardless of the back-end that we're using, angular can likely talk to it through an API.

In this chapter, we're going to focus on two types of back-ends, custom server-side back-ends that we'll develop and server-less back-ends with back-ends as a service.

Custom Server-side

In this section, we're going to focus on the process of building a custom server-side application in NodeJS. Although we're going to focus on building this server app in Node, we can build our back-end in any server-side language that supports HTTP API routes.



If you're a Ruby on Rails developer, we've written a book specifically on demonstrating how to use Rails. Check [Riding Rails with AngularJS⁶⁷](#).

To start our node-backed app, we're going to need to have NodeJS installed.

Install NodeJS

NodeJS is a server-side platform built on the Chrome JavaScript runtime. It is an event-driven, non-blocking lightweight JavaScript runtime that enables us to write JavaScript on the server.

To install NodeJS, we can head to [nodejs.org⁶⁸](#) and click on the big *Install* button. It will detect and download the appropriate installer for our platform.

i> If for some reason it downloads the wrong package, no problem, we can click on the *Downloads* button and manually select the appropriate package.

Now we can run the installer and let it run. Once it's complete, we'll have the two packages available on the command-line:

- node
- npm

node is the node binary that we will call to run our node app while npm is the node package manager which we'll use to install node libraries.

⁶⁷<http://www.fullstack.io/edu/angular/rails/>

⁶⁸<http://nodejs.org/>

Install Express

We're going to use a web application framework called `express.js` which will give us some syntactic sugar around dealing with HTTP. It allows us to only work with the functionality of our web app as opposed to needing to deal with the nitty gritty details of node's HTTP server.

Its features are extensive, such as providing a clean routing syntax, dynamic middleware, tons of open-source packages built for it, and much more. Additionally, it is used by many well-known companies in production.

To install express, we'll use the `npm` binary:

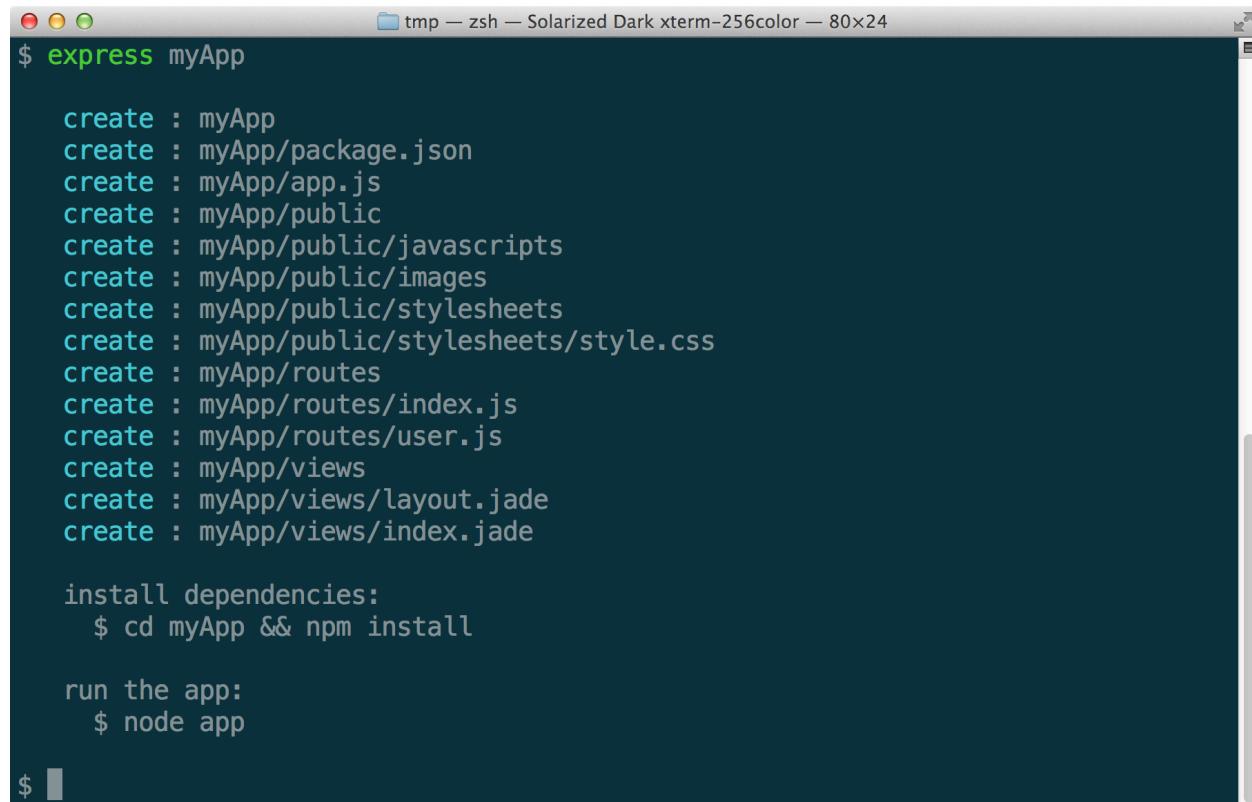
```
1 $ npm install -g express
```

i> We'll use the `-g` flag to install the package *globally*. We can leave this off if we don't want to install it globally where it will be installed in the directory `node_modules/` in the current directory. We recommend installing it globally, however.

Now, we can use the express generator to generate our express app.

```
1 $ express myApp
```

This will generate a very basic express app with a set of requirements and a loosely opinionated directory structure.



```
$ express myApp

create : myApp
create : myApp/package.json
create : myApp/app.js
create : myApp/public
create : myApp/public/javascripts
create : myApp/public/images
create : myApp/public/stylesheets
create : myApp/public/stylesheets/style.css
create : myApp/routes
create : myApp/routes/index.js
create : myApp/routes/user.js
create : myApp/views
create : myApp/views/layout.jade
create : myApp/views/index.jade

install dependencies:
$ cd myApp && npm install

run the app:
$ node app

$
```

Running express generator

To run our app, we'll need to install the basic dependencies locally using `npm` again. This time, we'll use it to install the dependencies set in the `package.json` locally.

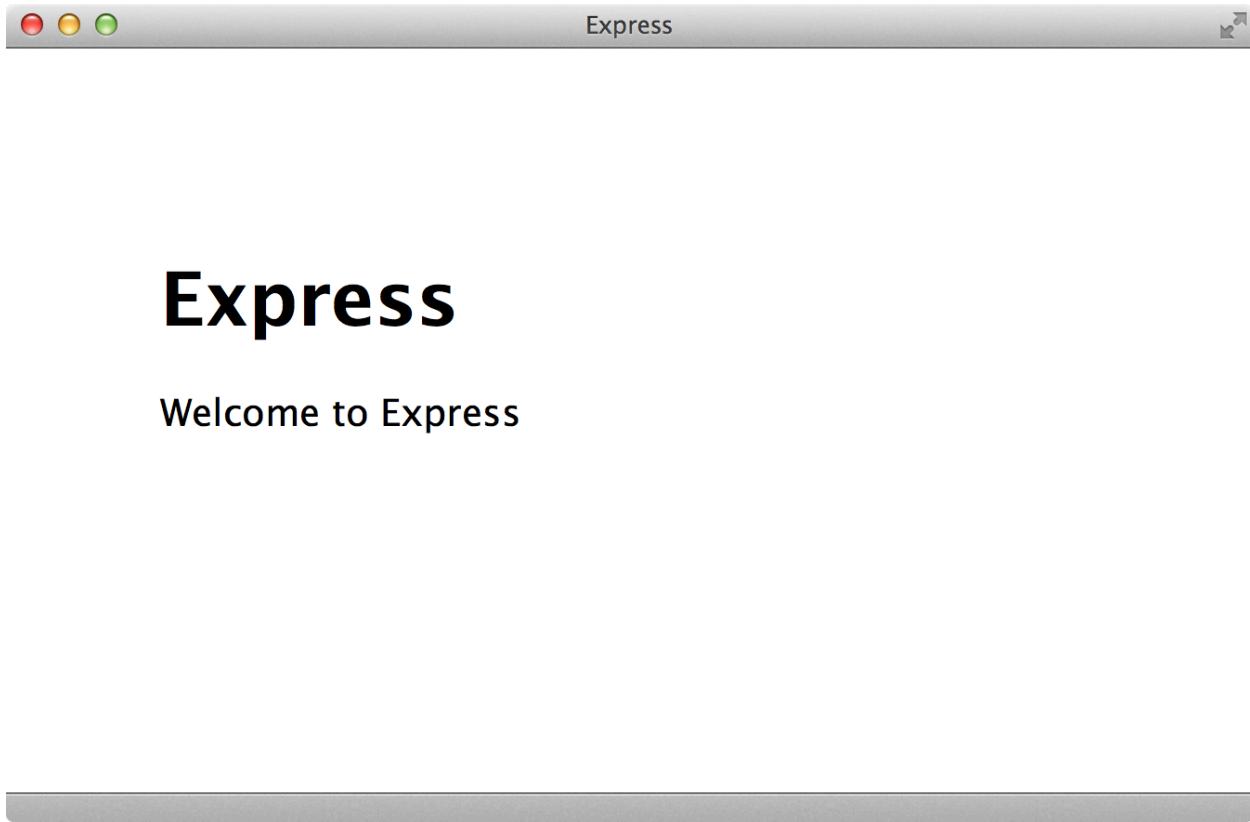
```
1 $ cd myApp && npm install -d
```



The `-d` flag tells `npm` to install the dependencies locally. This is overly explicit and we can leave the `-d` off as it's set to install dependencies locally by default.

Now, let's run the app to make sure everything is working as we expect it should. We can do this simply by running it with the `node` binary:

```
1 $ node app.js
```



Running express

The default page of our express app shows us that the express app has been generated if we open the url `http://localhost:3000` in our web browser.

Everytime that we make a change to our `app.js` file, we'll need to stop that server and restart it. When in development, this can be a cumbersome process, so we suggest using the `nodemon` server instead of `node.js`.

To install `nodemon`, we'll use `npm` again:

```
1 $ npm install --save-dev nodemon
```



The `--save-dev` flag tells `npm` to save the package in the `devDependencies` section in the `package.json`. We recommend using this practice as it helps when introducing multiple developers to a team in ensuring the whole team has the *right* dependencies for the codebase:

Instead of using `node app.js` to start the app, we can replace it with:

```
1 $ nodemon app.js
```

Everytime that we make a chance to the `app.js` file and save it, `nodemon` will restart our node app automatically.

The app starts out in `app.js`. There are two important components to notice in the `app.js` file. The static path where static files are served from and the routes which are and how routes are resolved:

```
1 // ...
2 app.use(express.methodOverride());
3 app.use(app.router);
4 // Line 1
5 app.use(express.static(path.join(__dirname, 'public')));
6 // ...
7 app.get('/', routes.index);
8 app.get('/users', user.list);
9 // ...
```

The first line, with the `express.static()` call tells node to look in the `public/` directory for any files it may find that match the requested route. For instance, if the requested route is `'/users'`, then it would look for a file called `'users'`.

The second set of lines: `app.get()` match the routes for the cases that static files are not matched in the `public/` directory.

To work with our angular app, we'll make two modifications to the generated `app.js`.

First, swap the lines of the `express.static()` and the `app.router` line, like so:

```
1 // ...
2 app.use(express.methodOverride());
3 // Moved this line above the next line
4 app.use(express.static(path.join(__dirname, 'public')));
5 app.use(app.router);
6 // ...
```

Although this is not strictly necessary, it will help support HTML5Mode later and will tell express to prefer the files in the `public/` directory above those defined in our app.

Second, remove the line pointing to the `'/'` route:

```
1 // ...
2 // app.get('/', routes.index); // remove this line
3 app.get('/users', users.list);
4 // ...
```

Now we can write our angular app like normal inside of our public directory.

Calling APIs

Now that we have our app being served by our local node server, we can call our own APIs that we'll develop inside of our express server.

For instance, let's develop an application that will record the amount of times a button was hit. We'll need to write two routes:

GET /hits This route will return our total list of hits to the button.

POST /hit This route will record a new hit to the button and return us the latest total number of hits.

First, let's build the basic view of our app's `index.html` page. We'll place this in the `public/` directory of our node app so that if the route requested is either `/` or `/index.html`, express will serve this file as our route:

```
1 <!doctype html>
2 <html lang="en" ng-app="myApp">
3   <head>
4     <title>Node app</title>
5     <link rel="stylesheet" href="stylesheets/style.css">
6     <script src="bower_components/angular/angular.min.js"></script>
7   </head>
8   <body>
9     <div ng-controller="HomeCtrl">
10       <h3>Button hits: {{ hits }}</h3>
11       <button ng-click="registerHit()">
12         HIT ME, if you dare
13       </button>
14     </div>
15     <script src="javascripts/services.js"></script>
16     <script src="javascripts/app.js"></script>
17   </body>
18 </html>
```

Inside of our public/javascripts/app.js we'll add a controller on top of our myApp angular module:

```
1 angular.module('myApp', [
2   'ngRoute',
3   'myApp.services'
4 ])
5 .controller('HomeCtrl', function($scope, HitService) {
6   HitService.count()
7   .then(function(data) {
8     $scope.hits = data;
9   });
10
11   $scope.registerHit = function() {
12     HitService.registerHit()
13     .then(function(data) {
14       $scope.hits = data;
15     });
16   }
17 });
```

We'll build an angular service that is responsible for calling these routes as we can see in the controller above:

```
1 angular.module('myApp.services', [])
2 .factory('HitService', function($q, $http) {
3   var service = {
4     count: function() {
5       var d = $q.defer();
6       $http.get('/hits')
7       .success(function(data, status) {
8         d.resolve(data.hits);
9       }).error(function(data, status) {
10        d.reject(data);
11      });
12      return d.promise;
13    },
14    registerHit: function() {
15      var d = $q.defer();
16      $http.post('/hit', {})
17      .success(function(data, status) {
18        d.resolve(data.hits);
19      });
20    }
21  };
22  return service;
23});
```

```
19     }).error(function(data, status) {
20         d.reject(data);
21     });
22     return d.promise;
23 }
24 }
25 return service;
26});
```

For more information on services, check out the [services](#) chapter.

This service exposes two methods to us which call the routes that we defined above:

- count
- registerHit

Inside our node app's `app.js`, we will need to register two new routes and create the functionality that defines the routes for us.

These two new node routes will match the service routes that we're calling above:

```
1 // ...
2 var hits = require('./routes/hits');
3 // ...
4 app.get('/hits', hits.count);
5 app.post('/hit', hits.registerNew);
6 // ...
```

The only component that is left is building the actual back-end server-side logic that registers the hit count.

In NodeJS, each required module exposes methods through the method exports. To expose the two methods `count` and `registerNew` (from above), we'll need to attach these two the exports object inside the `routes/hits.js` file.

Inside of our `routes/hits.js` file, we'll create a `hits` store that stores the number of hits in memory, so that if we restart the server, the number of hits will also reset.

```
1  /*
2   * HIT service
3   */
4  var hits = 0;
5  exports.count = function(req, res){
6    res.send(200, {
7      hits: hits
8    });
9  }
10 exports.registerNew = function(req, res) {
11   hits += 1;
12   res.send(200, {
13     hits: hits
14   });
15 }
```

Now, if we start our node app and head to the route at `http://localhost:3000`, we will see that the functionality has been added for our angular app as we expect.





Server-less with Amazon AWS

One of the biggest benefits to building a single page app (SPA) is the ability to host flat files, rather than needing to build and service a back-end infrastructure.

However, most of the applications that we will build need to be powered by a back-end server with custom data. There are a growing number of options to enable us developers to focus on building only our front-end code and leave the back-ends alone.

Amazon released a new option for us late last week to allow us to build server-less web applications from right inside the browser: [Amazon AWS JavaScript SDK⁶⁹](#).

Their browser-based (and server-side with NodeJS) SDK allows us to confidently host our applications and interact with production-grade back-end services.

Now, it's possible to host our application stack entirely on Amazon infrastructure, using S3 to host our application and files, DynamoDB as a NoSQL store, and other web-scale services. We can even securely accept payments from the client side and get all the benefits of the Amazon CDN.

With this release, the JavaScript SDK now allows us to interact with 5 of the dozens of Amazon AWS services. These five services are:

⁶⁹<http://aws.amazon.com/>

DynamoDB

The fast, fully managed NoSQL database service that allows you to scale to *infinite* size with automatic triplicate replication with secure access controls.

Simple Notification Service (SNS)

The fast, flexible fully managed push notification service that allows us to push messages to mobile devices as well as other services, such as email or even to amazon's own Simple Queue Service (SQS).

Simple Queue Service (SQS)

The fast, reliable, fully managed queue service that allows us to create huge queues in a fully managed way. It allows us to create large request objects so we can fully decouple our application's components from each other using a common queue.

Simple Storage Service (S3)

The web-scale and fully managed data store that allows us to store large objects (up to 5 terabytes) with an unlimited number of objects. We can use S3 to securely store encrypted and protected data all over the world. We'll even use S3 to host our own Angular apps.

Security Token Service (STS)

The web-service that allows us to request temporary and limited privileged credentials for IAM users. We won't cover this in-depth, but it does provide a nice interface to creating limited secure operations on our data.

AWSJS + Angular

In this section, we intend on demonstrating how to get our applications up and running on the AWSJS stack in minutes.

To do so, we're going to create a mini, bare-bones version of [Gumroad⁷⁰](#) that we will allow our users to upload screenshots and we'll let them sell their screenshots by integrating with the fantastic [Stripe⁷¹](#) API.

We cannot recommend enough these two services and this mini-demo is not intended on replacing their services, only to demonstrate the power of Angular and the AWS API.

⁷⁰<https://gumroad.com/>

⁷¹<http://stripe.com>

To create our product, we'll need to:

- Allow users to login to our service and store their unique emails
- Allow users to upload files that are associated with them
- Allow buyers to click on images and present them with an option to buy the uploaded image
- Take credit card charges and accept money, directly from a single page angular app

Getting started

We'll start with a standard structured `index.html`:

```
1 <!doctype html>
2 <html>
3   <head>
4     <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.0-rc.3/angular.min.js"></script>
5     <script src="http://code.angularjs.org/1.2.0-rc.3/angular-route.min.js"></script>
6   </head>
7   <body>
8     <div ng-view></div>
9     <script src="scripts/app.js"></script>
10    <script src="scripts/controllers.js"></script>
11    <script src="scripts/services.js"></script>
12    <script src="scripts/directives.js"></script>
13  </body>
14 </html>
```

In this standard angular template, we're not loading anything crazy. We're loading the base angular library as well as `ngRoute` and our custom application code.

Our application code is also standard. Our `scripts/app.js` simply defines an angular module along with a single route `/`:

```
1 angular.module('myApp', [
2   'ngRoute',
3   'myApp.services',
4   'myApp.directives'])
5 .config(function($routeProvider) {
6   $routeProvider
7     .when('/', {
8       controller: 'MainCtrl',
9       templateUrl: 'templates/main.html',
10    })
11    .otherwise({
12      redirectTo: '/'
13    });
14  }));

```

Our `scripts/controllers.js` creates controllers from the main module:

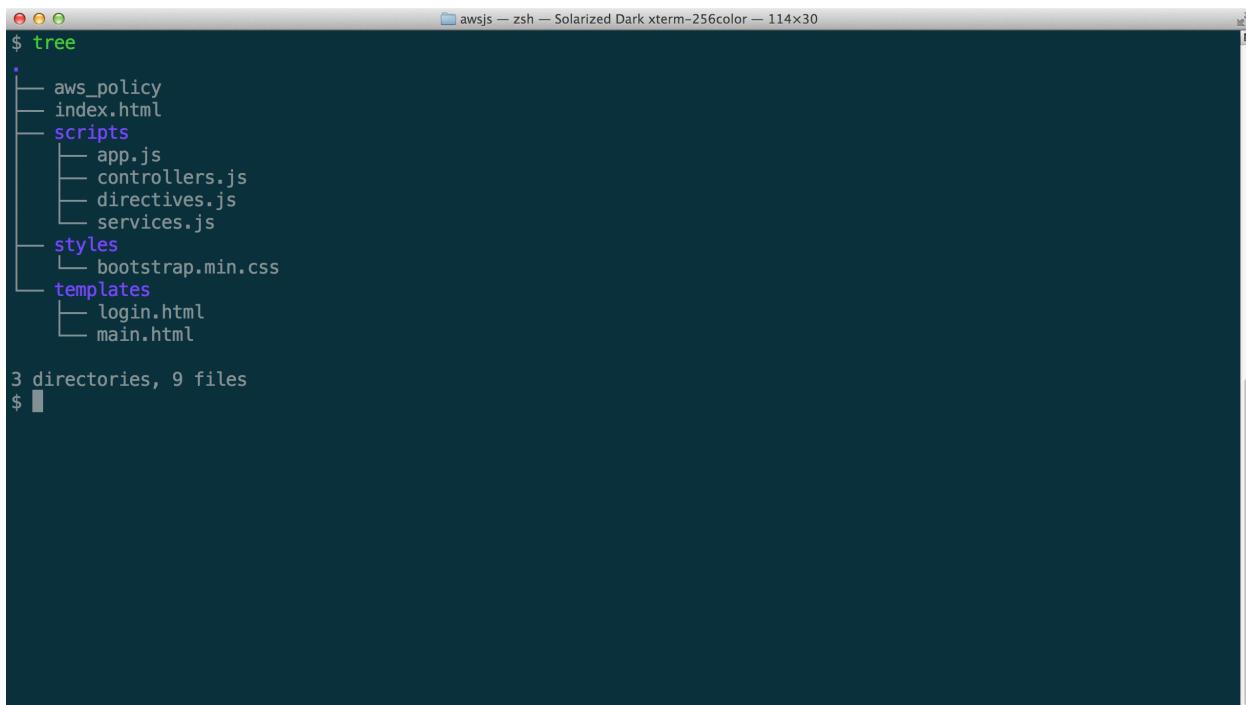
```
1 angular.module('myApp')
2 .controller('MainCtrl', function($scope) {
3
4 });

```

And our `scripts/services.js` and `scripts/directives.js` are simple as well:

```
1 // scripts/services.js
2 angular.module('myApp.services', []);

1 // scripts/directives.js
2 angular.module('myApp.directives', [])
```



```
$ tree
.
├── aws_policy
├── index.html
└── scripts
    ├── app.js
    ├── controllers.js
    ├── directives.js
    └── services.js
└── styles
    └── bootstrap.min.css
└── templates
    ├── login.html
    └── main.html

3 directories, 9 files
$
```

Angular Structure

Introduction

The aws ecosystem is huge and is used all over the world, in production. The gross amount of useful services that Amazon runs makes it a fantastic platform to power our applications on top of.

Historically, the APIs have not always been the easiest to use and understand, so we hope to address some of that confusion here.

Traditionally, we'd use a signed request with our applications utilizing the client_id/secret access key model. Since we're operating in the browser, it's not a good idea to embed our client_id and our client_secret in the browser where anyone can see it. (It's not much of a secret anyway if it's embedded in clear text, right?).

Luckily, the AWS team has provided us with an alternative method of identifying and authenticating our site to give access to the aws resources.

The first steps to creating an AWS-based angular app will be to set up this relatively complex authentication and authorization we'll use throughout the process.

Currently (at the time of this writing), the AWS JS library integrates cleanly with three authentication providers:

- Facebook
- Google Plus

- Amazon Login

In this section, we'll be focusing on integrating with the Google+ API to host our login, but the process is very similar for the other two authentication providers.

Installation

First things first, we'll need to *install* the files in our index.html. Inside of our index.html, we'll need to include the aws-sdk library as well as the Google API library.

We'll modify our index.html to include these libraries:

```
1 <!doctype html>
2 <html>
3   <head>
4     <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.0-rc.3/angular.min.js"></script>
5     <script src="http://code.angularjs.org/1.2.0-rc.3/angular-route.min.js"></script>
6   <script src="https://sdk.amazonaws.com/js/aws-sdk-2.0.0-rc1.min.js"></script>
7   <link rel="stylesheet" href="styles/bootstrap.min.css">
8   </head>
9   <body>
10    <div ng-view></div>
11    <script src="scripts/app.js"></script>
12    <script src="scripts/controllers.js"></script>
13    <script src="scripts/services.js"></script>
14    <script src="scripts/directives.js"></script>
15    <script type="text/javascript" src="https://js.stripe.com/v2/"></script>
16    <script type="text/javascript">
17      (function() {
18        var po = document.createElement('script'); po.type = 'text/javascript'; po.async = true;
19        po.src = 'https://apis.google.com/js/client:plusone.js?onload=onLoadCallback';
20        var s = document.getElementsByTagName('script')[0]; s.parentNode.insertBefore(po, s);
21      })();
22    </script>
23  </body>
24</html>
```

Now, notice that we added an `onload` callback for the Google JavaScript library and we **did not** use the `ng-app` to bootstrap our application. If we let angular automatically bootstrap our application, we'll run into a race condition where the Google API may not be loaded when the application starts.

This non-deterministic nature of our application will make the experience unusable, so instead we will manually bootstrap our app in the `onLoadCallback` function.

To manually bootstrap the application, we'll add the `onLoadCallback` function to the `window` service. Before we can call to bootstrap angular, we'll need to ensure that the google login client is loaded.

The google API client, or `gapi` gets included at run-time and is set by default to lazy-load its services. By telling the `gapi.client` to load the `oauth2` library in advance of starting our app, we will avoid any potential mishaps of the `oauth2` library being unavailable.

```
1 // in scripts/app.js
2 window.onLoadCallback = function() {
3     // When the document is ready
4     angular.element(document).ready(function() {
5         // Bootstrap the oauth2 library
6         gapi.client.load('oauth2', 'v2', function() {
7             // Finally, bootstrap our angular app
8             angular.bootstrap(document, ['myApp']);
9         });
10    });
11 }
```

With the libraries available and our application ready to be bootstrapped, we can set up the authorization part of our app.

Running

As we are using services that depend upon our URL to be an expected URL, we'll need to run this as a server, rather than simply loading the html in our browser.

We recommend using the incredibly simple python SimpleHTTPServer

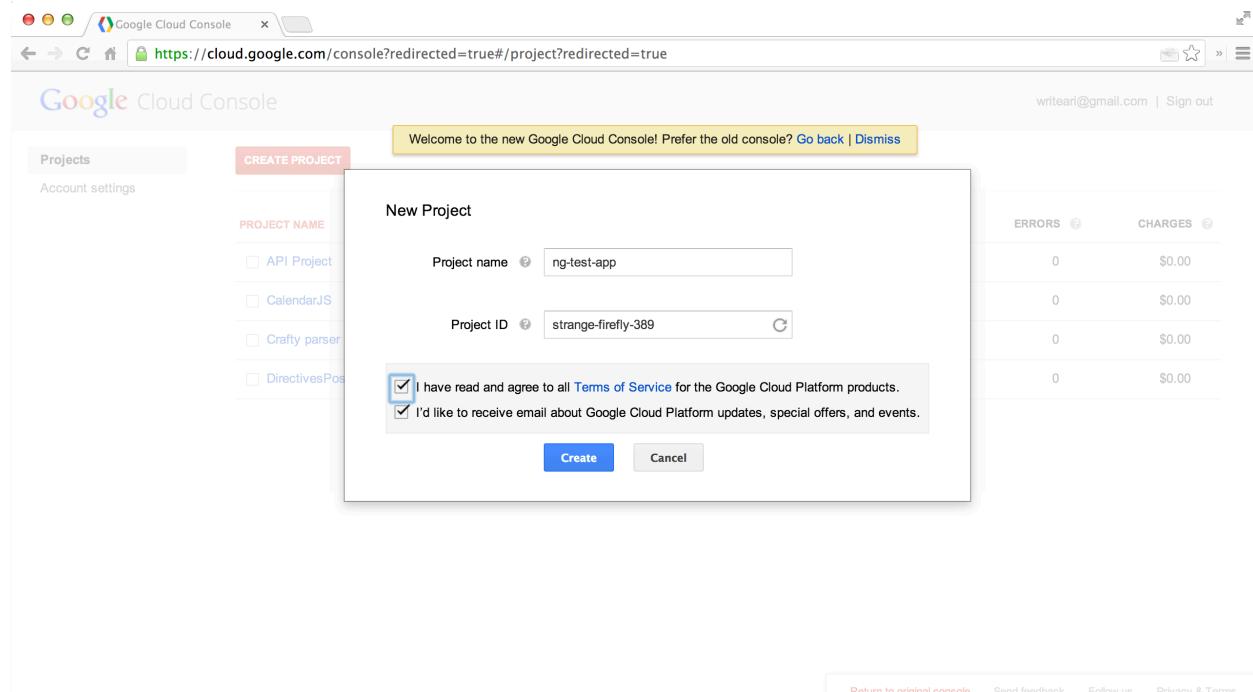
```
1 $ python -m SimpleHTTPServer 9000
```

Now we can load the url `http://localhost:9000/` in our browser.

User authorization/authentication

First, we'll need to get a `client_id` and a `client_secret` from Google so that we'll be able to actually interact with the google plus login system.

To get an app, head over to the [Google APIs console](#)⁷² and create a project.



Create a google plus project

Open the project by clicking on the name and click on the *APIs & auth* nav button. From here, we'll need to enable the Google+ API. Find the *APIs* button and click on it. Find the Google+ API item and click the OFF to ON slider.

⁷²<https://developers.google.com/console>

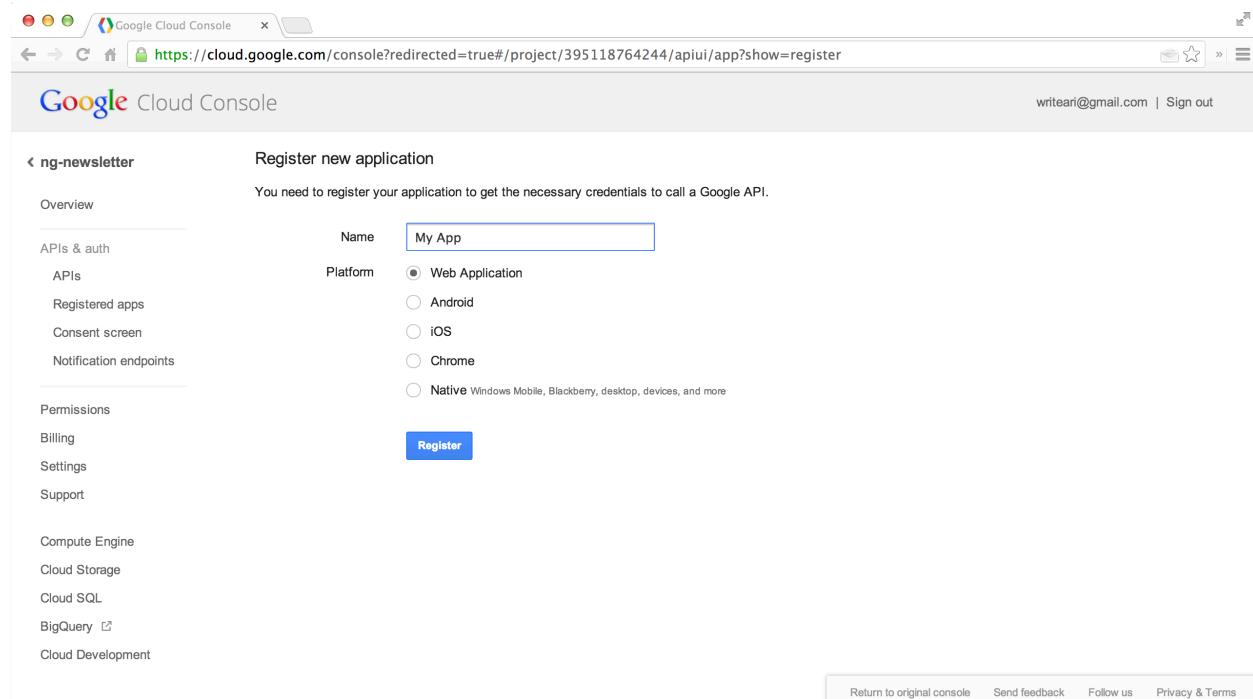
The screenshot shows the Google Cloud Console APIs page for the project 'ng-newsletter'. The left sidebar lists navigation options like Overview, APIs & auth, Permissions, and Compute Engine. The main area displays a table of APIs with columns for NAME and STATUS. The 'Google+ API' row is highlighted with a yellow background and has a green 'ON' button. Other APIs listed include BigQuery API, Google Cloud SQL, Google Cloud Storage, Google Cloud Storage JSON API, Ad Exchange Buyer API, Ad Exchange Seller API, Admin SDK, AdSense Host API, AdSense Management API, Analytics API, and Audit API, all currently set to OFF.

NAME	STATUS
BigQuery API	ON
Google Cloud SQL	ON
Google Cloud Storage	ON
Google Cloud Storage JSON API	ON
Google+ API	ON
Ad Exchange Buyer API	OFF
Ad Exchange Seller API	OFF
Admin SDK	OFF
AdSense Host API	OFF
AdSense Management API	OFF
Analytics API	OFF
Audit API	OFF

Enable Google+ API

Once that's set, we'll need to create and register an application and use it's application ID to make authenticated calls.

Find the Registered apps option and click on it to create an app. Make sure to select the Web Application option when it asks about the type of application.



Create a registered application

Once this is set, you'll be brought to the application details page. Select the OAuth 2.0 Client ID dropdown and take note of the application's Client ID. We'll use this in a few minutes.

Lastly, add the localhost origin to the WEB ORIGIN of the application. This will ensure we can develop with the API locally:

The screenshot shows the Google Cloud Console interface for managing OAuth 2.0 Client IDs. On the left, a sidebar lists various services like APIs & auth, Compute Engine, and Cloud Storage. The main area is titled 'My App' and shows the configuration for a 'Web Application'. It includes fields for 'CLIENT ID' (containing a long redacted string) and 'CLIENT SECRET' (also redacted). Below these are sections for 'CONSENT SCREEN' (with an 'Update' button), 'WEB ORIGIN' (set to 'http://localhost:9000'), and 'REDIRECT URI' (a text input field containing 'https:// or http://'). At the bottom is a blue 'Generate' button.

Registered app details

Next, we'll create a google+ login directive. This Angular directive will enable us to add a customized login button to our app with a single file element.

For more information about directives, check out our [in-depth post on directives⁷³](#).

We're going to have two pieces of functionality with our google login, we'll create an element that we'll attach a the standard google login button and we'll want to run a custom function after the button has been rendered.

The final directive will look like the following in `scripts/directives.js`:

```

1 angular.module('myApp.directives', [])
2 .directive('googleSignin', function() {
3   return {
4     restrict: 'A',
5     template: '<span id="signinButton"></span>',
6     replace: true,
7     scope: {
8       afterSignin: '&'
9     },
10    link: function(scope, ele, attrs) {

```

⁷³<http://www.ng-newsletter.com/posts/directives.html>

```
11     // Set standard google class
12     attrs.$set('class', 'g-signin');
13     // Set the clientid
14     attrs.$set('data-clientid',
15         attrs.clientId+'.apps.googleusercontent.com');
16     // build scope urls
17     var scopes = attrs.scopes || [
18         'auth/plus.login',
19         'auth/userinfo.email'
20     ];
21     var scopeUrls = [];
22     for (var i = 0; i < scopes.length; i++) {
23         scopeUrls.push('https://www.googleapis.com/' + scopes[i]);
24     };
25
26     // Create a custom callback method
27     var callbackId = "_googleSigninCallback",
28         directiveScope = scope;
29     window[callbackId] = function() {
30         var oauth = arguments[0];
31         directiveScope.afterSignin({oauth: oauth});
32         window[callbackId] = null;
33     };
34
35     // Set standard google signin button settings
36     attrs.$set('data-callback', callbackId);
37     attrs.$set('data-cookiepolicy', 'single_host_origin');
38     attrs.$set('data-requestvisibleactions', 'http://schemas.google.com/AddActi\
39 vity')
40     attrs.$set('data-scope', scopeUrls.join(' '));
41
42     // Finally, reload the client library to
43     // force the button to be painted in the browser
44     (function() {
45         var po = document.createElement('script'); po.type = 'text/javascript'; po\
46 .async = true;
47         po.src = 'https://apis.google.com/js/client:plusone.js';
48         var s = document.getElementsByTagName('script')[0]; s.parentNode.insertBef\
49 ore(po, s);
50     })();
51 }
52 }
```

```
53  });
```

Although it's long, it's fairly straightforward. We're assigning the google button class `g-signin`, attaching the `clientid` based on an attribute we pass in, building the scopes, etc.

One unique part of this directive is that we're creating a custom callback on the `window` object. Effectively, this will allow us to *fake* the callback method needing to be called in JavaScript when we make the function to allow us to actually make the call to the local `afterSignin` action instead.

We'll then clean up the global object because we're allergic to global state in AngularJS.

With our directive primed and ready to go, we can include the directive in our view. We're going to call the directive in our view like so, replacing the `client-id` and the `after-signin` attributes on the directive to our own.

Make sure to include the `oauth` parameter exactly as it's spelled in the `after-signup` attribute. This is called this way due to how angular directives call methods with parameters inside of directives.

```
1 <h2>Signin to ngroad</h2>
2 <div google-signin
3   client-id='CLIENT_ID'
4   after-signin="signedIn(oauth)"></div>
5 <pre>{{ user | json }}</pre>
```

The user data in the example is the returned `access_token` for your login (if you log in). It is **not** saved on our servers, not sensitive data, and will disappear when you leave the page.

Finally, we'll need our button to *actually* cause an action, so we'll need to define our `after-signin` method `signedIn(oauth)` in our controller.

This `signedIn()` method will kill off the authenticated page for us in our real application. Note, this method would be an ideal place to set a redirect to a new route, for instance the `/dashboard` route for authenticated users.

```

1 angular.module('myApp')
2 .controller('MainCtrl',
3   function($scope) {
4     $scope.signedIn = function(oauth) {
5       $scope.user = oauth;
6     }
7 });

```

UserService

Before we dive a bit deeper into the AWS-side of things, let's create ourselves a `UserService` that is responsible for holding on to our new user. This `UserService` will handle the bulk of the work for interacting with the AWS backend as well as keep a copy of the current user.

Although we're not quite ready to attach a backend, we can start building it out to handle holding on to a persistent copy of the user instance.

In our `scripts/services.js`, we'll create the beginnings of our `UserService`:

```

1 angular.module('myApp.services', [])
2 .factory('UserService', function($q, $http) {
3   var service = {
4     _user: null,
5     setCurrentUser: function(u) {
6       if (u && !u.error) {
7         service._user = u;
8         return service.currentUser();
9       } else {
10         var d = $q.defer();
11         d.reject(u.error);
12         return d.promise;
13       }
14     },
15     currentUser: function() {
16       var d = $q.defer();
17       d.resolve(service._user);
18       return d.promise;
19     }
20   };
21   return service;
22 });

```

Although this setup is a bit contrived for the time being, we'll want the functionality to set the `currentUser` as a permanent fixture in the service.

Remember, services are singleton objects that live for the duration of the application lifecycle.

Now, instead of simply setting our user in the return of the `signedIn()` function, we can set the user to the `UserService`:

```
1 angular.module('myApp')
2   .controller('MainCtrl',
3     function($scope) {
4       $scope.signedIn = function(oauth) {
5         UserService.setCurrentUser(oauth)
6           .then(function(user) {
7             $scope.user = user;
8           });
9       }
10    });
11 
```

For our application to work, we're going to need to hold on to actual user emails so we can provide a better method of interacting with our users as well as holding on to some persistent, unique data per-user.

We'll use the `gapi.client.oauth2 userinfo.get()` method to fetch the user's email address rather than holding on to the user's `access_token` (and other various access details).

In our `UserService`, we'll update our `currentUser()` method to include this functionality:

```
1 // ...
2 },
3 currentUser: function() {
4   var d = $q.defer();
5   if (service._user) {
6     d.resolve(service._user);
7   } else {
8     gapi.client.oauth2 userinfo.get()
9       .execute(function(e) {
10         service._user = e;
11       })
12   }
13   return d.promise;
14 }
15 // ...
```

All aboard AWS

Now, as we said when we first started this journey, we'll need to set up authorization with the AWS services.

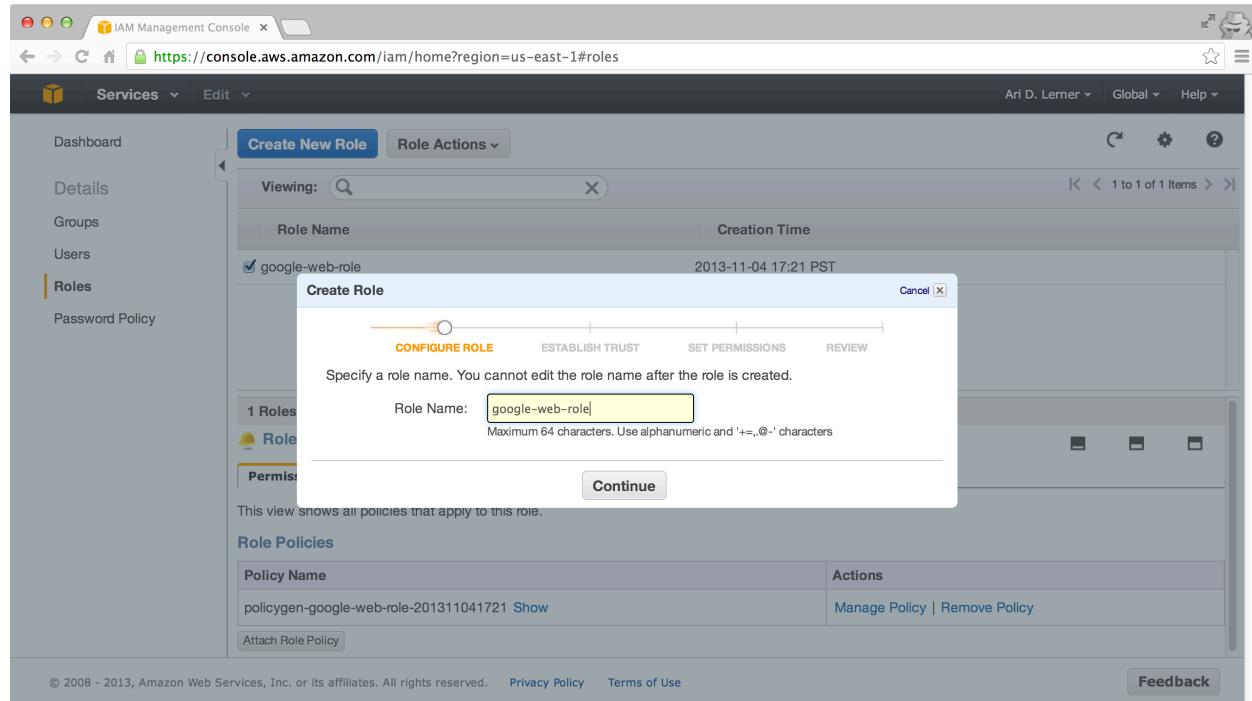
If you do not have an AWS account, head over to aws.amazon.com⁷⁴ and grab an account. It's free and quick.

Now, first things first: we'll need to create an IAM role. IAM, or AWS's Identity and Access Management service is one of the reasons why the AWS services are so powerful. We can create fine-grain access controls over our systems and data using IAM.

Unfortunately, this flexibility and power of IAM also makes it a bit more complex, so we'll walk through creating it here and making it as clear as we can.

Let's create the IAM role. Head to the [IAM console](#)⁷⁵ and click on the Roles navigation link.

Click the *Create New Role* button and give our new role a name. We'll call ours the `google-web-role`.

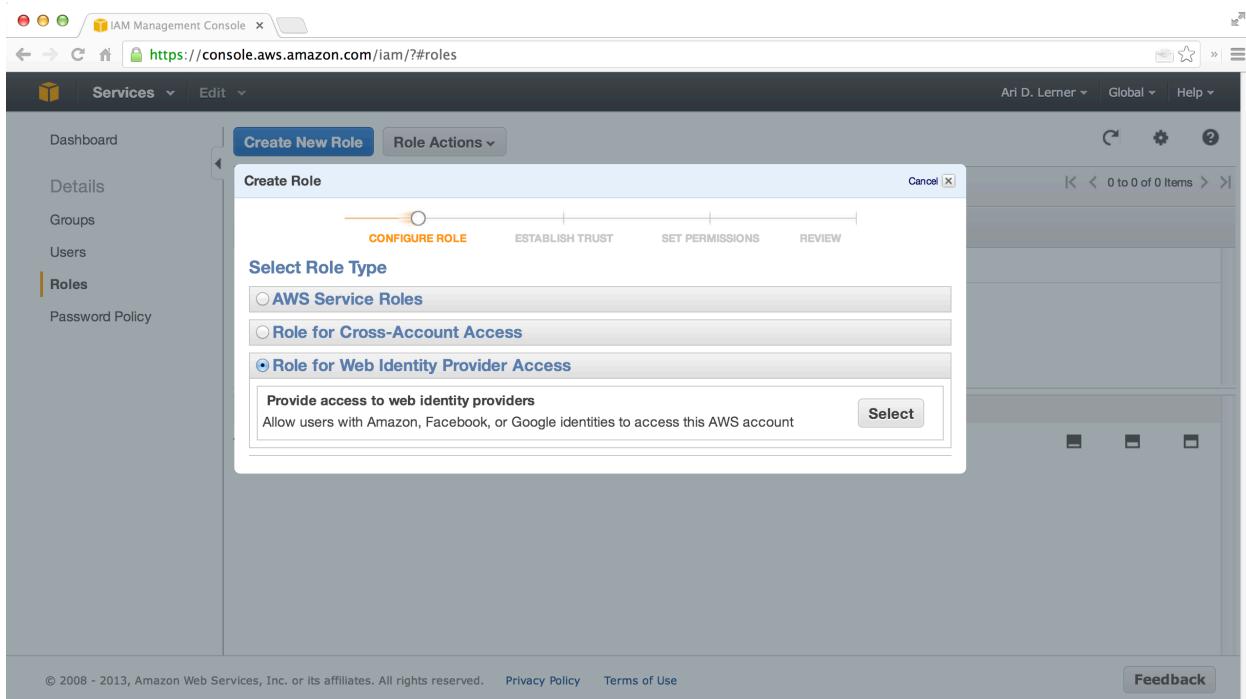


Create a new role

Next, we'll need to configure the IAM role to be a *Web Identity Provider Access* role type. This is how we'll be able to manage our new role's access to our AWS services.

⁷⁴<http://aws.amazon.com/>

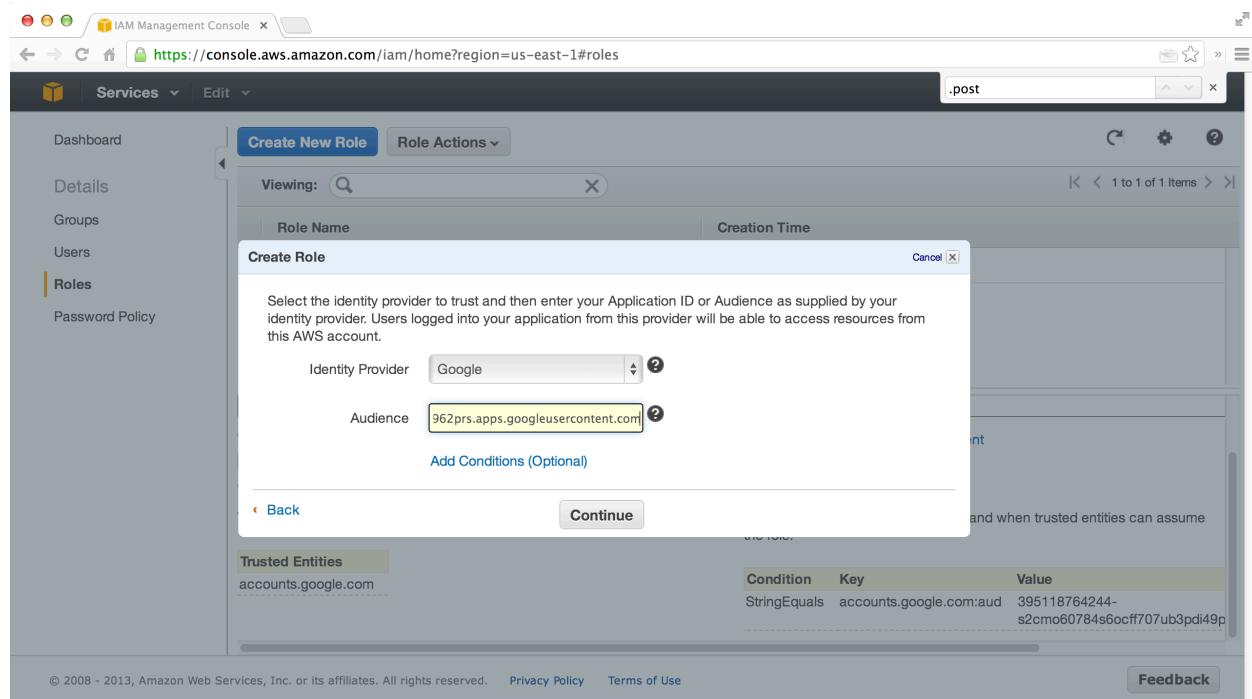
⁷⁵<https://console.aws.amazon.com/iam/home?region=us-east-1#roles>



Set the role type

Now, remember the CLIENT ID that we created from google above? In the next screen, select Google from the dropdown and paste the CLIENT ID into the Audience box.

This will join our IAM role and our Google app together so that our application can call out to AWS services with an authenticated Google user.



Google auth

Click through the *Verify Trust* (the next screen). This screen shows the raw configuration for AWS services. Next, we'll create the policy for our applications.

The Policy Generator is the easiest method of getting up and running quickly to build policies. This is where we'll set what actions our users can and cannot take.

In this step, we're going to be taking very specific actions that our web users can take. We're going to allow our users to the following actions for each service:

S3

On the specific bucket (`ng-newsletter-example`, in our example app), we're going to allow our users to take the following actions:

- GetObject
- ListBucket
- PutObject

The Amazon Resource Name (ARN) for our s3 bucket is:

```
1 arn:aws:s3:::ng-newsletter-example/*
```

DynamoDB

For two specific table resources, we're going to allow the following actions:

- GetItem
- PutItem
- Query

The Amazon Resource Name (ARN) for our dynamoDB tables are the following:

```

1  [
2    "arn:aws:dynamodb:us-east-1:<ACCOUNT_ID>:table/Users",
3    "arn:aws:dynamodb:us-east-1:<ACCOUNT_ID>:table/UsersItems"
4  ]

```

Your

The final version of our policy can be found [here](#)⁷⁶.

Effect	Action	Resource
Allow	s3:GetObject s3:PutObject	arn:aws:s3:::uploads/*

Adding the IAM policy

⁷⁶<http://d.pr/9Obj>

For more information on the confusing ARN numbers, check out the amazon documentation on them [here⁷⁷](#).

One final piece of information that we'll need to hold on to is the Role ARN. We can find this Role ARN on the summary tab of the IAM user in our IAM console.

Take note of this string as we'll set it in a moment.

Role Name	Creation Time
google-web-role	2013-11-04 17:21 PST

Role ARN: arn:aws:iam::123456789012:role/google-web-role

Role ARN

Now that we're finally done with creating our IAM user, we can move on to integrating it inside of our angular app.

AWSService

We'll move the root of our application for integrating with AWS into its own service we're going to build called the `AWSService`.

Since we are going to need to have the ability to custom configure our service at configure-time, we'll want to create it as a provider.

Remember, the *only* service-type that can be *injected* into the `.config()` function is the `.provider()` type.

First, we'll create the stub of our provider in `scripts/services.js`:

⁷⁷<http://docs.aws.amazon.com/general/latest/gr/aws-arns-and-namespaces.html>

```
1 // ...
2 .provider('AWSService', function() {
3   var self = this;
4   self.arn = null;
5
6   self.setArn = function(arn) {
7     if (arn) self.arn = arn;
8   }
9
10  self.$get = function($q) {
11    return {}
12  }
13});
```

As we can already start to notice, we'll need to set the `Role ARN` for this service so that we can attach the proper user to the correct services.

Setting up our `AWSService` as a provider like we do above enables us to set the following in our `scripts/app.js` file:

```
1 angular.module('myApp',
2   ['ngRoute', 'myApp.services', 'myApp.directives']
3 )
4 .config(function(AWSServiceProvider) {
5   AWSServiceProvider
6     .setArn(
7       'arn:aws:iam:<ACCOUNT_ID>:role/google-web-role');
8 })
```

Now, we can carry on with the `AWSService` and not worry about overriding our `Role ARN` as well as it becomes incredibly easy to share amongst our different applications instead of recreating it every time.

Our `AWSService` at this point doesn't really do anything yet. The last component that we'll need to ensure works is that we give access to our actual users who log in.

This final step is where we'll need to tell the AWS library that we have an authenticated user that can operate as our IAM role.

We'll create this `credentials` as a promise that will eventually be resolved so we can define the different portions of our application without needing to bother checking if the credentials have been loaded simply by using the `.then()` method on promises.

Let's modify our `$get()` method in our service adding a method that we'll call `setToken()` to create a new set of `WebIdentityCredentials`:

```
1 // ...
2 self.$get = function($q) {
3     var credentialsDefer = $q.defer(),
4         credentialsPromise = credentialsDefer.promise;
5     return {
6         credentials: function() {
7             return credentialsPromise;
8         },
9         setToken: function(token, providerId) {
10            var config = {
11                RoleArn: self.arn,
12                WebIdentityToken: token,
13                RoleSessionName: 'web-id'
14            }
15            if (providerId) {
16                config['ProviderId'] = providerId;
17            }
18            self.config = config;
19            AWS.config.credentials =
20                new AWS.WebIdentityCredentials(config);
21            credentialsDefer
22                .resolve(AWS.config.credentials);
23        }
24    }
25 }
26 // ...
```

Now, when we get our oauth.access_token back from our login through Google, we'll pass in the id_token to this function which will take care of the AWS config setup.

Let's modify the UserService service such that we call the setToken() method:

```
1 // ...
2 .factory('UserService', function($q, $http) {
3     var service = {
4         _user: null,
5         setCurrentUser: function(u) {
6             if (u && !u.error) {
7                 AWSService.setToken(u.id_token);
8                 return service.currentUser();
9             } else {
10                 var d = $q.defer();
11                 d.reject(u.error);
```

```
12     return d.promise;
13   }
14 },
15 // ...
```

Starting on dynamo

In our application, we'll want to associate any images that one user uploads to that unique user. To create this association, we'll create a dynamo table that stores our users as well as another that stores the association between the user and the user's uploaded files.

To start interacting with dynamo, we'll first need to instantiate a dynamo object. We'll do this inside of our `AWSService` service object, like so:

```
1 // ...
2 setToken: function(token, providerId) {
3   // ...
4 },
5 dynamo: function(params) {
6   var d = $q.defer();
7   credentialsPromise.then(function() {
8     var table = new AWS.DynamoDB(params);
9     d.resolve(table);
10    });
11   return d.promise;
12 },
13 // ...
```

As we discussed earlier, by using promises inside of our service objects, we only need to use the promise `.then()` api method to ensure our credentials are set when we're starting to use them.

You might ask why we're setting `params` with our dynamo function. Sometimes we'll want to interact with our dynamoDB with different configurations and different setups. This might cause us to need to recreate objects that we already use once in our page.

Rather than having this duplication around with our different AWS objects, we'll *cache* these objects using the built-in angular `$cacheFactory` service.

\$cacheFactory

The `$cacheFactory` service enables us to create an object if we need it or recycle and reuse an object if we've already needed it in the past.

To start caching, we'll create a `dynamoCache` object where we'll store our cached dynamo objects:

```
1 // ...
2 self.$get = function($q, $cacheFactory) {
3   var dynamoCache = $cacheFactory('dynamo'),
4     credentialsDefer = $q.defer(),
5     credentialsPromise = credentialsDefer.promise;
6
7   return {
8     // ...
9   }
10 }
```

Now, back in our dynamo method, we can draw from the cache if the object exists in the cache or we can set it to create the object when necessary:

```
1 // ...
2 dynamo: function(params) {
3   var d = $q.defer();
4   credentialsPromise.then(function() {
5     var table =
6       dynamoCache.get(JSON.stringify(params));
7     if (!table) {
8       var table = new AWS.DynamoDB(params);
9       dynamoCache.put(JSON.stringify(params), table);
10    };
11    d.resolve(table);
12  });
13  return d.promise;
14 },
15 //
```

Saving our currentUser

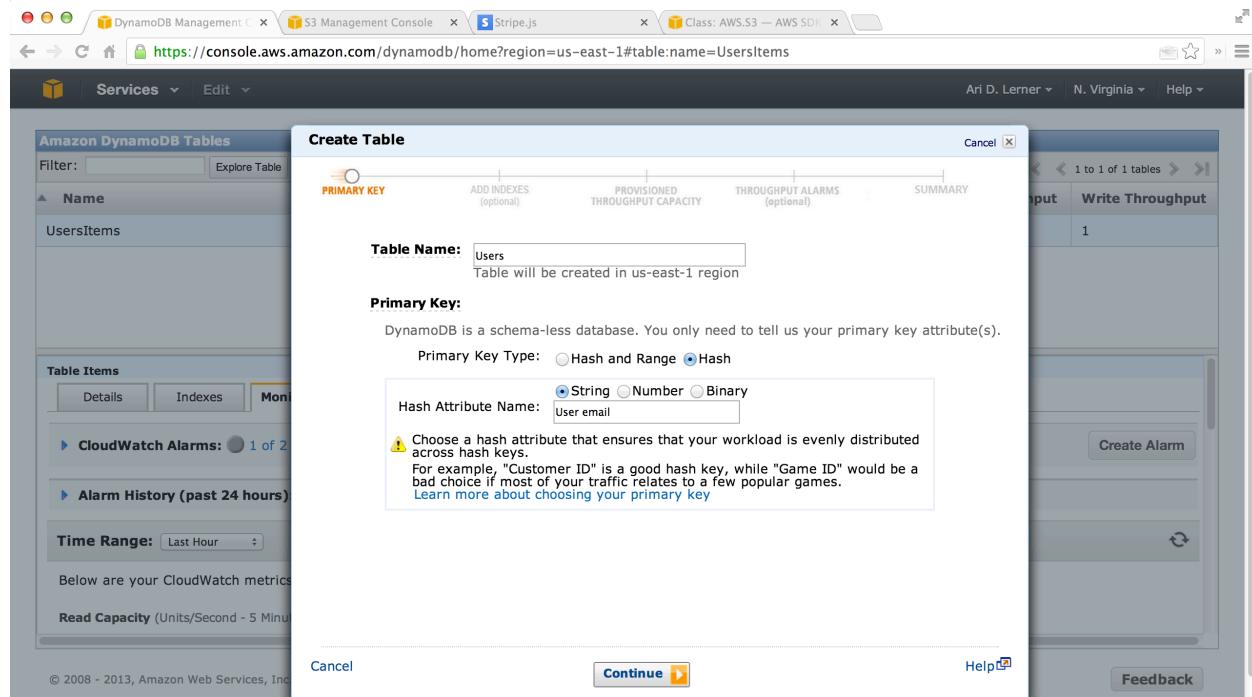
When a user logs in and we fetch the user's email, this is a good point for us to add the user to our user's database.

To create a dynamo object, we'll use the promise api method `.then()` again, this time outside of the service. We'll create an object that will enable us to interact with the User's table we'll create in the dynamo API console.

We'll need to manually create these dynamo tables the first time because we do not want to give access to our web users the ability to create dynamo tables, which might include us.

To create a dynamo table, head to the [dynamo console](#)⁷⁸ and find the Create Table button.

Create a table called `Users` with a primary key type of `Hash`. The Hash Attribute Name will be the primary key that we'll use to get and put objects on the table. For this demo, we'll use the string: `User_email`.



Create the Users dynamo table

Click through the next two screens and set up a basic alarm by entering your email. Although this step isn't 100% necessary, it's easy to forget that our tables are up and without being reminded, we might just end up leaving them up forever.

Once we've clicked through the final review screen and click create, we'll end up with a brand new Dynamo table where we will store our users.

While we are at the console, we'll create the *join* table. This is the table that will join the User and the items they upload.

Find the Create Table button again and create a table called `UsersItems` with a primary key type of `Hash and Range`. For this table, The Hash Attribute Name will also be `User_email` and the Range Attribute Name will be `ItemId`.

This will allow us to query for User's who have created items based on the User's email.

The rest of the options that are available on the next screens are optional and we can click through the rest.

At this point, we have two dynamo tables available.

⁷⁸<https://console.aws.amazon.com/dynamodb/home>

Back to our UserService, we'll first query the table to see if the user is already saved in our database, otherwise we'll create an entry in our dynamo database.

```
1 var service = {
2   _user: null,
3   UsersTable: "Users",
4   UserItemsTable: "UsersItems",
5   // ...
6   currentUser: function() {
7     var d = $q.defer();
8     if (service._user) {
9       d.resolve(service._user);
10    } else {
11      // After we've loaded the credentials
12      AWSService.credentials().then(function() {
13        gapi.client.oauth2 userinfo.get()
14          .execute(function(e) {
15            var email = e.email;
16            // Get the dynamo instance for the
17            // UsersTable
18            AWSService.dynamo({
19              params: {TableName: service.UsersTable}
20            })
21            .then(function(table) {
22              // find the user by email
23              table.getItem({
24                Key: {'User email': {S: email}}
25              }, function(err, data) {
26                if (Object.keys(data).length == 0) {
27                  // User didn't previously exist
28                  // so create an entry
29                  var itemParams = {
30                    Item: {
31                      'User email': {S: email},
32                      data: { S: JSON.stringify(e) }
33                    }
34                  };
35                  table.putItem(itemParams,
36                    function(err, data) {
37                      service._user = e;
38                      d.resolve(e);
39                    });
40                } else {
```

```
41         // The user already exists
42         service._user =
43             JSON.parse(data.Item.data.S);
44         d.resolve(service._user);
45     }
46 );
47 );
48 );
49 );
50 }
51 return d.promise;
52 },
53 // ...
```

Although it looks like a lot of code, this simply does a `find` or `create` by `username` on our dynamoDB.

At this point, we can finally get back to our view and check out what's happening in the view.

In our `templates/main.html`, we'll add a container that simply shows the Login form if there is no user and shows the user details if there is a user.

We'll do this with simple `ng-show` directives and our new `google-signin` directive.

```
1 <div class="container">
2   <h1>Home</h1>
3   <div ng-show="!user" class="row">
4     <div class="col-md-12">
5       <h2>Signup or login to ngroad</h2>
6       <div google-signin
7         client-id='395118764200'
8         after-signin="signedIn(oauth)"></div>
9     </div>
10   </div>
11   <div ng-show="user">
12     <pre>{{ user | json }}</pre>
13   </div>
14 </div>
```

With our view set up, we can now work with logged in users inside the second `<div>` (in production, it's a good idea to make it a separate route).

Uploading to s3

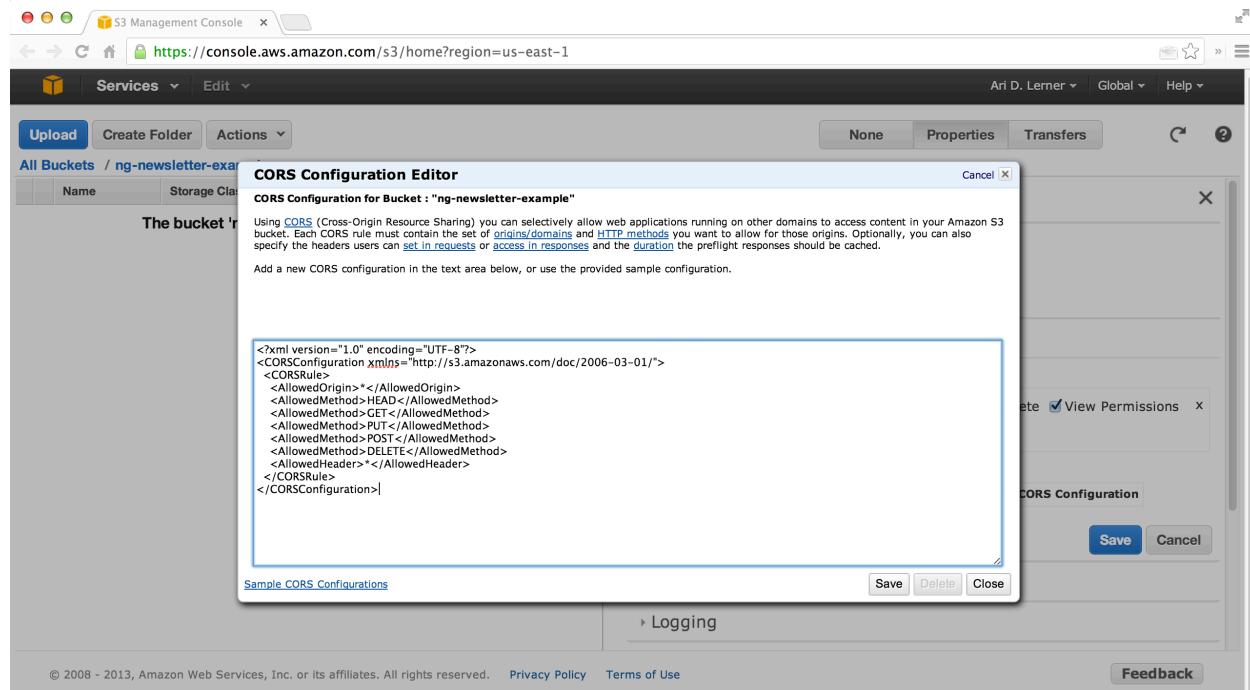
Now that we have our logged in user stored in dynamo, it's time we create the ability to handle file upload where we'll store our files directly on S3.

First and foremost, a shallow dive into CORS. CORS, or Cross-Origin Resource Sharing is a security features that modern browsers support allowing us to make requests to foreign domains using a standard protocol.

Luckily, the AWS team has made supporting CORS incredibly simple. If we're hosting our site on s3, then we don't even need to set up CORS (other than for development purposes).

To enable CORS on a bucket, head to the [s3 console⁷⁹](https://console.aws.amazon.com/s3/home?region=us-east-1) and find the bucket that we're going to use for file uploads. For this demo, we're using the `ng-newsletter-example` bucket.

Once the bucket has been located, click on it and load the Properties tab and pull open the Permissions option. click on the *Add CORS configuration* button and pick the standard CORS configuration.



Enable CORS on an S3 bucket

We'll create a simple file upload directive that kicks off a method that uses the HTML5 File API to handle the file upload. This way, when the user selects the file the file upload will immediately start.

To handle the file selection directive, we'll create a simple directive that binds to the change event and calls a method after the file has been selected.

⁷⁹<https://console.aws.amazon.com/s3/home>

The directive is simple:

```

1 // ...
2 .directive('fileUpload', function() {
3   return {
4     restrict: 'A',
5     scope: { fileUpload: '&' },
6     template: '<input type="file" id="file" /> ',
7     replace: true,
8     link: function(scope, ele, attrs) {
9       ele.bind('change', function() {
10         var file = ele[0].files;
11         if (file) scope.fileUpload({files: file});
12       })
13     }
14   }
15 })
```

This directive can be used in our view like so:

```

1 <!-- ... -->
2 <div class="row"
3   <div class="col-md-12"
4     <div file-upload="onFile(files)"></div>
5   </div>
6 </div>
```

Now, when the file selection has been made, it will call the method `onFile(files)` in our current scope.

Although we're creating our own file directive here, we recommend checking out the `ngUpload`⁸⁰ library for handling file uploads.

Inside the `onFile(files)` method, we'll want to handle the file upload to s3 and save the record to our dynamo database table. Instead of placing this functionality in the controller, we'll be nice angular citizens and place this in our `UserService` service.

First, we'll need to make sure we have the ability to get an s3 JavaScript object just like we made the `dynamo` available.

⁸⁰<https://github.com/twilson63/ngUpload>

```
1 // ...
2 var dynamoCache = $cacheFactory('dynamo'),
3     s3Cache = $cacheFactory('s3Cache');
4 // ...
5 return {
6     // ...
7     s3: function(params) {
8         var d = $q.defer();
9         credentialsPromise.then(function() {
10             var s3Obj = s3Cache.get(JSON.stringify(params));
11             if (!s3Obj) {
12                 var s3Obj = new AWS.S3(params);
13                 s3Cache.put(JSON.stringify(params), s3Obj);
14             }
15             d.resolve(s3Obj);
16         });
17         return d.promise;
18     },
19 // ...
```

This method works the exact same way that our dynamo object creation works, giving us direct access to the s3 instance object as we'll see shortly.

Handling file uploads

To handle file uploads, we'll create a method that we'll call `uploadItemForSale()` in our `UserService`. Planning our functionality, we'll want to:

- Upload the file to S3
- Get a signedUrl for the file
- Save this information to our database

We're going to be using our current user through this process, so we'll start out by making sure we have our user and get an instance

```
1 // in scripts/services.js
2 // ...
3 },
4 Bucket: 'ng-newsletter-example',
5 uploadItemForSale: function(items) {
6   var d = $q.defer();
7   service.currentUser().then(function(user) {
8     // Handle the upload
9     AWSService.s3({
10       params: {
11         Bucket: service.Bucket
12       }
13     }).then(function(s3) {
14       // We have a handle of our s3 bucket
15       // in the s3 object
16     });
17   });
18   return d.promise;
19 },
20 // ...
```

With the handle of the s3 bucket, we can create a file to upload. There are 3 required parameters when uploading to s3:

- Key - The key of the file object
- Body - The file blob itself
- ContentType - The type of file

Luckily for us, all this information is available on the file object when we get it from the browser.

```
1 // ...
2 // Handle the upload
3 AWSService.s3({
4   params: {
5     Bucket: service.Bucket
6   }
7 }).then(function(s3) {
8   // We have a handle of our s3 bucket
9   // in the s3 object
10  var file = items[0]; // Get the first file
11  var params = {
```

```

12     Key: file.name,
13     Body: file,
14     ContentType: file.type
15   }
16
17   s3.putObject(params, function(err, data) {
18     // The file has been uploaded
19     // or an error has occurred during the upload
20   });
21 });
22 // ...

```

By default, s3 uploads files in a protected form. It prevents us from uploading and having the files available to the public without some work. This is a definite *feature* as anything that we upload to s3 will be protected and it forces us to make conscious choices about what files will be public and which are not.

With that in mind, we'll create a temporary url that expires after a given amount of time. In our ngroad marketplace, this will give a time-expiry on each of the items that are available for sale.

In any case, to create a temporary url, we'll fetch a `signedUrl` and store that in our join table for User's Items:

```

1 // ...
2 s3.putObject(params, function(err, data) {
3   if (!err) {
4     var params = {
5       Bucket: service.Bucket,
6       Key: file.name,
7       Expires: 900*4 // 1 hour
8     };
9     s3.getSignedUrl('getObject', params,
10       function(err, url) {
11         // Now we have a url
12       });
13     }
14   });
15 });
16 // ...

```

Finally, we can save our User's object along with the file they uploaded in our Join table:

```
1 // ...
2 s3.getSignedUrl('getObject', params,
3   function(err, url) {
4     // Now we have a url
5     AWSService.dynamo({
6       params: {TableName: service.UserItemsTable}
7     }).then(function(table) {
8       var itemParams = {
9         Item: {
10           'ItemId': {S: file.name},
11           'User email': {S: user.email},
12           data: {
13             S: JSON.stringify({
14               itemId: file.name,
15               itemSize: file.size,
16               itemUrl: url
17             })
18           }
19         }
20       };
21       table.putItem(itemParams, function(err, data) {
22         d.resolve(data);
23       });
24     });
25   });
26 // ...
```

This method, all together is available [here⁸¹](#).

We can use this new method inside of our controller's `onFile()` method, which we can write to be similar to:

```
1 $scope.onFile = function(files) {
2   UserService.uploadItemForSale(files)
3   .then(function(data) {
4     // Refresh the current items for sale
5   });
6 }
```

⁸¹<https://gist.github.com/auser/7316267#file-services-js-L98>

Querying dynamo

Ideally, we'll want to be able to list all the products a certain user has available for purchase. In order to set up a listing of the available items, we will use the query api.

The dynamo query api is a tad esoteric and can be considerably confusing when looking at it at first glance.

The dynamo query documentation is available http://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_Query.html⁸²

Basically, we'll match object schemes up with a comparison operator, such as equal, lt (less than), or gt (greater than) and several more. Our join table's key is the `User email` key, so we'll match this key against the current user's email as the query key.

As we did with our other APIs related to users, we'll creat a method inside of our `UserService` to handle this querying of the database:

```

1 // ...
2 itemsForSale: function() {
3   var d = $q.defer();
4   service.currentUser().then(function(user) {
5     AWSService.dynamo({
6       params: {TableName: service.UserItemsTable}
7     }).then(function(table) {
8       table.query({
9         TableName: service.UserItemsTable,
10        KeyConditions: {
11          "User email": {
12            "ComparisonOperator": "EQ",
13            "AttributeValueList": [
14              {S: user.email}
15            ]
16          }
17        }
18      }, function(err, data) {
19        var items = [];
20        if (data) {
21          angular.forEach(data.Items, function(item) {
22            items.push(JSON.parse(item.data.S));
23          });
24        }
25      });
26    });
27  });
28}

```

⁸²http://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_Query.html

```
24         d.resolve(items);
25     } else {
26         d.reject(err);
27     }
28   })
29 });
30 });
31 return d.promise;
32 },
33 // ...
```

In the above query, the `KeyConditions` and "User email" are required parameters.

Showing the listing in HTML

To show our user's images in HTML, we'll simply assign the result of our new `itemsForSale()` method to a property of the controller's scope:

```
1 var getItemsForSale = function() {
2   UserService.itemsForSale()
3     .then(function(images) {
4       $scope.images = images;
5     });
6 }
7 // Load the user's list initially
8 getItemsForSale();
```

Now we can iterate over the list of items easily using the `ng-repeat` directive:

```
1 <!-- ... -->
2 <div ng-show="images">
3   <div class="col-sm-6 col-md-4"
4     ng-repeat="image in images">
5     <div class="thumbnail">
6       
8     </div>
9   </div>
10 </div>
```

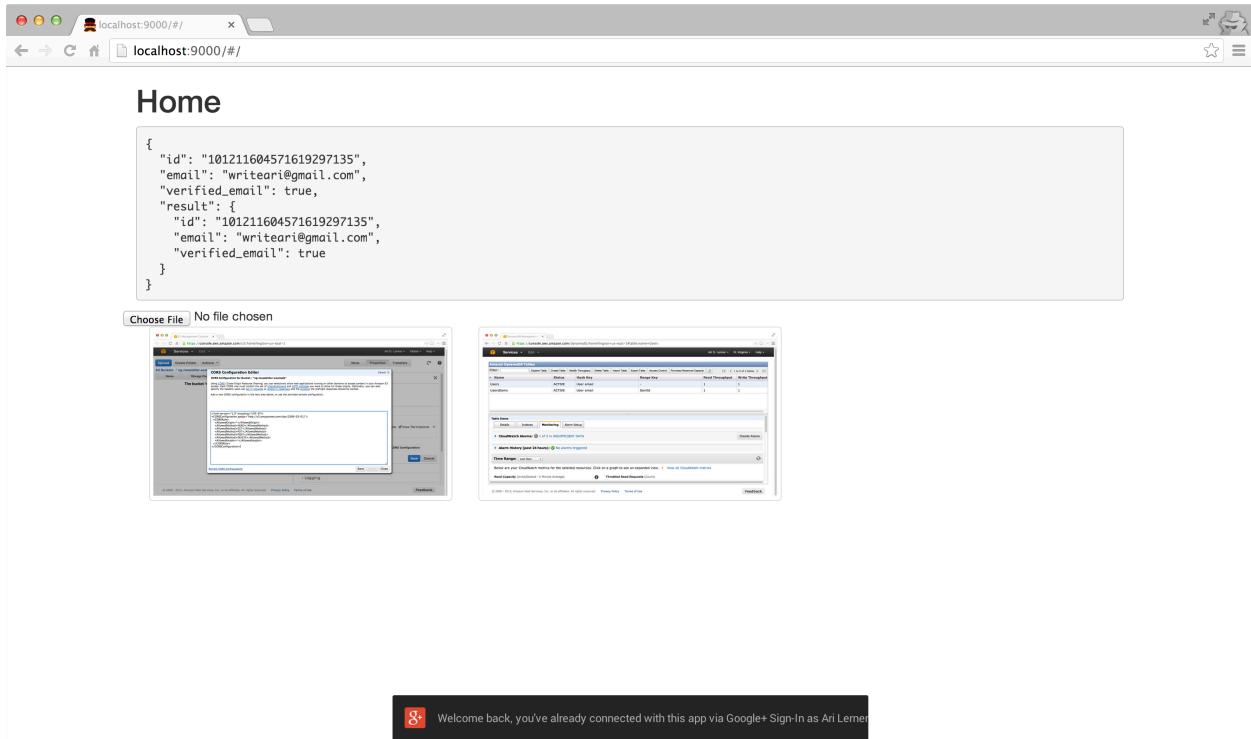


Image listing

Selling our work

The final component of our AWS-powered demo app is the ability to create sales from our Single Page App.

In order to actually take money from customers, we'll need a thin backend component that will need to convert stripe tokens into sales on Stripe. We cover this in our upcoming book that's available for pre-release at ng-book.com⁸³.

To start handling payments, we'll create a `StripeService` that will handle creating charges for us. Since we'll want to support configuring Stripe in the `.config()` method in our module, we'll need to create a `.provider()`.

The service itself is incredibly simple as it leverages the `Stripe.js` library to do the heavy lifting work.

⁸³<http://www.ng-book.com/>

```

1 // ...
2 .provider('StripeService', function() {
3   var self = this;
4
5   self.setPublishableKey = function(key) {
6     Stripe.setPublishableKey(key);
7   }
8
9   self.$get = function($q) {
10    return {
11      createCharge: function(obj) {
12        var d = $q.defer();
13
14        if (!obj.hasOwnProperty('number') ||
15            !obj.hasOwnProperty('cvc') ||
16            !obj.hasOwnProperty('exp_month') ||
17            !obj.hasOwnProperty('exp_year'))
18        ) {
19          d.reject("Bad input", obj);
20        } else {
21          Stripe.card.createToken(obj,
22            function(status, resp) {
23              if (status == 200) {
24                d.resolve(resp);
25              } else {
26                d.reject(status);
27              }
28            });
29        }
30
31        return d.promise;
32      }
33    }
34  }
35 });

```

If you do not have a Stripe account, get one at [stripe.com⁸⁴](http://stripe.com). Stripe is an incredibly developer friendly payment processing gateway, which makes it ideal for us building our ngroad marketplace on it.

Once you have an account, find your Account Settings page and locate the API Keys page. Find the publishable key (either the test one – which will not actually make charges or the production version) and take note of it.

⁸⁴<http://stripe.com>

In our `scripts/app.js` file, add the following line and replace the 'pk_test_YOUR_KEY' publishable key with yours.

```
1 // ...
2 .config(function(StripeServiceProvider) {
3   StripeServiceProvider
4     .setPublishableKey('pk_test_YOUR_KEY');
5 })
```

Using Stripe

When a user clicks on an image they like, we'll open a form in the browser that takes credit card information. We'll set the form to submit to an action on our controller called `submitPayment()`.

Notice above where we have the thumbnail of the image, we include an action when the image is clicked that calls the `sellImage()` action with the image.

Implementing the `sellImage()` function in the `MainCtrl`, it looks like:

```
1 // ...
2 $scope.sellImage = function(image) {
3   $scope.showCC = true;
4   $scope.currentItem = image;
5 }
6 // ...
```

Now, when the image is clicked, the `showCC` property will be true and we can show the credit card form. We've included an incredibly simple one here:

```
1 <div ng-show="showCC">
2   <form ng-submit="submitPayment()">
3     <span ng-bind="errors"></span>
4     <span>Card Number</span>
5     <input type="text"
6           ng-minlength="16"
7           ng-maxlength="20"
8           size="20"
9           data-stripe="number"
10          ng-model="charge.number" />
11     <span>CVC</span>
12     <input type="text"
13       ng-minlength="3"
```

```
14      ng-maxlength="4"
15      data-stripe="cvc"
16      ng-model="charge.cvc" />
17  <span>Expiration (MM/YYYY)</span>
18  <input type="text"
19      ng-minlength="2"
20      ng-maxlength="2"
21      size="2"
22      data-stripe="exp_month"
23      ng-model="charge.exp_month" />
24  <span> / </span>
25  <input type="text"
26      ng-minlength="4"
27      ng-maxlength="4"
28      size="4"
29      data-stripe="exp-year"
30      ng-model="charge.exp_year" />
31  <input type="hidden"
32      name="email"
33      value="user.email" />
34  <button type="submit">Submit Payment</button>
35 </form>
36 </div>
```

We're binding the form almost entirely to the `charge` object on the scope, which we will use when we make the charge.

The form itself submits to the function `submitPayment()` on the controller's scope. The `submitPayment()` function looks like:

```
1 // ...
2 $scope.submitPayment = function() {
3   UserService
4     .createPayment($scope.currentItem, $scope.charge)
5     .then(function(data) {
6       $scope.showCC = false;
7     });
8 }
9 // ...
```

The last thing that we'll have to do to be able to *take charges* is implement the `createPayment()` method on the `UserService`.

Now, since we're taking payment on the client-side, we're technically not going to be able to process payments, we can only accept the `stripeToken` which we can set a background process to manage handling turning the stripe tokens into actual payments.

Inside of our `createPayment()` function, we'll call our `StripeService` to generate the `stripeToken`. Then, we'll add the payment to an Amazon SQS queue so that our background process can make the charge.

First, we'll use the `AWSService` to access our SQS queues.

Unlike our other services, the SQS service requires a bit more integration to make it work as they require us to have a URL to interact with them. In our `AWSService` service object, we'll need to cache the URL that we're working with and create a new object every time using that object instead. The idea behind the workflow is the exact same, however.

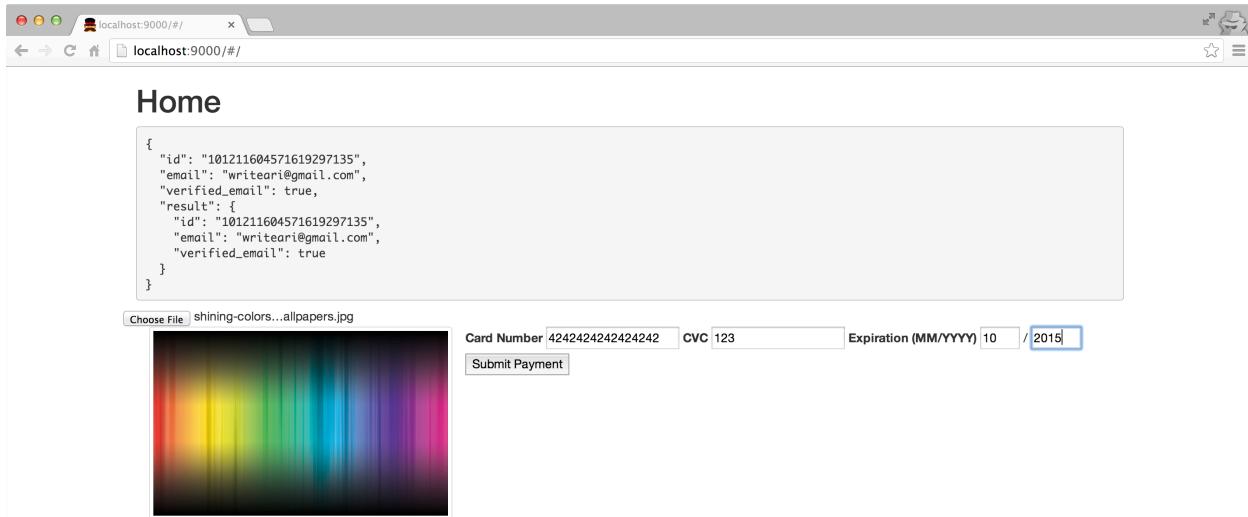
```
1 // ...
2 self.$get = function($q, $cacheFactory) {
3     var dynamoCache = $cacheFactory('dynamo'),
4     s3Cache = $cacheFactory('s3Cache'),
5     sqsCache = $cacheFactory('sqrs');
6 // ...
7 sqs: function(params) {
8     var d = $q.defer();
9     credentialsPromise.then(function() {
10        var url = sqsCache.get(JSON.stringify(params)),
11        queued = $q.defer();
12        if (!url) {
13            var sqs = new AWS.SQS();
14            sqs.createQueue(params,
15                function(err, data) {
16                    if (data) {
17                        url = data.QueueUrl;
18                        sqsCache.put(JSON.stringify(params), url);
19                        queued.resolve(url);
20                    } else {
21                        queued.reject(err);
22                    }
23                });
24        } else {
25            queued.resolve(url);
26        }
27        queued.promise.then(function(url) {
28            var queue =
29            new AWS.SQS({params: {QueueUrl: url}});
```

```
30     d.resolve(queue);
31   });
32 }
33 return d.promise;
34 }
35 // ...
```

Now we can use SQS inside of our `createPayment()` function. One caveat to the SQS service is that it can only send simple messages, such as with strings and numbers. It cannot send objects, so we'll need to call `JSON.stringify` on our objects that we'll want to pass through the queue.

```
1 // ...
2 ChargeTable: "UserCharges",
3 // ...
4 createPayment: function(item, charge) {
5   var d = $q.defer();
6   StripeService.createCharge(charge)
7   .then(function(data) {
8     var stripeToken = data.id;
9     AWSservice.sqs(
10       {QueueName: service.ChargeTable}
11     ).then(function(queue) {
12       queue.sendMessage({
13         MessageBody: JSON.stringify({
14           item: item,
15           stripeToken: stripeToken
16         })
17       }, function(err, data) {
18         d.resolve(data);
19       })
20     })
21   }, function(err) {
22     d.reject(err);
23   });
24 return d.promise;
25 }
```

When we submit the form...



Payment handling

Our SQS queue grows and we have a payment just waiting to be completed.

The screenshot shows the AWS SQS Management Console interface. At the top, there's a navigation bar with links for Services, Edit, and Help, along with user information for Ari D. Lerner and the N. Virginia region. Below the navigation is a toolbar with buttons for Show/Hide and Refresh. The main area is titled "Queues" and contains a table with one row. The table has columns for Name, Messages Available, Messages in Flight, and Created. The single entry is "UserCharges" with 3 messages available, 0 in flight, and created on 2013-11-05 02:40:51 GMT-08:00. There are also "Create New Queue" and "Queue Actions" buttons at the top of the table. At the bottom of the page, there's a copyright notice for Amazon Web Services, a link to Privacy Policy and Terms of Use, and a Feedback button.

SQS queue

Conclusion

The entire source for this section is available at <http://d.pr/aL9q⁸⁵>.

Amazon's AWS presents us with powerful services so that we can completely change the way we work and deploy our angular apps.

⁸⁵<http://d.pr/aL9q>

Testing

AngularJS is a framework that encourages writing clean and solid testable code. This is one of the most useful features of AngularJS that you get out of the box.

The Angular's team emphasis on testing is so strong that they built a test runner to make the process easier. In their words:

JavaScript is a dynamically typed language which comes with great power of expression, but it also comes with almost no help from the compiler. For this reason we feel very strongly that any code written in JavaScript needs to come with a strong set of tests. We have built many features into Angular which makes testing your Angular applications easy. So there is no excuse for not testing.

Why test?

When building any non-trivial application for any business purpose beyond prototyping (and even then) it's important to be confident about our code. When we have tests backing up our code-base, we can discretely know that our code is working as intended or not.

Bugs in our code are inevitable and without tests it's difficult to know where they are, thus tests make it easy to isolate and eliminate them. They make it easy to on-board other developers and provide working documentation about the code.

Testing is essential for understanding what is happening in our app.

Testing strategies

When building a testing suit to build Angular apps, it's always good to have a strategy for how and what we are going to test in our app. If we end up testing nothing or meaningless tests, we'll have nothing real in terms of tests. Conversely, if we test everything we can possibly think of, we'll end up spending more time writing tests and fixing minor bugs in our test code than we will on our app.

It's important to be realistic in terms of what value we will get out of the tests we do write and about what we should be testing.

At the end of the day, our tests will be a tool for us to gauge both the health of our app and a measuring stick to tell us if we've broken our code when introducing new functionality.

Getting started testing

One of the major hurdles in getting started with testing is having a test runner that runs tests on our code. JavaScript testing is also a bit more difficult as it requires us to build automation into browsers.

Building a development testing suite is difficult enough, but what about supporting continuous integration so that new deployments can be automated and we can be confident about the quality of the code before making a new one?

Continuous integration is a practice in software engineering of merging development working copies of a shared mainline several times a day and running the test suite upon update

Karma⁸⁶ is a testing tool that was built from the ground-up to remove the burden of setting up testing and allow us to focus on building our core application logic.

Karma spawns a browser instance (or multiple different browser instances) and runs the tests against the different instances to see if they pass different tests. Karma communicates with the browsers through socket.io, which enables karma to keep in constant contact. Thus, it provides real-time feedback as to what tests are running and gives us a human-readable output as to which tests pass and which ones fail or timeout.

Karma is capable of communicating with several different browsers natively and removes the need for us to need to manually test our code in different browsers. For example, it can run the tests in Chrome, IE and Firefox and spit out the results to your console. We can even hook up to our own native devices (yes, like an iphone or ipad) to test our code.

Types of AngularJS tests

There are several different ways to test our angular apps, depending upon what level of granularity we want to focus on and what features we want to target.

Unit testing

We can focus on building our tests to isolate specific, isolated components of our code. This is called “unit testing” where we test specific units of code for all sorts of different input, at different stages and under different conditions.

Unit testing is specifically for testing small, individual units of code, single functions or small and contained function interactions. It is *not* about testing large feature-sets.

A tricky part about unit testing is setting up the isolation of one piece of logic to be able to test it. We’ll discuss strategies of accomplishing this isolation in this chapter.

⁸⁶<http://karma-runner.github.io/0.10/index.html>

When Unit testing is the *right choice*

When we're writing our functional code, we're going to create little components of functionality. For instance, in building an application that handles live-filter of elements in a list, we're going to build the filtering functionality.

This specific filter functionality is a 'unit' of functionality that we'll need to implement this feature. To be confident that this filter functionality has been implemented and is working as expected, we'll need to isolate the component and test it for different inputs.

Imagine we're building a rocket ship. We'll want to test each individual part of the ship, the thrusters, the joystick controls, the oxygen system to verify the ship in general is working how we expect it to work.

E2E testing

On the other hand, we can *blackbox* test our application, otherwise known as end-to-end (or e2e) test where we test the application from the point of view that we are an end-user and know nothing about the underlying components of the system. This is great for testing large features of the application.

E2E testing works well for testing the user interaction with the page, without forcing us to refresh the page manually and test with the browser.

E2E testing is nothing new and there are great tools that enable us to set up automated browser testing. We can use tools like PhantomJS or CasperJS for headless browser testing (without opening a browser) or tools such as Karma that will open a real browser and perform all the tests in an iframe.

When End to End testing is the *right choice*

When we're writing features of functionality, it's always a good idea to write a test to walk the path of our user. End-to-end testing is great as it maps out the *real* experiences our users will take when using our application.

For instance, building a user login flow, we test that the user is logged in and redirected to their homepage. We don't worry about *how* the user is logged in, just that they are logged in and directed to the proper place.

Imagine we're building a rocket ship. End-to-end testing doesn't care about the engines or the landing gear, it cares that the rocket takes off and flies our astronauts to space.

Both Unit testing and E2E testing are supported out-of-the-box with the Karma test runner.



Note that writing unit tests instead of E2E tests will allow our tests to run super fast. Setting up our tests to run synchronously and using mocking libraries will greatly speed up our testing as well.

Getting started

In order to run our tests, we'll need to install the karma test runner. At this point in the book, you'll likely already have them available to you, but you'll need to make sure you have [NodeJS⁸⁷](#) and NPM installed. Once you do, to install the karma tool, we'll use the `npm` command:

```
1 $ npm install -g karma
```



To get npm installed, we'll be saving our dependencies in the package.json file that lists our dependencies. To set up package.json, with `npm` installed simply run `npm init` and walk through the wizard.

To start testing your application, you'll need to set up a reasonable structure for both your application code as well as for your test code.

The file structure that we recommend is storing your application files in the following format:

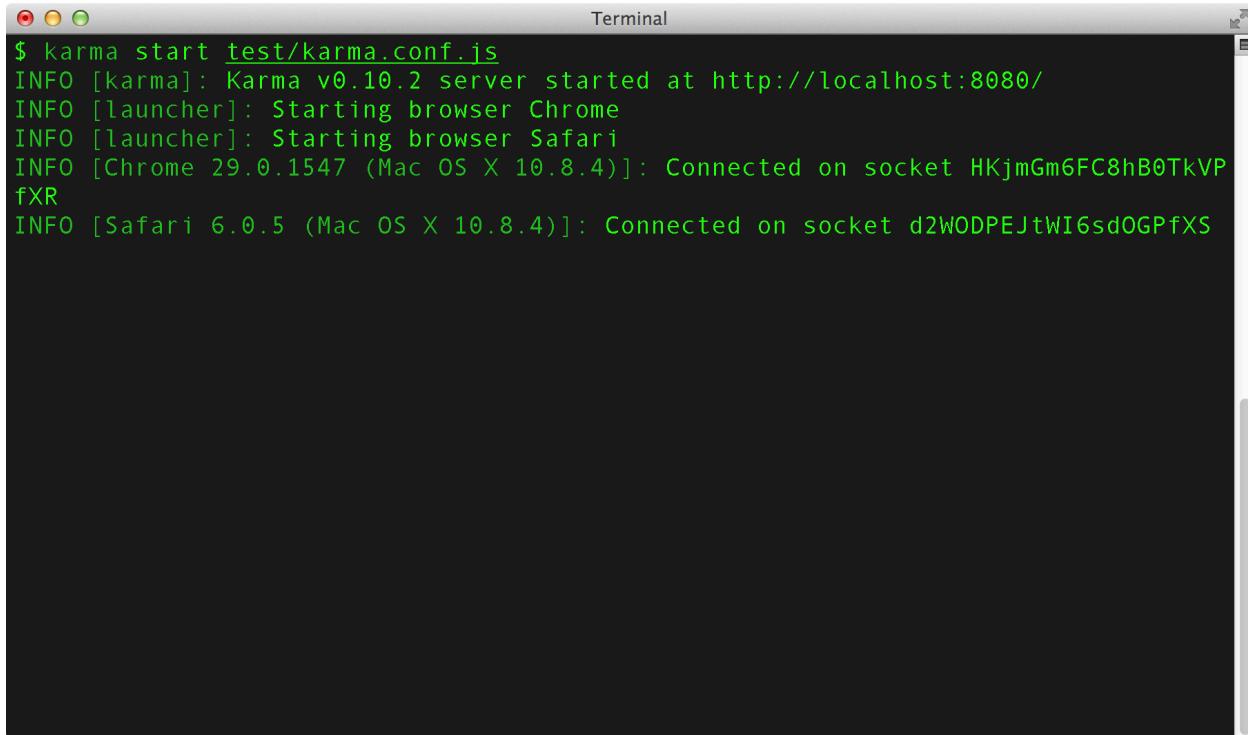
```
1 app/
2   index.html
3   js/
4     app.js
5     controllers.js
6     directives.js
7     services.js
8     filters.js
9   views/
10  home.html
11  dashboard.html
12  calendar.html
13 test/
14  karma-e2e.conf.js
15  karma.conf.js
16  lib/
17  angular-mocks.js
18  helpers.js
19  unit/
20  e2e/
```

The `app/` layout is standard, where we divide our application code. The `test/` directory has our tests nested in the appropriate directories that reflect the type of test, `unit/` or `e2e/`.

⁸⁷<http://nodejs.org>

There are two different karma configuration files in the test/ directory. Each of these files contains the specific type of test that they are going to be running. As we walk through each type of test, we'll discuss how each karma configuration should look and how to customize them for our use.

Running a karma test is simple: `karma start path/to/karma.config.js`. When the test runner starts up, it will start the browsers listed in the karma config file.

A screenshot of a Mac OS X terminal window titled "Terminal". The window shows the command \$ karma start test/karma.conf.js followed by several INFO log messages. The first message indicates the Karma server started at http://localhost:8080. Subsequent messages show the launcher starting Chrome and Safari, and the browser instances connecting to the socket. The text is in white on a black background.

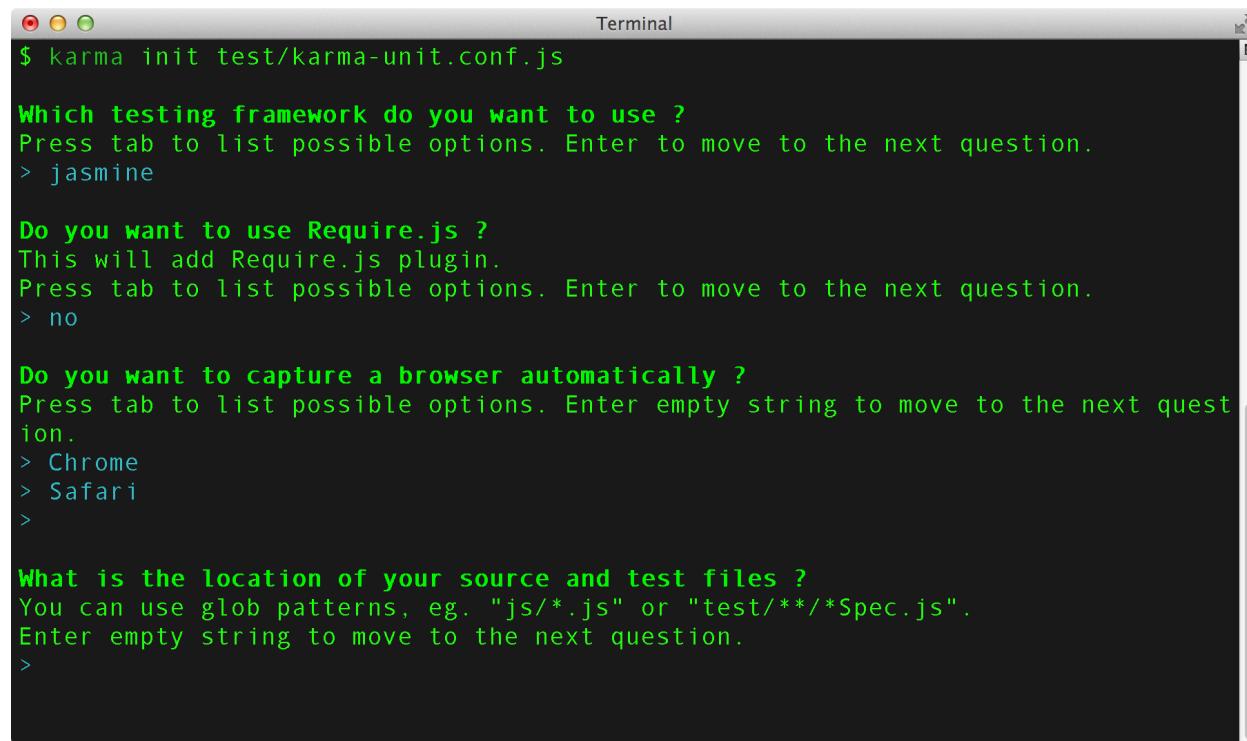
```
$ karma start test/karma.conf.js
INFO [karma]: Karma v0.10.2 server started at http://localhost:8080/
INFO [launcher]: Starting browser Chrome
INFO [launcher]: Starting browser Safari
INFO [Chrome 29.0.1547 (Mac OS X 10.8.4)]: Connected on socket HKjmGm6FC8hB0TkVP
fXR
INFO [Safari 6.0.5 (Mac OS X 10.8.4)]: Connected on socket d2WODPEJtWI6sdOGPfXS
```

Running karma with Chrome and Safari

By default, if not otherwise specified Karma will watch all the files listed in the karma configuration. Anytime that a file changes, karma will run the tests.

Initializing Karma config file

Karma gives us a generator to help us build configuration files. This generator will ask a few questions about how we want our configuration set up. Each question suggests a default value so it is possible to simply accept all the default values, which we'll do in a moment.

A screenshot of a Mac OS X terminal window titled "Terminal". The window shows the command \$ karma init test/karma-unit.conf.js followed by a series of questions from the karma generator:
Which testing framework do you want to use ?
> jasmine

Do you want to use Require.js ?
This will add Require.js plugin.
Press tab to list possible options. Enter to move to the next question.
> no

Do you want to capture a browser automatically ?
Press tab to list possible options. Enter empty string to move to the next question.
> Chrome
> Safari
>

What is the location of your source and test files ?
You can use glob patterns, eg. "js/*.js" or "test/**/*Spec.js".
Enter empty string to move to the next question.
>

Karma init

The process of setting up testing with unit tests and e2e tests is largely the same. We'll use the karma init generator to create the karma.conf.js files.

Setting up unit testing

First, run the karma init command with the path of your test file. In this case, we'll build our karma config in our tests directory:

```
1 $ karma init test/karma.conf.js
```

Unit tests need all of the dependencies available to run our tests against. When building our unit tests with the karma generator, it's important that our unit tests contain references to code for:

- a testing framework (choose one):
 - jasmine (default)
 - mocha
 - qunit
- custom test configuration (needed w/ mocha)
- any vendor required code
- our app-specific code

- our test code
- angular-mocks.js library for mocking

Unit tests need references to all of the app code that we'll be testing as well as all of the tests that we'll be writing.

For instance, a sample unit-test karma config file might look like. This is similar to the one we will generate, with comments removed for simplicity:

```
1 module.exports = function(config) {
2   config.set({
3     basePath: '..',
4     frameworks: ['jasmine'],
5     files: [
6       'lib/angular.js',
7       'lib/angular-route.js',
8       'test/lib/angular-mocks.js',
9       'js/**/*.js',
10      'test/unit/**/*.js'
11    ],
12    exclude: [],
13    port: 8080,
14    logLevel: config.LOG_INFO,
15    autoWatch: true,
16    browsers: ['Safari'],
17    singleRun: false
18  });
19};
```

Once this is set, we can run our unit tests like so:

```
1 $ karma run test/karma.conf.js
```

Alternatively, if you want the tests to run anytime the code changes (if you set autoWatch to true).

```
1 $ karma start test/karma.conf.js
```

Setting up e2e testing

To set up end to end testing, we'll run the karma generator with the path of our e2e karma config file.

```
1 $ karma init test/karma-e2e.conf.js
```

The e2e tests will be using the ng-scenario framework. Unlike unit tests, we do not need to reference all of our library code as the e2e tests run against our server. It simply needs to load all of the tests in the browser.

A sample karma config for e2e tests might look like:

```
1 module.exports = function(config) {
2   config.set({
3     basePath: '..',
4     frameworks: ['ng-scenario'],
5     files: [
6       'test/e2e/**/*.js'
7     ],
8     exclude: [],
9     port: 8080,
10    logLevel: config.LOG_INFO,
11    autoWatch: false,
12    browsers: ['Chrome'],
13    singleRun: false,
14    urlRoot: '/_karma_',
15    proxies: {
16      '/': 'http://localhost:9000/'
17    }
18  });
19};
```

Once this is set, we can run our unit tests like so:

```
1 $ karma run test/karma-e2e.conf.js
```

Alternatively, if you want the tests to run anytime the code changes (if you set autoWatch to true).

```
1 $ karma start test/karma-e2e.conf.js
```

Configuration options

Karma includes a lot of configuration options for you to choose and customize testing the way you like it.

framework

The generator will ask us which testing framework we'd like to use for tests. Jasmine is the *default* testing framework, although it supports Mocha, QUnit, Jasmine, and others by default.

These testing frameworks will require an additional `npm` library to be installed. For instance, to use the `jasmine` as a framework, you'll need to install the `jasmine` plugin.

```
1 $ npm install --save-dev karma-jasmine
```



Using the `--save-dev` flag to write the dependency to the `package.json` file and place it under `devDependencies`.

In the configuration file, this takes an array, which allows us to use multiple frameworks. Typically we'll only use one, so this is usually going to be set as `['jasmine']` or `['mocha']`.

For example:

```
1 frameworks: ['jasmine'],
```

RequireJS

If the project is using the [RequireJS⁸⁸](#) library, then select 'yes' for the question asking to include RequireJS. If your project does include it, then instead of listing all of the files in your karma config (which we'll see momentarily), you'll include your single test file which will be responsible for loading the specific modules.

RequireJS is a javascript file and module loader that's specifically designed for the browser. It enables us to write javascript libraries that we can export a library and use the name of the module to set up an expectation that it will be available when our module loads.

It's main benefits are:

- It sets up an import process
- It can load nested dependencies
- It enables easy packaging dependencies

In effect, it allows us to define javascript through modules and require those modules in our javascript. For instance:

⁸⁸<http://requirejs.org/>

```
1 define(['jquery', 'underscopre'],
2   function($, _) {
3     // $ references jQuery
4     // _ references underscore
5   });

```

For more information, see [RequireJS](#) for more information on how to set up testing.

Browser captures

The karma generator will ask which browsers you want to start automatically and capture their test results. Upon termination of the test runner, karma will shut down these browsers as well. You can also test any browser by opening the URL that karma's web server is listening on (defaults to `http://localhost:9876`), something worth keeping in mind if you want to test Internet Explorer from another machine(or VM) on your local network.

Browser captures require an additional plugins to be installed to launch and run the browsers using Karma. These plugins can be installed using `npm`. To enable Chrome, you'll need to install the chrome launcher plugin, like so:

```
1 $ npm install --save-dev karma-chrome-launcher
```

If you want to use safari, you'll need to install the safari launcher, firefox, the firefox launcher, and so on.

```
1 browsers: ['Chrome', 'Safari'],
```

Source and test files

The karma generator will ask where your javascript source and test files are located. This array can contain either simple strings and/or objects.

Strings can be patterns, such as `app/js/**/*.js` or specific file locations `app/js/vendor/angular/angular.js`. These are files and patterns that are relative to the `basePath`.

Files can also be specified using an object(instead of a string), which is useful when we want to configure certain aspects of a given file path or pattern. In the following example, this object tells karma to watch the file `public/js/watch-me.js` for changes but not to include the file on the page or serve it to a URL:

```
1 {pattern: 'public/js/watch-me.js', watched: true, included: false, served: false}
```

Note that the reason for using an object is to provide fine grain control over a file or file pattern, thus the `pattern` property is required. The other properties, such as `included` have defaults, and therefore only need to be set when your pattern deviates from the norm.

Let's discuss each option and its defaults in detail:

pattern The pattern to match files, this can be either a path to a single file or a pattern of files in the same manner that strings are listed above.

watched This boolean specifies that this file will be watched or not if karma is set up to use autoWatch. If it is listed as true, then the karma will run the tests when this file is changed. If it is set to false, then they will not be run.

If not listed as a property in the object then the files listed in this object will be watched by default (true).

included This boolean sets the file to be loaded using the `<script>` tag in the browser. If this is set to true, then the files will be loaded by the browser. If set to false then you are responsible for loading them manually. This is generally used in conjunction with [RequireJS](#).

By default, files are set to be included with the `<script>` tag (true).

served This boolean sets the file to be served by the karma webserver. If it is set to true, then the file will be accessible over the webserver. If it's set to false, then it will not be.

By default, this is set to true and thus the files will be set to be able to be fetched over the webserver.

Ordering

The order that files are listed matters, so we'll list our libraries before we list our application files as they'll need to be required. If a pattern is listed, then the files are sorted in alphabetical order and included.

Every file that is included is only included once, so if a file is matched more than once in a pattern match it is only included once.

An full example of the `files` property:

```
1 files: [
2   // simple strings
3   // we can target a single file
4   'js/app/vendor/angular/angular.js',
5   // or we can target a glob of files in a pattern
6   'js/app/*.js',
7   // objects
8   // when the index.html file changes, we won't
9   // run the tests
10  {pattern: 'public/index.html', watched: false},
11  // And we can set the file to not be included
12  // but still be watched
13  {pattern: 'public/index.html', included: false}
14 ]
```

exclude

It will ask if there are any files that you do not want to include in loading the tests. This array is useful if you are using RequireJS for example.

```
1 exclude: [
2   'public/index.html'
3 ]
```

basePath

The basePath is the root path location that will be used to resolve relative paths defined in the files and exclude properties. If the basePath is a relative path, then it will be resolved relative to where the karma configuration file is (`__dirname`).

```
1 basePath: '.',
```

autoWatch

Setting autoWatch to true will trigger karma to execute the tests whenever the files in files are changed. It is useful to set this to false when using a continuous integration server where watching file changes are unnecessary.

```
1 autoWatch: true,
```

captureTimeout

If the browser loads in more than the captureTimeout (defaults to 60 seconds or 60000 ms), then karma will kill the process and try again. If it fails after 3 tries, then karma will give up trying to launch the browser.

```
1 captureTimeout: 60000
```

colors

Karma's default output will include color. If you do not want to include color output in your terminal, you can disable this by setting the colors property to false:

```
1 colors: true,
```

hostname

The hostname by default is localhost, but if you want to change it, you can set the hostname property.

```
1 hostname: '127.0.0.1',
```

logLevel

When something goes wrong or unexpected with Karma, it's useful to look at more detailed information from Karma. We can set the level of detailed output by setting the `logLevel` property. This is also useful on the other end when using a continuous integration server, it can be optimal to disable output entirely.

The possible log values are:

- `config.LOG_DISABLE`
- `config.LOG_ERROR`
- `config.LOG_WARN`
- `config.LOG_INFO`
- `config.LOG_DEBUG`

```
1 LogLevel: config.LOG_INFO,
```

port

The default port for karma's webserver to launch and listen on is 9876. It's possible to customize the port by setting it in the config file.

```
1 port: 9875,
```

preprocessors

It's possible to tell Karma to preprocess files before the tests are run. This is useful for using a language like [Coffeescript⁸⁹](#) to write tests in and not need to process these files manually.

Preprocessors other than coffeescript, which is baked-in by default require additional plugins to run via `npm`.

The available preprocessors for Karma that are included are:

- `coffeescript`
- `html2js`

Other plugins are available as preprocessors and can be included by a plugin are:

- `coverage`
- `ng-html2js`
- `ember`

To include one or more of these, install them with the `npm` command:

⁸⁹<http://coffeescript.org>

```
1 $ npm install karma-coverage --save-dev
```

To configure which preprocessors are to be used, set them in the config file map. By default, this is set to `{ '**/*coffee': 'coffee' }`.

```
1 preprocessors: {
2   '**/*.coffee': ['coffee']
3 }
```

It's possible to configure some of the preprocessors as well. Configuration is dependent upon the plugin we're using. To configure coffeescript, for example:

```
1 coffeePreprocessor: {
2   options: { bare: true }
3 }
```

You can also customize the preprocessors using a `customPreprocessor` property:

```
1 customPreprocessor: {
2   mini_coffee: {
3     base: 'coffee',
4     options: { bare: true }
5   }
6 }
```

proxies

Karma will set up http proxies for us so that when we fetch a route it can fetch it off a remote server. This is useful and required for e2e tests (that use a server).

This object will be a list of key-value pairs that point from a path to a remote server.

```
1 proxies: {
2   '/': 'http://localhost:9000'
3 }
```

reporters

Reporting from Karma is able to be customized as well. It's possible to set reporters to set all sorts of output in the terminal to display useful information about the state of the tests.

By default, this is set to `['progress']`, which will report the progress of the tests in human readable form. By default, progress and dots are included by default as reporters for karma. It's possible to include other reporters as well through plugins.

You can include other reporters, such as `growl` and `coverage` via npm plugins. These plugins can be installed using `npm`:

```
1 $ npm install karma-[plugin-name] --save-dev
```

singleRun

If this boolean is set to true, then Karma will run the tests once for all the configured browsers and exit with an exit code of 0 if they all pass and 1 if any tests fail.

This is particularly useful when running the tests on a continuous integration server.

urlRoot

This is the base url where Karma runs. Any of the urls Karma uses gets prefixed with the `urlRoot` parameter. It's a good idea to use this when using a proxy so it doesn't collide with existing functions on the server.

Using RequireJS

To use RequireJS with Karma, we'll need an additional file after our `karma.conf.js` config file, the `test-main.js` file.

- `karma.conf.js` - responsible for configuring Karma (as we've seen)
- `test-main.js` - responsible for configuring Require.js for the tests

karma.conf.js

We'll configure `karma` like normal with the `karma` configuration file generator:

```
1 $ karma init test/karma.conf.js
```

When the question to use `Require.js` prompts to use `Require.js`, select yes.

When it asks which files you want to be loaded by default, you'll need to select all of the files that are *not* loaded by `Require.js`. It's safe to only include the `test/test-main.js` file, which we'll be creating shortly.

When we list the source and tests files, we're choosing all of the files we want to load with `Require.js`. We should list every file that we're loading with `Require.js`. This includes all external libraries, all of our code and all of our test files.

These files will need to be configured with the configuration object and set to not be included by default.

At this point, our `karma.conf.js` should be:

```
1 module.exports = function(config) {
2   config.set({
3     basePath: '..',
4     frameworks: ['jasmine', 'requirejs'],
5     files: [
6       {pattern: 'app/lib/angular.js', included: false},
7       {pattern: 'app/lib/angular-route.js', included: false},
8       {pattern: 'app/lib/angular-mocks.js', included: false},
9       {pattern: 'app/js/**/*.js', included: false},
10      {pattern: 'test/**/*.js', included: false}
11      {pattern: 'test/lib/**/*.js', included: false},
12      'test/test-main.js'
13    ],
14    exclude: [
15      'js/main.js'
16    ],
17    reporters: ['progress'],
18    port: 9876,
19    colors: true,
20    logLevel: config.LOG_INFO,
21    autoWatch: true,
22    browsers: ['Chrome'],
23    captureTimeout: 60000,
24    singleRun: false
25  });
26};
```



Notice that we're excluding our `main.js` file, the application file that starts the application.

Since karma serves files under the `app/js` directory, we're configuring karma's file server with a starting context for modules that load with a relative path. Since we want our `baseUrl` for our tests to be in the same folder as our source files, we'll need to set the `basePath` to the local directory `(.)`.

test/test-main.js

Our `test-main.js` file will operate as a substitute for our main application file, providing us with the ability to reference our test files without actually kicking off our app.

Karma will include all the files in the array `window.__karma__.files`, so we'll find our test files from here. After we've found our tests, we'll configure our RequireJS like normal:

```
1 var tests = [];
2 for (var file in window.__karma__.files) {
3     if (window.__karma__.files.hasOwnProperty(file)) {
4         if (/Spec\.js$/.test(file)) {
5             tests.push(file);
6         }
7     }
8
9 requirejs.config({
10     baseUrl: 'app',
11     paths: {
12         'jquery': 'lib/jquery',
13         'angular': 'lib/angular',
14         'angularRoute': 'lib/angular-route',
15         'angularMocks': 'lib/angular-mocks',
16     },
17     shim: {
18         'underscore': {
19             exports: '_'
20         },
21     },
22     // ask Require.js to load these files (all our tests)
23     deps: tests,
24     // start test run, once Require.js is done
25     callback: window.__karma__.start
26 });

```

Our tests will look a bit different from those that do not use RequireJS by default. We can simply use RequireJS like normal and wrap our tests in `define()`. For example:

```
1 define([
2     'app', 'jquery', 'angular',
3     'angular', 'angularRoute', 'angularMocks'
4 ],
5     function() {
6         describe('UnitTest: App', function() {
7             // just like normal
8             it('is defined', function() {
9                 expect(_.size([1,2,3])).toEqual(3);
10            });
11        });
12    });

```

Jasmine

In this book, we're going to cover the Jasmine testing framework. Although Karma does support multiple testing frameworks, the default option for Karma is Jasmine.

Jasmine is a *Behavior-driven* development framework for testing JavaScript code. Since we'll be working closely with the Jasmine syntax heavily, we'll run through an overview of how to write Jasmine-based test suites.

Spec suite

At the heart of a Jasmine suite is the `describe` function. This is a global function defined in the Jasmine suite, so we can call it directly from the test.

The `describe()` function takes two parameters, a string and a function. The string is a name or description of the spec suite we're setting up. The function encapsulates the test suite.

```
1 describe('Unit test: MainCtrl', function() {  
2 });
```

We can nest these `describe()` functions so that we can create a test tree that executes the different conditions that we'll set up throughout the tests:

```
1 describe('Unit test: MainCtrl', function() {  
2   describe('index method', function() {  
3     // Specs go in here  
4   });  
5 });
```

It's a good idea to group related specs to each other. We'll use the `describe()` function to do this. When each `describe()` block runs, the strings will be concatenated together along with the spec's name. Thus, the above example's title would become: "Unit test: MainController index method".

These `describe` block titles will then be appended to the spec title. This is specifically intended on allowing us to read our specs as full sentences so it's important that we name our tests in an English readable fashion.

Defining a spec

Defining a spec is done by calling the `it()` function. This too is a global function that is defined in the Jasmine test suite, so we can call it directly from our tests as well.

The `it()` function takes two arguments as well. A string title or description of the spec and a function that contains one or more expectations to test our code.

These expectations are functions that, when executed evaluate to true or false. A test with all true expectations is considered a passing spec, whereas a spec with one or more expectations that evaluate to false is considered a failing test.

A simple test might look like:

```
1 describe('A spec suite', function() {  
2   it('contains a passing spec', function() {  
3     expect(true).toBe(true);  
4   });  
5 });
```

This single spec's title, as appended to the `describe()` title becomes: "A spec suite contains a passing spec."

Expectations

When testing our app, we'll want to *assert* that conditions are how we expect them to be at different stages of the app. Tests we'll write will likely be able to be red like so "if we click on this button, then we *expect* this result". For instance, "If we navigate to the home page, then we *expect* the welcome message to be rendered."

Expectations are set up with the `expect()` function. The `expect()` function takes a single value argument. This argument is called the *actual* value.

To set up an expectation, we'll chain a *matcher* function that takes a single value argument. This argument is the *expected* value.

These matcher functions implements a boolean comparison between the actual value and the expected value. We can create a negation of the test by calling `not` before calling the matcher.

```
1 describe('A spec suite', function() {  
2   it('contains a passing spec', function() {  
3     expect(true).toBe(true);  
4   });  
5   it('contains another passing spec', function() {  
6     expect(false).not.toBe(true);  
7   });  
8 });
```

Jasmine comes built-in with a large set of matchers that we'll use throughout testing our app. It also is incredibly easy to write a custom matcher.

Included matchers

toBe

The `toBe()` matcher compares values with the javascript operator: `==`.

```
1 describe('A spec suite', function() {
2   it('contains passing specs', function() {
3     var value = 10,
4         another_value = value;
5     expect(value).toBe(another_value);
6     expect(value).not.toBe(null);
7   });
8 });
```

toEqual

The `toEqual()` matcher compares values and works for simple literals and variables.

```
1 describe('A spec suite', function() {
2   it('contains a passing spec', function() {
3     var value = 10;
4     expect(value).toEqual(10);
5   });
6 });
```

toMatch

The `toMatch()` matcher matches strings with a regular expression.

```
1 describe('A spec suite', function() {
2   it('contains a passing spec', function() {
3     var value = "<h2>Header element: welcome</h2>";
4     expect(value).toMatch(/welcome/);
5     expect(value).toMatch('welcome');
6     expect(value).not.toMatch('goodbye');
7   });
8 });
```

toBeDefined

The `toBeDefined()` matcher compares values against `undefined`.

```
1 describe('A spec suite', function() {
2   it('contains a passing spec', function() {
3     var value = 10,
4       undefined_value = undefined;
5     expect(value).toBeDefined();
6     expect(undefined_value).not.toBeDefined();
7   });
8 });
```

toBeUndefined

The `toBeUndefined()` matcher does the exact opposite as the `toBeDefined()` matcher.

```
1 describe('A spec suite', function() {
2   it('contains a passing spec', function() {
3     var value = 10,
4       undefined_value = undefined;
5     expect(undefined_value).toBeUndefined();
6     expect(value).not.toBeUndefined();
7   });
8 });
```

toBeNull

The `toBeNull()` matcher compares values against the null value.

```
1 describe('A spec suite', function() {
2   it('contains a passing spec', function() {
3     var value = null,
4       not_null_value = 10;
5     expect(value).toBeNull();
6     expect(not_null_value).not.toBeNull();
7   });
8 });
```

toBeTruthy

The `toBeTruthy()` matcher compares values for boolean casting of a truthy value.

```
1 describe('A spec suite', function() {
2   it('contains a passing spec', function() {
3     var value = 10,
4       undefined_value;
5     expect(value).toBeTruthy();
6     expect(undefined_value).not.toBeTruthy();
7   });
8 });
```

toBeFalsy

The `toBeFalsy()` matcher compares values for boolean casting testing of a falsy value.

```
1 describe('A spec suite', function() {
2   it('contains a passing spec', function() {
3     var value = 10,
4       undefined_value;
5     expect(undefined_value).toBeFalsy();
6     expect(value).not.toBeFalsy();
7   });
8 });
```

toContain

The `toContain()` matcher looks for an item in an array.

```
1 describe('A spec suite', function() {
2   it('contains a passing spec', function() {
3     var arr = [1,2,3,4];
4     expect(arr).toContain(4);
5     expect(arr).not.toContain(12);
6   });
7 });
```

toBeLessThan

The `toBeLessThan()` matcher mathematically compares a value to be less than the expected.

```
1 describe('A spec suite', function() {
2   it('contains a passing spec', function() {
3     var value = 10;
4     expect(value).toBeLessThan(20);
5     expect(value).not.toBeLessThan(5);
6   });
7 });
```

toBeGreaterThan

The `toBeGreaterThan()` matcher mathematically compares a value to be more than expected.

```
1 describe('A spec suite', function() {
2   it('contains a passing spec', function() {
3     var value = 30;
4     expect(value).toBeGreaterThan(40);
5     expect(value).not.toBeGreaterThan(20);
6   });
7 });
```

toBeCloseTo

The `toBeCloseTo()` matcher compares values to be close to based off a specific precision comparison.

```
1 describe('A spec suite', function() {
2   it('contains a passing spec', function() {
3     var value = 30.02;
4     expect(value).toBeCloseTo(30, 0);
5     expect(value).not.toBeCloseTo(20, 2);
6   });
7 });
```

toThrow

The `toThrow()` matcher tests if a function throws an exception or not.

```
1 describe('A spec suite', function() {
2   it('contains a passing spec', function() {
3     expect(function() {
4       return a + 10;
5     }).toThrow();
6     expect(function() {
7       return 2 + 10;
8     }).not.toThrow();
9   });
10});
```

Creating custom matchers

Jasmine makes it incredibly easy to create our own matchers for more complex situations in our code. To create a matcher, inside a jasmine block, we can call the `addMatcher()` function with a function that takes the `value`.

```
1 describe('A spec suite', function() {
2   this.addMatchers({
3     toBeLessThanOrEqual: function(expected) {
4       return this.actual <= expected;
5     }
6   });
7 });
```

We can then call this `toBeLessThanOrEqual()` matcher in any of our tests that are defined in the test suite.

Setup and teardown

Rather than setting up our test conditions manually every time in our tests, we can use the `beforeEach` method to run a group of setup functions. The `beforeEach()` function takes a single argument, a function that is called once before each spec is run. This can be used in a `describe` block, like so:

```
1 describe('A spec suite', function() {
2   var message;
3   beforeEach(function() {
4     message = "hello ";
5   });
6   it('should say hello world', function() {
7     expect(message + "world").toEqual("hello world");
8   });
9   it('should say hello ari', function() {
10    expect(message + "ari").toEqual("hello ari");
11  });
12});
```

We can also reset conditions, such as clear a database or flush all requests from a mock using the `afterEach()` function. Similarly to the `beforeEach()` function, it takes a single argument, a function to be executed after each spec is run.

```
1 describe('A spec suite', function() {
2   var count;
3   afterEach(function() {
4     count = 0;
5   });
6   it('should add one to count', function() {
7     count += 1;
8     expect(count).toEqual(1);
9   });
10  it('should check for the reset value', function() {
11    expect(count).toEqual(0);
12  });
13});
```

These `beforeEach` and `afterEach` methods are chained when inside nested `describe` blocks, so we can set up complex test trees without duplicating our code.

End to end introduction

When we are end-to-end testing we are going to use the Angular scenario runner. The Angular scenario runner simulates user interactions so that we can more accurately assess the status of the application.

When we write scenario tests, we'll describe how the application should behave from different stages. Just like in unit testing, we'll be using Jasmine to set up our expectations and behavior.

We will be working directly with the scenario runner API to control our browsers as they work through the application tests. The API allows us to run the browser through many various actions, including entering data into input fields, selecting elements, navigating the browser, controlling browser flow and more.

The core fundamental API method we'll use is the `browser()` method. This method returns an object that we'll be able to chain methods to control the browser.

The scenario runner works by opening a browser window with an iframe embedded inside. This iframe is where karma runs our app tests and tracks the success or failures of the scenario runner.

Navigating pages

To load a url into the test browser frame, we'll use the `navigateTo` function that takes a single argument: the url to load.

```
1 browser().navigateTo(url)
```

We can also dynamically load a url by calling a function to find the URL. This is common for cases when we don't know the destination url when we're writing a test, checking the outcome of a certain action, for instance.

```
1 browser().navigateTo(title, function() {  
2   // return the dynamic url here;  
3   return '/';  
4 });
```

Reload the page

We can refresh the currently loaded page in the test frame:

```
1 browser().reload()
```

Handling the window object

We can get the current href of the currently loaded page in the test frame:

```
1 browser().window().href()
```

To get the path of the currently loaded page in the test frame:

```
1 browser().window().path()
```

To get the current search of the page loaded in the test frame:

```
1 browser().window().search()
```

We can get the latest hash of the currently loaded page in the test frame:

```
1 // The hash is returned without the #
2 browser().window().hash();
```

Location location location

To get the current \$location.url() of the page loaded in the test frame:

```
1 browser().location().url()
```

We can get the \$location.path() of the currently loaded page in the test frame:

```
1 browser().location().path()
```

It's also easy to get the \$location.search() of the current page:

```
1 browser().location().search()
```

Finally, we can get the hash of the current page as well:

```
1 browser().location().hash()
```

Setting expectations

To actually verify that our application does work the way we expect it to act, we'll need to set up expectations that assert a certain state. We can do this with a combination of the e2e and scenario APIs.

Using expect(), we will *assert* that the value of a given *future* object matches the matcher. Anything given back by the scenario API is a future object that the scenario runner will resolve and we're validating that the eventual value will result in what is expected.

```
1 expect(browser().location().path())
2   .toBe('/')
3 // Or negate the expectation with not()
4 expect(browser().location().path())
5   .not().toBe('/home')
```

Interacting with the content

End-to-end testing is particularly powerful as we are *actually* loading the page our users will see, so we can peek into the result that they see and verify that it looks right and works according to our expectations.

We can select elements, enter values into input fields, click on buttons, verify content is where it should be, run through repeaters, etc.

To select an element on a page, we'll use the `element()` API method. This API method takes two parameters:

- selector - the jQuery HTML element selector
- label - a string of text used for output in the browser or terminal

```
1 element("form", "the signup form")
```

With this element selected, we can execute methods to query the state of the it on the page.

To check the number of elements that match a certain jQuery selector:

```
1 element("input", "input elements").count()
```

To click on an element, for instance on a submit button, we can call:

```
1 element("button", "submit button").click()
```

We can run a function on a certain jQuery selector using the `query()` function.

```
1 // select all links on the page
2 element("a", "all links").query(
3   // all of the links will be passed to the
4   // function as elements
5   function(elements, done) {
6     // Do what we'd like with each element
7     angular.forEach(elements, function(ele) {
8       expect(ele.attr('ng-click'))
9         .toBeDefined();
10    });
11    done(); // Tell the scenario runner we are done
12  });

```

We can look at each element and set different expectations on the jQuery attributes.

We can fetch or set the value of an element:

```
1 element("button", "submit button").val()
2 // Or set
3 element("button", "submit button").val("Enter")
```

We can fetch or set the text:

```
1 // The text of a block of html
2 element("h1", "header").text()
3 // Or set
4 element("h1", "header").text("Header text")
```

We can get or fetch the html of an element

```
1 // HTML of the html
2 element("h1", "header").html()
3 // Or set
4 element("h1", "header").html("<h2>New header</h2>")
```

To set or fetch the height

```
1 // Height of an element
2 element("div", "signup box").height()
3 // To set
4 element("div", "signup box").height('200px')
```

To fetch or set the innerHeight:

```
1 // innerHeight of an element
2 element("div", "signup box").innerHeight()
3 // To set
4 element("div", "signup box").innerHeight('190px')
```

To set or fetch the outerHeight:

```
1 // outerHeight of an element
2 element("div", "signup box").outerHeight()
3 // To set
4 element("div", "signup box").outerHeight('210px')
```

To set or fetch the width:

```
1 // width of the element
2 element("div", "signup box").width()
3 // Setting
4 element("div", "signup box").width('300px')
```

To set or fetch the innerWidth:

```
1 // innerWidth of the element
2 element("div", "signup form").innerWidth()
3 // Setting
4 element("div", "signup form").innerWidth('200px')
```

To set or fetch the outerWidth:

```
1 // outerWidth of the element
2 element("div", "signup form").outerWidth()
3 // Setting
4 element("div", "signup form").outerWidth('305px')
```

To set or fetch the position of the element:

```
1 // The position of the element
2 element(".logo", "our logo").position()
3 // Or
4 element(".logo", "our logo").position("absolute")
```

To get or set the scrollLeft:

```
1 // The scrollLeft value of the element
2 element("#signup_form", "signup form").scrollLeft()
3 // Setting
4 element("#signup_form", "signup form").scrollLeft(0)
```

To get or set the scrollTop value, where you can force the browser to scroll to a specific element:

```
1 // The scrollTop value of the element
2 element("#signup_form", "signup form").scrollTop()
3 // Setting
4 element("#signup_form", "signup form").scrollTop(0)
```

To fetch or set the offset

```
1 // The element's offset
2 element("#signup_form", "signup form").offset()
3 // Setting
4 element("#signup_form", "signup form").offset(0);
```

We can also query and/or change the value of an element in a jQuery selector. We can get an attribute (with attr):

```
1 element("div", "signup box").attr('width')
2 // To set it
3 element("div", "signup box").attr('width', '100%')
```

We can fetch a property (with prop)

```
1 element("div", "signup box").prop('width')
2 // To set it
3 element("div", "signup box").prop('width', '100%')
```

And we can fetch css (with css):

```
1 element("div", "signup box").css('border-color')
2 // To set it
3 element("div", "signup box").css('border-color', 'red')
```

We can interact with the content in different ways, other than simply by fetching the element using `element()`. Angular's scenario runner includes a few different helper methods that enable us to both query and interact with the rendered DOM.

We can introspect into angular's understanding of different elements that we're interested in. We can select them, find bindings, interact with input elements, and query the page for testing native angular bindings.

Selecting elements on the page

One of the lowest level helpers the scenario runner sets up for us is the `using()` function. The `using()` method allows us to target specific elements using jQuery-style element selectors.

```
1 it('does not test anything yet', function() {
2     // Target a specific element
3     using('.input_email').binding('email');
4 });
```

The `using()` method takes up to two parameters:

jQuery selector This is the selector we'll use to select the element on the page.

Label (string optional) This is a label that the runner will use to identify the selector in the output of our tests.

Interacting with the angular binding

The scenario runner includes a way for us to query into the bindings set up by angular. This enables us to query into our angular bindings on the DOM and select the *first* binding for the specific element.

For instance, if we have the HTML, where the property `name` on the `$scope` element is available on the DOM:

```
1 <input type="text" ng-model="name" />
```

We can query for this specific binding in the scope by using the `binding()` method:

```
1 it('should update the name', function() {
2   using('.form').input('name').enter('Ari');
3   expect(
4     using('.form').binding('name')
5   ).toBe('Ari');
6 });
```

The `binding()` method takes a single argument:

- name (string)

This is the name of the binding for the DOM element we're interested in querying.

Interacting with input elements

We can interact with input elements on our page as well. If we want to input text in a text box, checking a checkbox or selecting a value for an option element, we can use the `input()` method.

The `input()` method itself returns an object that is intended on allowing us to call methods to interact with the element. It takes a single argument:

- name (string)

The name of the corresponding `ng-model` name.

The available methods that can be called on the input field are:

enter() The `enter()` method enters a value into an input field.

Given the HTML:

```
1 <input type="text" ng-model="name" />
```

We can enter 'Ari' into the input with:

```
1 input('name').enter('Ari');
```

check() The `check()` method will check a checkbox input field.

Given the HTML:

```
1 <input type="checkbox" ng-model="save" />
```

We can check the save checkbox by calling:

```
1 input('name').check();
```

select() The `select()` method will select a given value for a radio button.

Given the HTML:

```
1 <input type="radio" ng-model="color" value="red" />
2 <input type="radio" ng-model="color" value="blue" />
3 <input type="radio" ng-model="color" value="yellow" />
```

We can select the radio button with the following test:

```
1 input('color').select('red');
```

val() Lastly, we can get the current value of the input field simply by calling `.val()` on the input element. We'll use this to check the current value of the specific input element.

```
1 input('color').select('red');
2 input('color').val(); // This will be "red"
```

Option inputs

It's also easy to select an option for a given option input field. We'll use the `select()` method to enable us select one option over another option in a selection tag.

Given the HTML: {lang="html"}

```
1 select('color')
```

The `select()` method returns an object with a method to select a single option from the select input as well as one with the ability to select multiple inputs for multi-selects.

option() The `option()` method allows us to select a single value from the list.

```
1 select('color').option('red');
```

The `option()` method takes a single argument:

- value (string)

The value argument is a single string that selects the input with the given value.

options() The `options()` method allows us to select multiple values from a multi-select option.

```
1 select('color').options('Ghostbusters', 'Titanic');
```

The `options()` method takes any number of arguments to select the values from the option field as necessary:

- values (list of strings)

The values to select from the multiselect.

Repeating repeating elements elements

Angular makes it incredibly easy to build DOM elements for lists through the `ng-repeat` directive and the angular scenario enables us to test these repeating directives easily.

The `repeater()` function itself returns an object with several methods we can use to query the different list of elements. It accepts up to two arguments:

- selector (string)

The jQuery selector that points to the selector of the element's we are interested in.

- label (optional string)

The label is a string that is used in test outputs.

The available methods that we can call on the list of elements returned by the repeater are as listed below. For each of these tests, we'll use the following HTML as example HTML:

```
1 <table id="phonebook">
2   <tr ng-repeat="person in people">
3     <td>{{ person.name }}</td>
4     <td>{{ person.email }}</td>
5   </tr>
6 </table>
```

The methods are as follows:

count() The `count()` method returns the number of rows in the repeater that match the jQuery selector in the DOM.

```
1 repeater('#phonebook tr').count();
```

The `count` method takes no arguments and will simply return a single integer.

column() The `column()` method returns an array with the values in the column with the given binding in the repeater that matches the jQuery selector in the DOM.

```
1 repeater('#phonebook tr')
2   .column('person.name');
```

The `column()` method takes a single argument:

- binding (string)

This is the binding for the specific element of the repeater. This is the name of the binding that is rendered in the element.

row() The `row()` method returns an array with the bindings in the row at a specific index in the repeater that matches the given jQuery selector.

```
1 repeater("#phonebook tr").row(0);
```

The `row()` method takes a single argument:

- index (integer)

The `index` is the number of the row to return the given bindings.

Mocking and test helpers

Before we can start writing tests, we need to understand a core feature of testing: **mocking**. Mocking in testing is an old concept that allows us to define simulated objects that mimic the behavior of real objects under controlled circumstances.

AngularJS provides its own mocking library (called `angular-mocks` and available as the `angular-mocks.js` file). Mocking objects are specifically designed to be used in unit testing.

To set up a mock object in a unit test, we need to make sure we're including the `angular-mocks.js` file in our karma configuration.

Ensure that your `test/karma.conf.js` file contains the `angular-mocks.js` in the `files` array. Once we have the dependency included, we can create a mock reference to the angular module.

For example in this common unit test setup, we'll create a `describe` execution context where we call `angular.mock.module` before every test runs in our `describe` context:

```
1 describe('myApp', function() {  
2   // Mock our 'myApp' angular module  
3   beforeEach/angular.mock.module('myApp'));  
4  
5   it('...')  
6 });
```

Note that we can also just call `module` because the function `angular.mock.module` is published on the `window` interface for global access.

After we have set up our mock angular module, we can *inject* any of our services connected to that module into our test code.

With the tests, we'll need to *inject* the dependencies just as Angular will at runtime. In our unit tests, this is necessary as it's important that we isolate the functionality we want to test.

To inject a dependency, we'll use the `angular.mock.inject` method in a `beforeEach` function call similarly to how we did so above.

```
1 describe('myApp', function() {
2   var scope;
3
4   // Mock our 'myApp' angular module
5   beforeEach(module('myApp'));
6   beforeEach(module.inject(function($rootScope) {
7     scope = $rootScope.$new();
8   }));
9   it('...')
10});
```

Similarly to the `module` function, the `inject` function is also available on the `window` object for global access, so we can just call `inject`.

In this test, like most all of our unit tests, we'll want to save a reference to an instance of the object we are working with (in the above example, we're saving `scope`). This way we can work with the reference of the object in all of our `it()` clauses.

Often times we'll want to store the reference as the same name as we're injecting into our test. For example, if we are testing a service we can inject the service and store a reference to the service using a slightly different naming scheme. We'll enclose the injected service in `_` underscores which will cause the injector to ignore the name when it's injected.

```
1 describe('myApp', function() {
2   var myService;
3
4   // Mock our 'myApp' angular module
5   beforeEach(module('myApp'));
6   beforeEach(module.inject(function(_myService_) {
7     myService = _myService_;
8   }));
9   it('...')
10});
```

Mocking the `$httpBackend`

Angular also comes built-in with an `$httpBackend` mocking library so that we can mock any external XHR requests in our app and can avoid making expensive `$http` requests in our tests.

The `$httpBackend` service is a *Fake HTTP backend* implementation that allows us to isolate and specify conditions that external servers will put our app and we can determine how exactly our app behaves.

Using the `$httpBackend`, we can verify a request gets made, stub responses, stub calls and set assertions that verify how we expect our app to behave based on the response of the remote server. We'll use the `$httpBackend` solely in unit tests.



It is possible to use the `$httpBackend` service in end-to-end testing, but doing so will generally not test the app fully because we are no longer using our real server.

Testing with the `$httpBackend` works simply by hijacking the dependency injection chain where we'll inject the mock `$httpBackend` instead of the real `$httpBackend` service that makes the actual http requests by the `$http` service. In this way, we don't need to change our app at all to support testing it.

Flushing HTTP requests

When in production, the `$httpBackend` responds to requests asynchronously, which is fundamentally difficult to set up in testing environments. Thus, we will need to manually flush any pending requests at the end of our tests so that we can clean the execution environment, yet still keep the asynchronous behavior of the `$httpBackend`.

The `$httpBackend` has two methods for setting up handling http responses by a mock backend system. These two methods are the `expect` and `when` methods and have different use-cases.

Typically, in a unit test we'll want to ensure that all of the requests we set up with expectations are run at the end of each test and throw an exception if they don't. Additionally, we'll want to ensure that there are no outstanding requests that are pending at the end of each test.

We can take care of these two cases with two methods in an `afterEach` block:

```
1 // ...
2 afterEach(function() {
3   $httpBackend.verifyNoOutstandingExpectation();
4   $httpBackend.verifyNoOutstandingRequest();
5 });


```

There are cases when we want to reset all of the request expectations we've set. These cases occur when we want to reuse the same instance of `$httpBackend` when inside of a multiple-phase test.

We can reset them with the `resetExpectations()` method:

```
1 // ...
2 it('should be a multiple-phase test', function() {
3   // ...
4   $httpBackend.resetExpectations();
5   // ...
6 });

});
```

expect

The `expect` method sets up a request expectation and is used to make assertions about the requests made by the application and define responses for them. The test will fail if the expected requests are not made or they are incorrectly made. These are used primarily to set up an assertion that a request has been made.

The `expect` method takes two required arguments with two additional optional arguments:

- method

The string HTTP method, like ‘GET’ or ‘POST’

- url

The HTTP url where we’re expecting the call

- data (optional)

The HTTP request body or function that receives a data string and returns true if the data is expected, or a javascript object to send the body in JSON format.

- headers (optional)

The HTTP headers or function that will receive the header object and return true if the headers match the expectation.

The `expect` method returns an object with a `respond` method that controls how the matched request is handled inside the tests.

```
1 describe('Remote tests', function() {
2   var $httpBackend, $rootScope, myService;
3
4   beforeEach(inject(
5     function(
6       _$httpBackend_, _$rootScope_, _myService_) {
7       $httpBackend = _$httpBackend_;
8       $rootScope = _$rootScope_;
9       // myService is a service that makes HTTP
10      // calls for us
11      myService = _myService_;
12    }));
13
14  it('should make a request to the backend', function() {
15    // Set an expectation that myService will
16    // send a GET request to the route
17    // /v1/api/current_user
18    $httpBackend.expect('GET', '/v1/api/current_user')
19      .respond(200, {userId: 123});
20    myService.getCurrentUser();
21    // Important to flush requests
22    $httpBackend.flush();
23  });
24});
```

The `expect` method has several methods that are more descriptive of the expectation:

`expectGET()` creates a new request expectation for a GET method. `expectGET()` takes two arguments:

- url - a HTTP url
- headers - (optional) HTTP headers.

```
1 // ...
2 $httpBackend.expectGET("/v1/api/current_user")
```

`expectHEAD()` creates a new request expectation for a HEAD method. It accepts two arguments:

- url - a HTTP url
- headers - (optional) HTTP headers.

```
1 // ...
2 $httpBackend.expectHEAD("/v1/api/current_user")
```

`expectJSONP()` creates a new request expectation for a JSONP request. It accepts a single argument:

- url - a HTTP url

```
1 // ...
2 $httpBackend.expectJSONP("/v1/api/current_user")
```

`expectPATCH()` creates a new request expectation for PATCH requests. It accepts three arguments:

- url - a HTTP url
- data - (optional) HTTP request body or a function that receives the data string and returns true if the data is as expected or a JSON object
- headers - (optional) HTTP headers.

```
1 // ...
2 $httpBackend.expectPATCH("/v1/api/current_user")
```

`expectPOST()` creates a new request expectation for POST requests. It accepts three arguments:

- url - a HTTP url
- data - (optional) HTTP request body or a function that receives the data string and returns true if the data is as expected or a JSON object
- headers - (optional) HTTP headers.

```
1 // ...
2 $httpBackend.expectPOST("/v1/api/sign_up", { 'userId': 1234});
```

`expectPUT()` creates a new request expectation for PUT requests. It takes three arguments:

- url - a HTTP url
- data - (optional) HTTP request body or a function that receives the data string and returns true if the data is as expected or a JSON object
- headers - (optional) HTTP headers.

```
1 // ...
2 $httpBackend.expectPUT("/v1/api/user/1234", { 'name': 'Ari'});
```

`expectDELETE()` creates a new request expectation for DELETE requests. It takes two arguments:

- url - a HTTP url
- headers - (optional) HTTP headers.

```
1 // ...
2 $httpBackend.expectDELETE("/v1/api/user/123")
```

requestHandler

Our `expect()` methods all return a `requestHandler` object that has a single function: `respond`. The `respond` method gives us the ability to set up a response for the mocked HTTP request.

The `requestHandler` `response` function is a function that can take one of two forms.

The first form allows us to set a response code, response data, headers, or all three.

```
1 // ...
2 $httpBackend.expectGET("/v1/api/current_user")
3   // Respond with a 200 status code
4   // and the body "success"
5   .respond(200, 'Success');
6   // Or only return data
7   .respond("Fail");
8   // Or only headers
9   .respond({ 'X-RESPONSE', 'Failure'});
```

The second form enables us to set a request handler function that gets executed upon the successful execution of the request. Instead of returning data, we'll return a function handler that can return an array that contains the response status code, response data, and response headers

```
1 // ...
2 $httpBackend.expectGET("/v1/api/current_user")
3   // Respond with a 200 status code
4   // and the body "success"
5   .respond(function(method, url, data, headers) {
6     return [200, "DATA", {"header1": "Header1"}];
7   });
```

when

The `$httpBackend` also has the `when` method that differs from the `expect` method in that it doesn't create an expectation for a request at all. In fact, its purpose is primarily to create a fake backend for an app and return fake data.

Unlike expectations, when using `when()`, every single request that matches the URL can be handled by a single `when` definition. Additionally, a response is required when using `when`, whereas with `expect` a response is not required.

The `when()` method is great for setting up backend definitions that are common for all tests, for instance when testing a controller that is using the `resolve` property that depends upon foreign data being loaded.

The `when()` function takes two required arguments with two additional optional requirements:

- method

The string HTTP method, like 'GET' or 'POST'

- url

The HTTP url where we're expecting the call.

- data (optional)

The HTTP request body or function that receives a data string and returns true if the data is expected, or a javascript object to send the body in JSON format.

- headers (optional)

The HTTP headers or function that will receive the header object and return true if the headers match the expectation.

```
1 // ...
2 $httpBackend.when('GET', "/v1/api/current_user")
3   // Respond with a 200 status code
4   // and the body "success"
5   .respond(200, 'success');
```

Similar to the expect method, we have the same helper methods that make the use of when more descriptive.

whenGET() creates a new backend definition for a GET method. whenGET() takes two arguments:

- url - a HTTP url
- headers - (optional) HTTP headers.

```
1 // ...
2 $httpBackend.whenGET("/v1/api/current_user")
3   .respond(200, {userId: 123});
```

whenHEAD() creates a new backend definition for a HEAD method. It accepts two arguments:

- url - a HTTP url
- headers - (optional) HTTP headers.

```
1 // ...
2 $httpBackend.whenHEAD("/v1/api/current_user")
3   .respond(200);
```

whenJSONP() creates a new backend definition for a JSONP request. It accepts a single argument:

- url - a HTTP url

```
1 // ...
2 $httpBackend.whenJSONP("/v1/api/current_user")
3   .respond({userId: 123});
```

whenPOST() creates a new backend definition for POST requests. It accepts three arguments:

- url - a HTTP url
- data - (optional) HTTP request body or a function that receives the data string and returns true if the data is as expected or a JSON object
- headers - (optional) HTTP headers.

```
1 // ...
2 $httpBackend.whenPOST("/v1/api/sign_up",
3   {'userId': 1234})
4   .respond(200);
```

`whenPUT()` creates a new backend definition for PUT requests. It takes three arguments:

- url - a HTTP url
- data - (optional) HTTP request body or a function that receives the data string and returns true if the data is as expected or a JSON object
- headers - (optional) HTTP headers.

```
1 // ...
2 $httpBackend.whenPUT("/v1/api/user/1234", {'name': 'Ari'});
```

`whenDELETE()` creates a new backend definition for DELETE requests. It takes two arguments:

- url - a HTTP url
- headers - (optional) HTTP headers.

```
1 // ...
2 $httpBackend.whenDELETE("/v1/api/user/123")
3   .respond(200);
```

Testing an app

With our testing harness setup, we can start testing the different components of our app. Any part of our module that has logic that may change are good candidates for testing. Does a *route* work the way we expect, does the page contain specific content, does the controller code execute, etc.

We're going to focus on testing different components of our app, the most common places to test our app as well as tips and tricks on how to test the various components.

We'll cover testing the following components of our app:

- Routes
- Requests and page content
- Controllers
- Services & Factories

- Filters
- Templates and views
- Directives
- Resources
- Animations

For each component, we'll look at our options for testing and then we'll run through how to test it with the available methods.

For most all of our tests, our base code that we'll start with looks like:

```
1 describe('NAME', function() {  
2 });
```

Testing routes

When we test our routes, we want to set up a test that ensures the route that we're in is routed properly by our app. We'll need to check where the route works, if it's found or if it's a 404. We'll check if the routing events get fired and we'll check if the template that we expect is actually loaded.

In testing routing, we can use either unit testing or end-to-end testing. Since our routes will change page locations (URL) and page content, we'll need to check if the route has been loaded, if the page has been found and what's in-between.

To test these routes, we'll assume we have the simple routing code set up that looks like:

```
1 angular.module('myApp', ['ngRoute'])  
2   .config(function($routeProvider) {  
3     $routeProvider  
4       .when('/', {  
5         templateUrl: 'views/main.html',  
6         controller: 'HomeCtrl'})  
7       .when('/login', {  
8         templateUrl: 'views/login.html',  
9         controller: 'LoginCtrl'})  
10      .otherwise({redirectTo '/'});  
11  })
```

Unit testing routes

In order to set up our unit tests so that they can test our routing code, we'll need to do a few things:

- Inject the \$route, \$location, and \$rootScope services

- Set up a mock backend to handle XHR fetching template code
- Set a location and run a \$digest lifecycle

We'll store a copy of our three services we'll use in our tests: location, route, and rootScope so we can later reference these services in our tests.

```
1 describe('Routes test', function() {
2   // Mock our module in our tests
3   beforeEach(module('myApp'));
4
5   var location, route, rootScope;
6   beforeEach(
7     inject(_$location_, _$route_, _$rootScope_) {
8       location = _$location_;
9       route = _$route_;
10      rootScope = _$rootScope_;
11    });
12    // Our test code will go here
13  });
```

Now that we have our services injected into the controller, we'll set a mock backend to handle the angular fetching of the templates from the templateUrl. We'll use the \$httpBackend to create an expectation that we are expecting the template to be fetched:

```
1 describe('Routes test', function() {
2   // Mock our module in our tests
3   beforeEach(module('myApp'));
4
5   var location, route, rootScope;
6   beforeEach(inject(
7     function(_$location_, _$route_, _$rootScope_) {
8       location = _$location_;
9       route = _$route_;
10      rootScope = _$rootScope_;
11    }));
12  describe('index route', function() {
13    beforeEach(inject(
14      function($httpBackend) {
15        $httpBackend.expectGET('views/home.html')
16          .respond(200, 'main HTML');
17      }));
18    // Our tests will code here
```

```
19 });
20 });
```

With our test code all set up, we can start writing our tests. In order to test the router with unit tests, we'll need to mimic how the router works in production. The router works with the [digest life-cycle](#) where after the location is set, it will take a single digest loop cycle to process the route, transform the page content, and finish the routing. Knowing this, we'll need to account for the change in paths in our test.

Inside our test, we're going to test two states of the application in the index route:

- When a user navigates to the index page, they are shown the index page with the proper controller
- When a user navigates to an unknown route, they are taken to the index page as defined by our `otherwise` function.

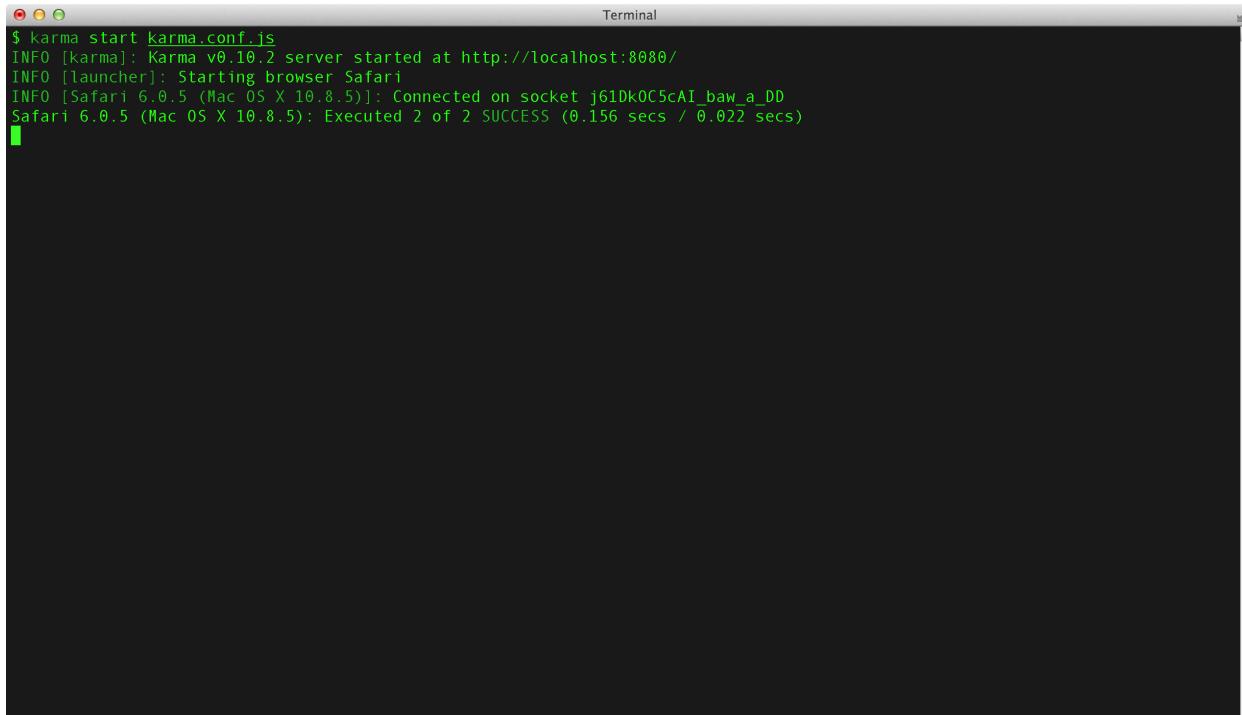
We can test the conditions by setting up our `$location` service to pass the paths. In order to trigger the location request, we'll run a digest cycle (on the `$rootScope`) and check that the controller is as expected (in this case, the 'HomeCtrl').

```
1 it('should load the index page on successful load of /',
2   function() {
3     location.path('/');
4     rootScope.$digest(); // call the digest loop
5     expect(route.current.controller)
6       .toBe('HomeCtrl')
7   });
8 it('should redirect to the index path on non-existent
9   route', function() {
10   location.path('/definitely/not/a/_route');
11   rootScope.$digest();
12   expect(route.current.controller)
13     .toBe('HomeCtrl')
14 });
```

To run these tests, make sure you have your grunt server running:

```
1 $ cd myApp
2 $ grunt server
```

Run `karma start karma.conf.js` in your app file and you will see immediate output in our terminal.

A screenshot of a Mac OS X Terminal window titled "Terminal". The window shows the command \$ karma start karma.conf.js followed by several lines of green INFO log output from Karma and the browser launcher. The log includes details about the Karma server starting at http://localhost:8080, the browser launching, and test execution results. The terminal has a dark background with light-colored text and standard OS X window controls.

```
$ karma start karma.conf.js
INFO [karma]: Karma v0.10.2 server started at http://localhost:8080/
INFO [launcher]: Starting browser Safari
INFO [Safari 6.0.5 (Mac OS X 10.8.5)]: Connected on socket j61Dk0C5cAI_baw_a_DD
Safari 6.0.5 (Mac OS X 10.8.5): Executed 2 of 2 SUCCESS (0.156 secs / 0.022 secs)
```

Unit testing routes

We did a lot of work to set up our routes test and we only tested one route location. Since we are testing flow changes for our user features, we can shuffle this work off to the application and test this more rigorously in end-to-end testing.

End-to-end route testing

With end-to-end testing, we don't need to mock any part of angular as we're black-box testing the app. In this way, we can just describe how we want our app to behave and write our tests accordingly.

In writing our end-to-end tests, we'll think about how the user navigates around our application. Our tests should be readable in that we're sending our user to a particular page and describing what they should experience in our app.

Our base test for all of our end-to-end tests will simply be:

```
1 describe('E2E: NAME', function() {
2   // Our tests will go here
3 });
```

That's it. We're going to use the `browser()` API function to modify the source of the iframe inside our browser.

To test our index route, we'll point our browser to the index route and ensure that the location is in-fact at the index page.

```
1 describe('E2E: Routes', function() {  
2   it('should load the index page', function() {  
3     browser().navigateTo('/#/');  
4     expect(browser().location().path()).toBe('/');  
5   });  
6 });
```

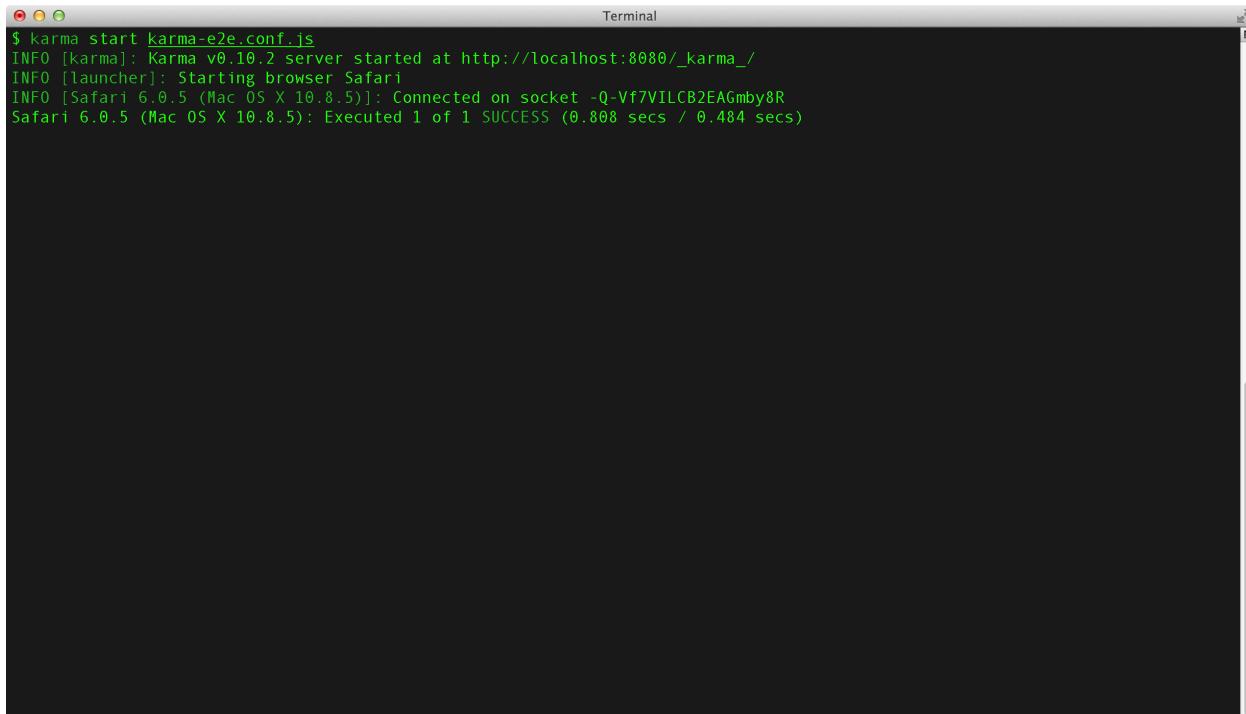
To run this test, we'll need to make sure that our grunt server is running:

```
1 $ cd myApp  
2 $ grunt server
```

Now we'll run karma:

```
1 $ karma start karma-e2e.conf.js
```

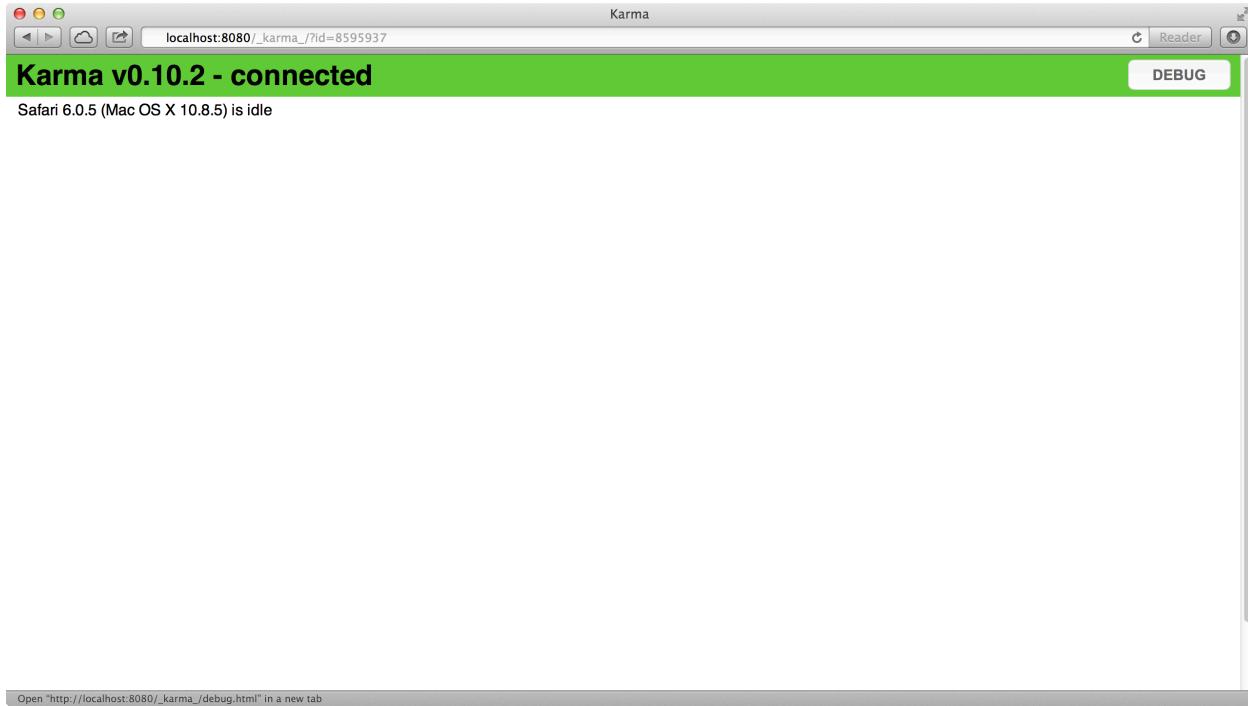
In here, we'll see immediate output in our terminal. If the test is successful, you'll see that it passes all of it's tests. If not, it will report it's failures to you.



```
$ karma start karma-e2e.conf.js  
INFO [karma]: Karma v0.10.2 server started at http://localhost:8080/_karma_/  
INFO [launcher]: Starting browser Safari  
INFO [Safari 6.0.5 (Mac OS X 10.8.5)]: Connected on socket -Q-Vf7VILCB2EAGmby8R  
Safari 6.0.5 (Mac OS X 10.8.5): Executed 1 of 1 SUCCESS (0.808 secs / 0.484 secs)
```

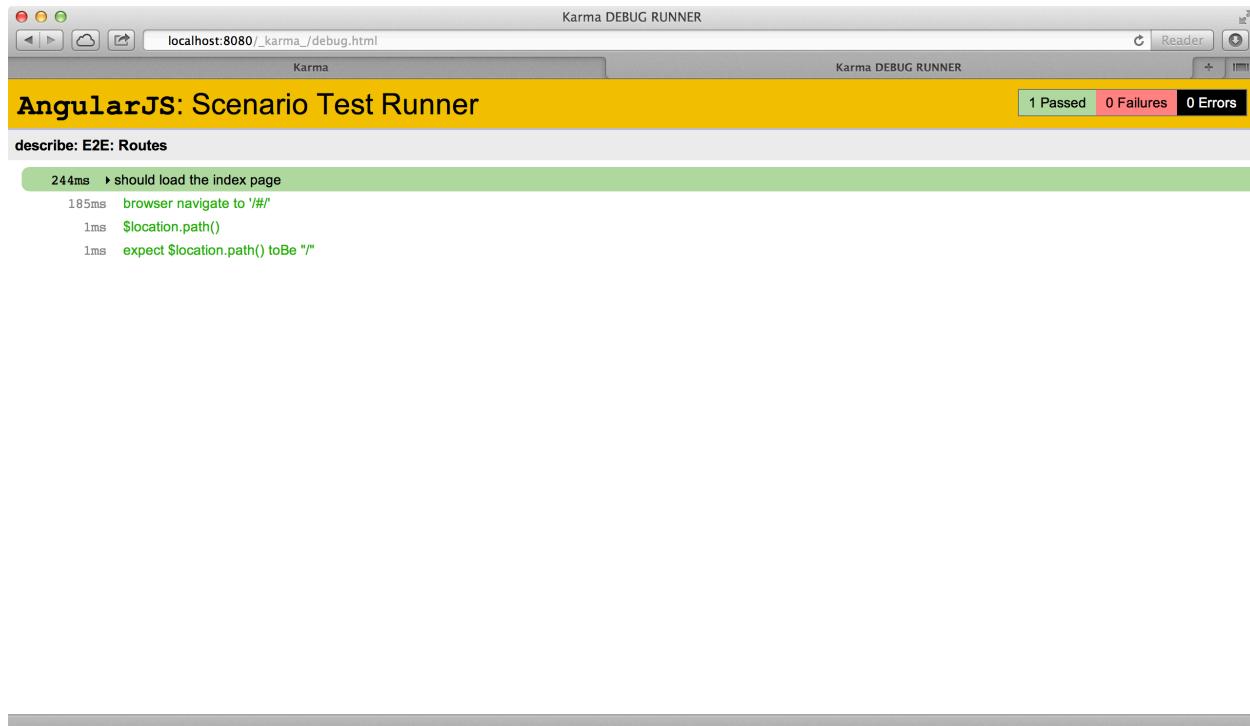
End-to-end testing routes in our terminal

With end-to-end tests, we can also use the browser to debug our tests. When we started karma, a browser opened in the background. Open the browser and click on the debug button in the top right corner:



Debug button in the browser

This will open a new page that shows us all of our tests, a list of the ones that pass, and a list of the ones that fail. It's useful when we're developing our tests to use our browser as reference as we're debugging our application and tests.



Visual representation of our tests in-browser

Testing page content

When we're testing the page content gets rendered correctly to our browser, we need to assert that certain content gets delivered in the browser to our users.

Unit testing content in the browser won't give us much detail as to the state of the application as we don't have direct access to the browser content in unit testing.

We can confirm that the controllers are executing the expected functions and set assertions that confirm the content will be loaded, which we will cover in-depth.

End-to-end testing page content

End-to-end testing, on the other hand is ideal for setting up expectations for loaded HTML. With end-to-end testing, every part of the application responsible for executing a successful browser request will fire.

To set up end-to-end content testing, we'll set up our test like normal:

```
1 describe('E2E: Content', function() {
2 });
```

We're going to set up two scenarios in this route test:

- That we have a link to the login route at our index page
- That we can click on the link and be brought to the login page

In our first test, we'll simply test that there IS a login button on our page. We'll set an assertion that we have a login button to click.

In our `index.html`, suppose we have the following content:

```
1 <div id="authorize">
2   <a id="login" class="radius" href="#/login">Try it! Sign in</a>
3 </div>
```

Inside of this first test, we'll just confirm that we have the element that matches the text in the button:

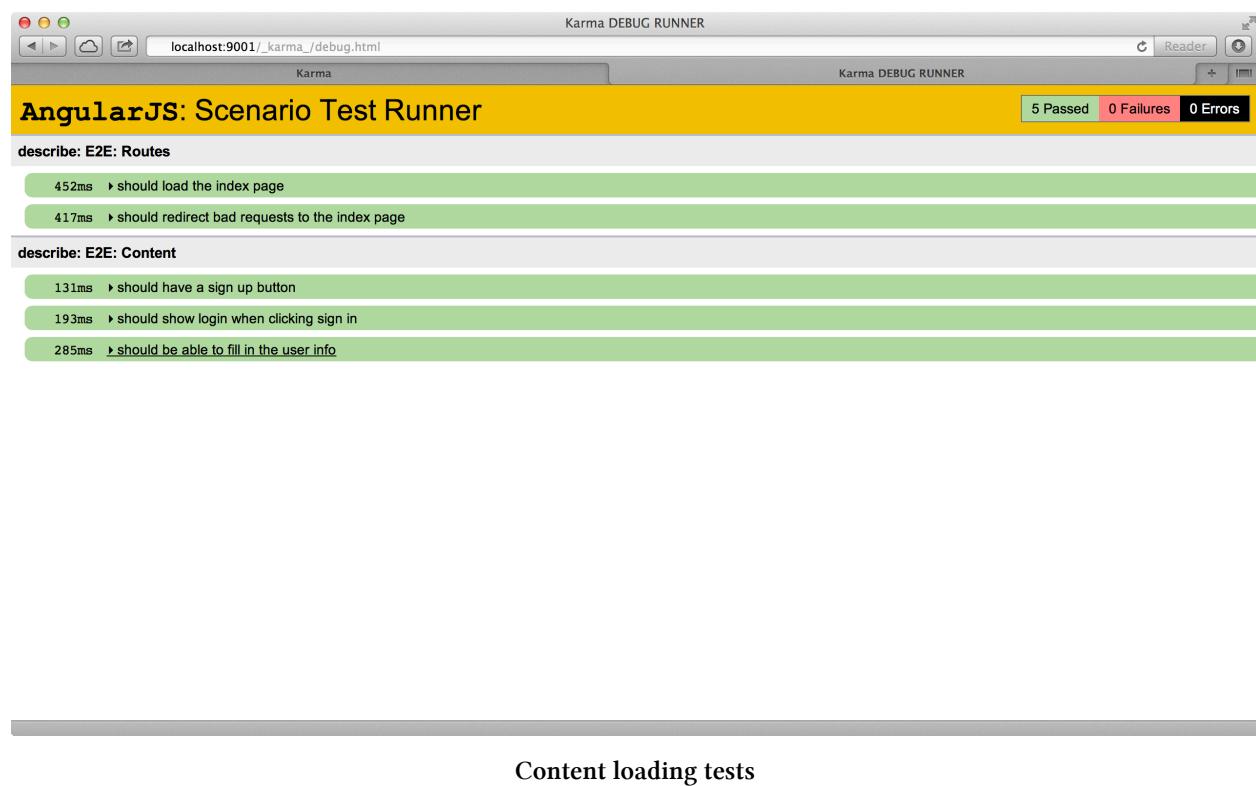
```
1 it('should have a sign up button', function() {
2   browser().navigateTo('/');
3   expect(
4     element("a#login").html()
5   ).toEqual("Try it! Sign in");
6 });
```

Now, we want to ensure that when our users click on this link, they are brought to the login page. We'll set up another tests that asserts if we click on the link that our new location is at the login route:

```
1 it('should show login when clicking sign in', function() {
2   browser().navigateTo('/');
3   element("a#login", "Sign in button").click();
4   expect(browser().location().path())
5     .toBe('/login');
6 });
```

Lastly, if we had a user to tests against, we can set up our test to fill out our login form simply by selecting the input elements and setting their value.

```
1 it('should be able to fill in the user info',
2   function() {
3     browser().navigateTo('/#/');
4     element("a#login", "Sign in button").click();
5     input("user.email").enter("ari@fullstack.io");
6     input("user.password").enter('123123');
7     element('form input[type="submit"]').click();
8     expect(browser().location().path())
9       .toBe('/dashboard');
10  });
});
```



Testing controllers

The business logic of our application is contained in our controllers. This is where the \$scope marries the controller to the view. Since we're going to do most of the work of updating the views of our application in our controllers, we can set up tests to ensure that their behavior is does execute as expected.

Unit testing controllers

When we're unit testing controllers, we'll need to set up our tests to mimic the behavior of Angular.

Setting up our unit tests we'll need to make sure we:

- Set up our tests to mock the module
- Store an instance of the controller with an instance of a known scope
- Test our expectations against the scope

To instantiate a new controller instance, we'll need to create a new instance of a scope from the \$rootScope with the \$new() method. This will set up the scope inheritance that angular uses at runtime.

With this scope, we can instantiate a new controller and pass the scope is as the \$scope of the controller.

```
1 describe('Unit controllers: ', function(){
2   // Mock the myApp module
3   beforeEach(module('myApp'));
4   describe('FrameCtrl', function() {
5     // Local variables
6     var FrameController, scope;
7     beforeEach(inject(
8       function($controller, $rootScope) {
9         // Create a new child scope
10        scope = $rootScope.$new();
11        // Create a new instance of the FrameController
12        FrameController = $controller('FrameCtrl',
13          { $scope: scope });
14      }));
15
16      // Our tests go here
17    });
18  });
```

With our test set up, we have both an instance of our FrameController as well as the \$scope for the controller. Now we can use that scope to test the scope on the FrameController.

In our FrameController we have a clock that ticks with the current time at the top of the app. We also have access to a user and their timezone.

The relevant parts of the controller code look like:

```
1 angular.module('myApp.controllers', [])
2   .controller('FrameCtrl',
3     ['$scope', '$timeout', function($scope, $timeout) {
4       $scope.time = {
5         today: new Date()
6       };
7       $scope.user = {
8         timezone: 'US/Pacific'
9       }
10      var updateClock = function() {
11        $scope.time.today = new Date();
12      };
13      var tick = function() {
14        $timeout(function() {
15          $scope.$apply(updateClock);
16          tick();
17        }, 1000);
18      }
19      tick();
20    }]);
```



Our FrameController is purposefully simplified for focus on the test. To see the entire test suite for the sample app, see the code that accompanies this book.

We'll test two features of our controller:

- The time is defined
- The user is defined and has a timezone

```
1 // Testing the FrameController values
2 it('should have today set', function() {
3   expect(scope.time.today).toBeDefined();
4 });
5
6 it('should have a user set', function() {
7   expect(scope.user).toBeDefined();
8 });
```

End-to-end testing controllers

The result of end-to-end testing controllers will look very similar to that of our tests where we test the page content renders as expected as we're testing that all of the functions in the controller actually fire.

To test if a page has been rendered or not, we can call the page browser in question in the browser and test if the content we expect has been rendered in the view.

Our boilerplate for our end-to-end tests looks like:

```
1 describe('E2E controllers: ', function() {  
2   // Our tests go here  
3 });
```

With our test set up, we can add our specs. We're going to test that the date (or at least a portion of it) exists on the page as well as the timezone.

```
1 beforeEach(function() {  
2   browser().navigateTo('/#/');  
3 });  
4  
5 it('should have the date in the browser', function() {  
6   var d = new Date();  
7   expect(  
8     element("#time h1").html()  
9   ).toMatch(d.getFullYear());  
10});  
11  
12 it('should have the user timezone in the header', function() {  
13   expect(  
14     element('header').html()  
15   ).toMatch('US/Pacific');  
16});
```

It is convenient to know if our timeout function has been called inside our controller. We can confirm that we are in fact setting the timeout to be called using the jasmine helper `createSpy`.

If we modify our `beforeEach()` function from above, we can include the `$timeout` service in our controller.

```

1 var FrameController, scope, timeout;
2 beforeEach(inject(
3   function($controller, $rootScope) {
4     scope = $rootScope.$new();
5     timeout = jasmine.createSpy('timeout');
6     FrameController = $controller('FrameCtrl', {
7       $scope: scope,
8       $timeout: timeout
9     });
10   }));

```

Now, in our tests, we can set an expectation that the service does in fact get called:

```

1 it('should set the clock a foot', function() {
2   expect(timeout).toHaveBeenCalled();
3 });

```

Testing services & factories

Services are easy to test as they are isolated objects that offer localized functionality. Since they are singleton objects, we can create these objects in isolation and test their behavior.

Unit testing services

Unit testing services are pretty easy as we only need to inject our services into our tests.

Starting with the simple example, suppose we have the service that provides a version:

```

1 angular.module('myApp.services', [])
2   .value('version', '0.0.1');

```

In this case, our *service* provides a single string value. We can inject the version service into the current test.

Our test boilerplate for this unit test looks like:

```

1 describe('Unit: services', function() {
2   beforeEach(module('myApp'));
3 });

```



Up until now, we've been implicitly calling the `$injector` service. This example shows how to use it explicitly.

To isolate the test, we'll nest this test into a `describe()` block and inject our `version` service in a `beforeEach()` block.

```
1 describe('version', function() {
2   var version;
3   beforeEach(inject(function($injector) {
4     // use the $injector to get the version service
5     version = $injector.get('version');
6   }));
7
8   it('should have the version as a service',
9    function() {
10    // set our expectation on the version service
11    expect(version).toEqual('0.0.1');
12  });
13});
```

As we can see, testing services is really simple. However, our services are not always this simple. In our sample app, we're interacting with the googleApi. This service is a bit more complex.

The full source for the googleServices.googleApi service:

```
1 // Our google services module
2 angular.module('googleServices', [])
3   .factory('googleApi',
4     ['$window', '$document', '$q', '$rootScope',
5      function($window, $document, $q, $rootScope) {
6        // Create a defer to encapsulate the loading of
7        // our Google API service.
8        var d = $q.defer();
9
10       // After the script loads in the browser, we're going
11       // to call this function, which in turn will resolve
12       // our global defer which enables the
13       $window.bootGoogleApi = function(keys) {
14         // We need to set our API key
15         window.gapi.client.setApiKey(keys.apiKey);
16         $rootScope.$apply(function() {
17           d.resolve(keys);
18         });
19       };
20
21       // Load client in the browser
22       var scriptTag = $document[0].createElement('script');
23       scriptTag.type = 'text/javascript';
24       scriptTag.async = true;
```

```
25     scriptTag.src = 'https://apis.google.com/js/client:plusone.js?onload=onLoad\\
26 Callback';
27     var s = $document[0].getElementsByName('body')[0];
28     s.appendChild(scriptTag);
29
30     // Return a singleton object that returns the
31     // promise
32     return {
33         gapi: function() { return d.promise; }
34     }
35 }]);
```

The service itself returns us an object with a single function that returns a promise that will be resolved once the google API has been loaded and is ready on the page.

To set out our expectations for this service, we'll need to use the jasmine method `spyOn` to create a spy on the method we're calling as well as set up an expectation that it is actually called.

We'll set up our tests to run against the google api in an isolated `describe()` block:

```
1 describe('googleServices', function() {
2     var googleApi, resolvedValue;
3
4     beforeEach(inject(function($injector) {
5         // Fetch the defined googleApi from our service
6         googleApi = $injector.get('googleApi');
7         // Create a spy for the gapi function
8         // that tells us it's been called, but doesn't
9         // prevent the actual function from being called
10        spyOn(googleApi, 'gapi')
11            .andCallThrough();
12        // Use the actual function's resolve to
13        // set the resolved value
14        googleApi.gapi().then(function(keys) {
15            resolvedValue = keys;
16        });
17    }));
18
19    describe('googleApi', function() {
20        // Our tests go here
21    });
22});
```

We can also inject the `googleApi` service above by name as the `inject()` function uses the same syntax as the `$injector`. The above call would change to: `inject(function(googleApi))` instead.

Note: by using the `andCallThrough()` method, our test will wait for the `gapi` object to be present on the window. We can stub these requests by using a different method `andCallFake()`.

```
1 var q;
2 beforeEach(inject(function($injector) {
3   // Fetch the defined googleApi from our service
4   googleApi = $injector.get('googleApi');
5   // Get the $q object
6   q = $injector.get('$q');
7   // Create a spy for the gapi function
8   // and mock the actual response
9   spyOn(googleApi, 'gapi')
10  .andCallFake(function() {
11    var d = q.defer(); // fake the deferred function
12    setTimeout(function() {
13      resolvedValue = {
14        clientId: '12345'
15      }
16    }, 100);
17    return d.promise;
18  });
19  // Use the actual function's resolve to
20  // set the resolved value
21  googleApi.gapi().then(function(keys) {
22    resolvedValue = keys;
23  });
24 }));
```

Now, we can use our spy to determine if the function has actually been called.

Our first test in this `describe()` block we will simply test that the method exists and is a function. This is useful if we're still working on the api and we change the method name or signature.

To set up tests for waiting for a promise to resolve, we'll use a jasmine helper: `waitsFor()`. It takes a single parameter, a function that provides a true/false response for when the method can continue. We'll set it up to wait a maximum of 1/2 a second at most:

```
1 describe('googleApi', function() {
2   beforeEach(function() {
3     // pause the spec for up to
4     // half a second while we are waiting for
5     // the resolvedValue to be resolved
6     waitsFor(function() {
7       return resolvedValue !== undefined;
8     }, 500);
9   });
10
11  it('should have a gapi function', function() {
12    expect(
13      typeof(googleApi.gapi)
14    ).toEqual('function');
15  });
}
```

Now we can set expectations for the return values of the service and assert that they are equal to our assumptions:

```
1 it('should call gapi', function() {
2   expect(googleApi.gapi.callCount)
3     .toEqual(1);
4 });
5
6 it('should resolve with the browser keys', function() {
7   expect(resolvedValue.clientId)
8     .toBeDefined();
9 });
```

End-to-end testing services

Since services interact with our front-end through our controllers, it's not effective to test services specifically with end-to-end testing. We can, however test that services resolve their promises and the results populate the view.

For instance, we can test that a list of events gets populated into the view by a service. For instance, in an /events page, where we are showing a list of events we can assert that we actually are listing out the number of events we expect with the details that we expect:

```
1 beforeEach(function() {
2   browser().navigateTo('/#/events');
3 });
4
5 it('should show 10 events', function() {
6   expect(
7     repeater('.event_listing li').count()
8   ).toBe(10);
9 });
```

Testing filters

Filters are also easy to test as they are isolated functionality. The filter's job is specifically to limit or manipulate output, so we'll set assertions on the output of the filter functions.

Unit testing filters

Unit testing filters is simple. First, we'll need to get access to the filter. We can do this simply by injecting the \$filter service into our tests. This will give us access to looking up the filter in the process:

```
1 describe('Unit: Filter tests', function() {
2   var filter;
3
4   // Mock our module in our tests
5   beforeEach(module('myApp'));
6   beforeEach(inject(function($filter) {
7     filter = $filter;
8   }));
9 });
```

With this access to the filter, now it's simply a matter of setting expectations on the output of the filter.

```
1 it('should give us two decimal points',
2   function() {
3     expect(filter('number')(123, 2)).toEqual('123.00');
4   });
5});
```

End-to-end testing filters

We can also test the output of our filters in the view using End to End testing. End to end testing is slightly different than using unit tests to test our code as we're focused on what the end-user sees rather than the output of our filter function specifically.

In order to set up a filter test, we'll have our browser load the page(s) where we are testing our filter and we'll interact with the filter itself.

For instance, given the case where we have a live-search with an ng-repeat:

```

1 <input ng-model="search.$" type="text" placeholder="Search filter" />
2 <table id="emailTable">
3   <tbody>
4     <tr ng-repeat="email in emails | filter:search.$">
5       <td>{{ $index + 1 }}</td>
6       <td>{{ email.from }}</td>
7       <td>{{ email.subject | capitalize }}</td>
8     </tr>
9   </tbody>
10 </table>
```

Where our *emails* data looks like:

```
[ { from: 'ari@fullstack.io', subject: 'ng-book and things' }, { from: 'ari@fullstack.io', subject: 'Other things about ng-book and angular' }, { from: 'ted@google.com', subject: 'Conference speaking gig' } ];
```

Here, we'll test two filters, the live-search filter where we set an expectation against the ng-repeat as well as on the subject line (to ensure it's capitalized).

To ensure that the live-search is working, we'll enter a value into the input field and ensure that the field changes to what we expect.

```

1 it('should filter on the search', function() {
2   expect(repeater('#emailTable tbody tr')).count()
3     .toBe(3);
4   // things shows up in two of our emails
5   input('search.$').enter('things');
6   expect(repeater('#emailTable tbody tr')).count()
7     .toBe(2);
8 });
```

We can see that when we set a different input for the `search.$` field, the ng-repeat changes from 3 to 2.

We can also test our own filter (the `capitalize` filter) to ensure it's working as expected:

```
1 it('should capitalize the subject line', function() {
2   expect(
3     repeater('#emailTable tbody tr:first')
4       .column('email.subject')
5     ).toEqual(["Ng-book and things"]);
6 });
});
```

Testing templates

When we test templates, we're looking to ensure that the proper content template is loaded and that the correct data inside the template shows up in the view.

Unit testing templates

Since our templates are tied directly to the view, unit testing components *inside* the view doesn't make a lot of sense as we're making assertions upon the completion of multiple components to build the view.

We can make an assertion that the template is loaded properly. To do this, we'll need to set up our test to expect a request to the home template as well as execute a view change to test that it does, in fact get loaded.

```
1 describe('Unit: Templates', function() {
2   var $httpBackend,
3     location,
4     route,
5     rootScope;
6
7   beforeEach(module('myApp'));
8   beforeEach(inject(function(_$rootScope_, _$route_, _$httpBackend_, _$location_)\n9   {
10     location = _$location_;
11     rootScope = _$rootScope_;
12     route = _$route_;
13     $httpBackend = _$httpBackend_;
14   }));
15
16   afterEach(function() {
17     $httpBackend.verifyNoOutstandingExpectation();
18     $httpBackend.verifyNoOutstandingRequest();
19   });
20
21   // Our tests will go here
22 });
```

Now, we can set up our tests to reflect the expectation that our templates do, in fact load when we navigate to different parts of the app.

```
1 it('loads the home template at /', function() {
2   $httpBackend.expectGET('templates/home.html')
3   .respond(200);
4   location.path('/');
5   rootScope.$digest(); // call the digest loop
6   $httpBackend.flush();
7 });
8
9 it('loads the dashboard template at /dashboard', function() {
10  $httpBackend.expectGET('templates/dashboard.html')
11  .respond(200);
12  location.path('/dashboard');
13  rootScope.$digest(); // call the digest loop
14  $httpBackend.flush();
15});
```

Notice that we're not actually returning a template into our tests (.respond(200) instead of .respond(200, "<div></div>div>")). Since we're only verifying that the template is getting loaded on our requests, we don't need to worry about *what* shows up, just that it does.

End-to-end tests is where we'll verify that the view looks how it's supposed to look in end-to-end tests.

End-to-end testing templates

When testing the view in end-to-end testing, we'll focus our attention to the actual data being loaded in the view, rather than that the template simply loads up. These tests will give us a better picture into what our users will actually see when the view loads.

We'll test the content of the view when it's in the view of our user.

To test our templates, we'll test that the view contains our expected HTML and that it does not load other parts of our app.

```
1 describe('E2E: Views', function() {
2   beforeEach(function() {
3     browser().navigateTo('#/');
4   });
5
6   it('should load the home template', function() {
7     {
8       expect(
9         element('#emailTable').html()
10      ).toContain('tbody');
11    });
12
13   it('should not load the dashboard template',
14     function() {
15     expect(
16       element('#dashboard').count()
17     ).toBe(0);
18   });
19 });
```

Making assertions on view templates are very straightforward.

Testing directives

Directives are the root of the angular workflow. Since most of the interaction that we do within our angular apps is working on directives, testing directive functionality is one of the most important components to test, especially when we build larger components.

When we're testing our directives, we'll want to test that the directive does *actually* get loaded into the view as well as that it behaves as we expect it to behave on the \$scope in the DOM.

We'll focus on testing the functionality of the directive in unit testing, while we'll be able to verify that we're using the directive properly in our end-to-end tests.

Unit testing directives

Unit testing directives that we'll set up a test to ensure that the directive renders as we expect it to, that the bindings are set up properly, that errors are thrown when we expect them to be thrown, and that the view shows the full directive as expected.

In this test, we'll work with the following directive:

```
1 angular.module('myApp')
2 .directive('notification', function($timeout) {
3   var html = '<div class="notification">' +
4     '<div class="notification-content">' +
5       '<p>{{ message }}</p>' +
6       '</div>' +
7     '</div>';
8   return {
9     restrict: 'A',
10    scope: { message: '=' },
11    template: html,
12    replace: true,
13    link: function(scope, ele, attrs) {
14      scope.$watch('message', function(n, o) {
15        if (n)
16          $timeout(function() {
17            ele.addClass('ng-hide');
18          }, 2000);
19      });
20    }});
});
```

In order to unit test our directives, we'll need to expose them to the view. Similar to how we test our controllers, we'll need manually place the directive in an element that we'll create in a similar manner that angular uses to place the directive in the first place.

```
1 describe('Unit: Directives', function() {
2   var ele, scope;
3   // Load our app
4   beforeEach(module('myApp'));
5   // Our tests will go here
6 });
```

Now, in order to load the directive into the eventual view, we'll need to compile the html content and apply the bindings that angular will do automatically.

```
1 describe('Unit: Directives', function() {
2   var ele, scope;
3   // Load our app
4   beforeEach(module('myApp'));
5   beforeEach(inject(function($compile, $rootScope) {
6     scope = $rootScope;
7     ele = angular.element(
8       '<div notification message="note"></div>'
9     );
10    $compile(ele)(scope);
11    scope.$apply();
12  }));
13
14  // Our tests will go here
15});
```

Notice, that we're creating an element that calls the directive just like when we place it into the DOM. Then we'll need to compile the element and run the digest loop to effectively *place* the element in our fake DOM.

Now all we have to do to test the directive is interact with it as though it is in our DOM. In order to make any changes to any bindings in the directive, we'll need to force a digest loop to run.

In our directive, we're attaching a binding to the `message` property of the scope, so when we change it, we'll need to do it within an `$apply()` call:

```
1 // ... test setup
2 it('should display the welcome text', function() {
3   scope.$apply(function() {
4     scope.note = "Notification message";
5   });
6
7   expect(
8     ele.html()
9   ).toContain("Notification message");
10});
```

End-to-end testing directives

When e2e testing directives, we're not primarily concerned with the functionality of the directives as we're testing what the user sees and does in the overall view.

As we're simply testing the view, end-to-end testing of directives looks similar to testing templates.

```
1 describe('E2E: directives', function() {  
2  
3   beforeEach(function() {  
4     browser().navigateTo('/');  
5   });  
6  
7   it('should have the welcome message', function() {  
8     expect(  
9       element('.notification', 'Notification').html()  
10      ).toContain('Notification message');  
11    });  
12  });
```

Testing events

When we're using events in our app, especially events that cause interaction to happen in the DOM, we'll want to set up tests that ensure the actual event gets fired and that it gets the expected event data.

We'll also want to set up tests that set expectations for what happens when an event is caught in the browser; that is that our reaction to the event is operating how we expect it to respond.

Unit testing events

When we're unit testing that events get fired, we're interested to know that they are actually called and that the right events are actually called. Second, we're mostly interested to know that the handlers have the data they need.

Setting up event testing is incredibly easy using the `spyOn()` Jasmine helper.

Imagine that we're testing a controller firing an `$emit` function. With this function, we can set up an expectation that our event is fired and that it's called with any arguments we're interested in.

First, as usual we'll need to set up our tests such that we'll get access to the controller's scope:

```
1 describe('myApp', function() {  
2   var scope;  
3   beforeEach/angular.mock.module('myApp');  
4   beforeEach/angular.mock.inject(function($rootScope) {  
5     scope = $rootScope.$new();  
6   });  
7 });
```

With this, we can simply set a `spyOn()` call on the scope for the “`$emit`” or “`$broadcast`” event.

```
1 // ...
2 });
3 it('should have emit called', function() {
4   spyOn(scope, '$emit');
5   scope.closePanel(); // for example
6           // or any event that
7           // causes the emit to
8           // be called
9   expect(scope.$emit)
10    .toHaveBeenCalledWith("panel:closed",
11      panel.id);
12});
```

Now, we can also test the events that happen with the `$on()` method that gets called after an event is fired. In this case, we won't need to do anything deeper than what normal happens other than manually `$broadcast`-ing or `$emit`-ing the events:

```
1 // ...
2 it('should set the panel to closed state',
3   function() {
4     scope.$broadcast("panel:closed", 1);
5     expect(scope.panel.state).toEqual("closed");
6  });
```

End-to-end testing events

End-to-end testing events getting called is fairly easy as we'll simply test that the functionality the event causes is firing as we expect.

Continuous Integration for angular

Angular's karma works great with continuous integration services. Continuous integration services, or CI for short allows our app a *gateway* for deployment as well as allows us to be confident that every check-in of our code is working as-expected.

Continuous integration servers are used by developers and companies all over the world of both large and small size and it's good to understand how to set up angular to work them.

Both [JenkinsCI⁹⁰](#) and [TravisCI⁹¹](#) are easily integrated with Karma and we strongly encourage the use of one of them.

⁹⁰<http://jenkins-ci.org/>

⁹¹<https://travis-ci.org/>

Protractor

The new end-to-end testing framework called Protractor, which will eventually replace the current framework as the default framework.

Protractor, unlike the angular scenario runner is built on [WebDriver⁹²](#) which is an API for controller browsers, written as extensions.

WebDriver has controls for IE, Chrome, Safari, FireFox, and more to control the browser. This has many benefits, including being more stable and quicker.

Protractor, like the angular scenario runner implement Jasmine as it's test scaffolding so we don't need to learn a completely new testing framework to use it.

Protractor itself can be installed as a standalone runner or it can be embedded into our tests as a library.

Installation

Installing Protractor can be done through `npm`:

```
1 $ npm install -g protractor
```



The `-g` tells `npm` to install protractor globally.

Unlike the angular scenario runner, Protractor requires a separate standalone server to be running at (can be configured) `http://location:4444`.

Luckily, protractor itself comes with a tool to ease the installation of a selenium server.

In order to access the script, we'll need to install protractor locally in the top level directory of the angular app we want to test.

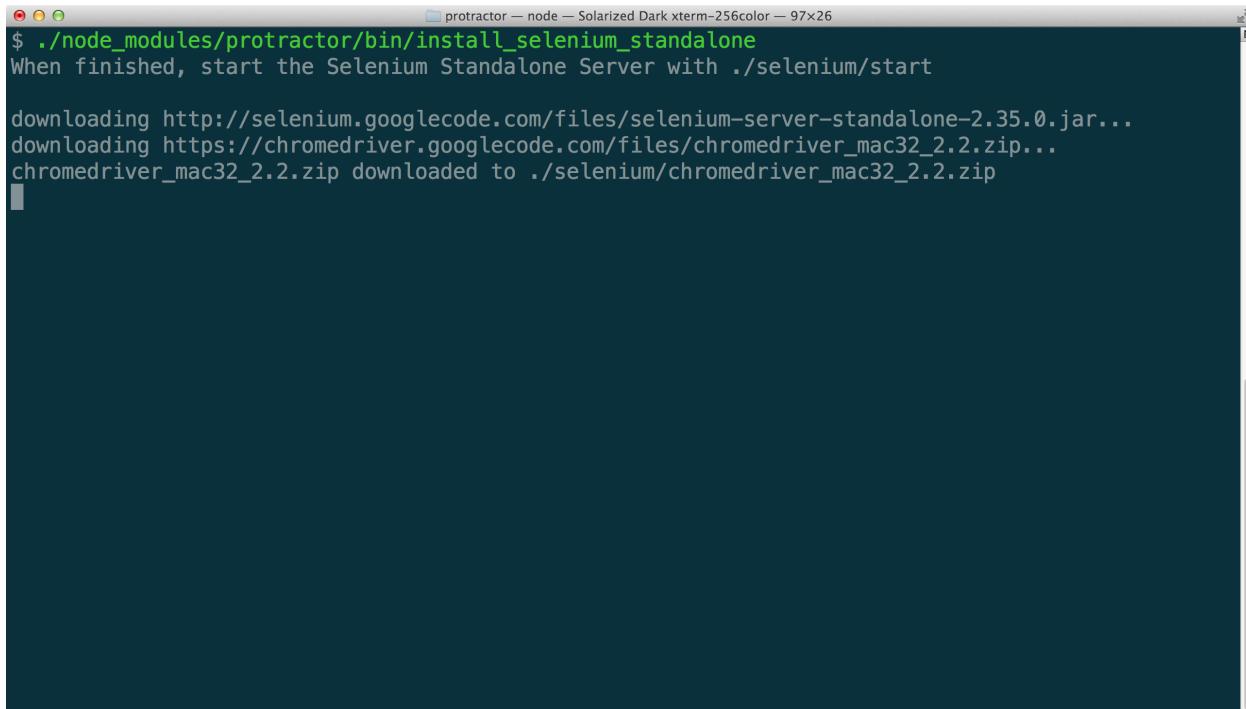
```
1 $ npm install protractor
```

Then we can run the selenium installation script located in the local `node_modules/` directory:

```
1 $ ./node_modules/protractor/bin/install.selenium_standalone
```

⁹²<https://code.google.com/p/selenium/wiki/WebDriverJs>

This script will download the required files to run selenium itself and build a directory with them as well as a start script.



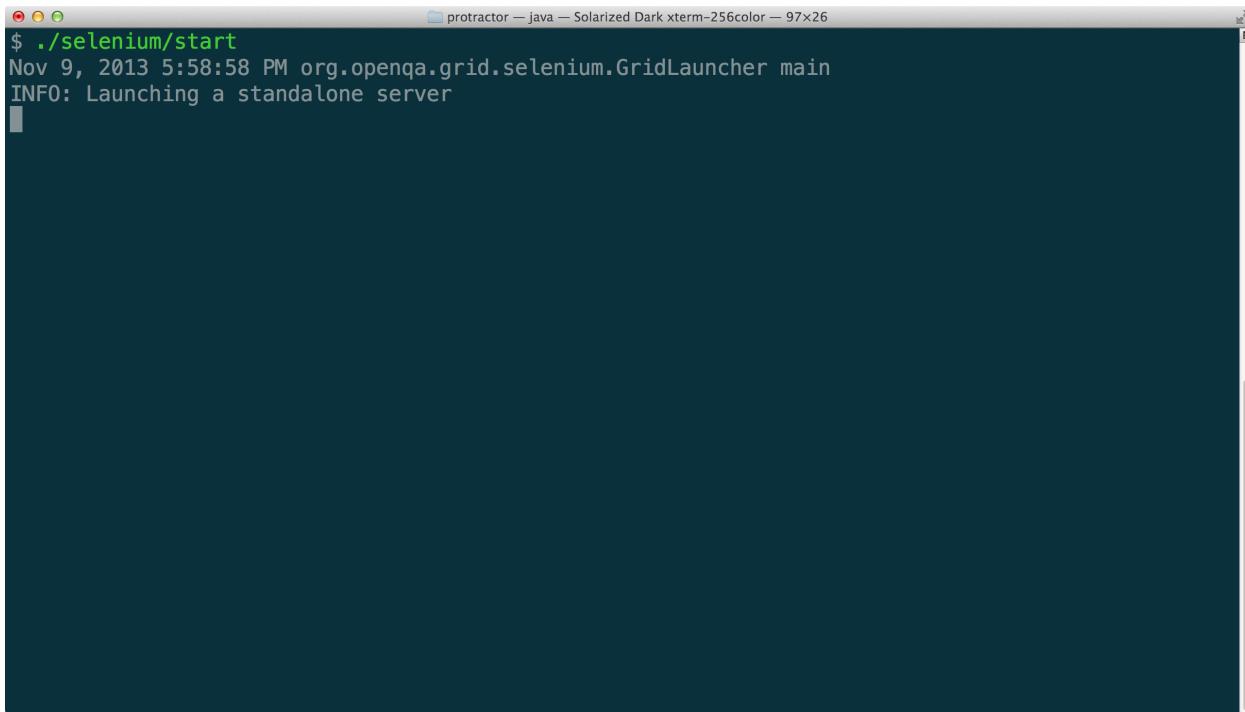
```
protractor — node — Solarized Dark xterm—256color — 97x26
$ ./node_modules/protractor/bin/install_selenium_standalone
When finished, start the Selenium Standalone Server with ./selenium/start

downloading http://selenium.googlecode.com/files/selenium-server-standalone-2.35.0.jar...
downloading https://chromedriver.googlecode.com/files/chromedriver_mac32_2.2.zip...
chromedriver_mac32_2.2.zip downloaded to ./selenium/chromedriver_mac32_2.2.zip
```

Install selenium

When this script is finished, we can start the standalone version of selenium with the Chrome driver by executing the start script:

```
1 $ ./selenium/start
```



```
$ ./selenium/start
Nov 9, 2013 5:58:58 PM org.openqa.grid.selenium.GridLauncher main
INFO: Launching a standalone server
```

Install selenium



If your installation of selenium is having trouble running, try updating the Chromedriver by downloading the latest version [here](#)⁹³.

Now we can use protractor to connect to our selenium server running in the background using protractor.

Configuration

Protractor, like Karma requires a configuration script to run that tells the protractor runner how to connect to selenium, which browser(s) to use and where the test files are located.

The easiest method to create a configuration file is by copying the reference configuration file from the protractor install:

```
1 $ cp ./node_modules/protractor/referenceConf.js protractor_conf.js
```

This will include all of the options available for running protractor.

⁹³<http://chromedriver.storage.googleapis.com/index.html>

Configuration options

Protractor includes several options to allow us to configure how we are running protractor with our spec files.

seleniumServerJar (string)

By setting the `seleniumServerJar` as the path to the standalone selenium server, we can tell protractor to launch the selenium server when we run the test.

This is useful when we're running continuous integration (CI) tests, but the startup time of selenium is slow, so this is quite inefficient for development time.

seleniumPort (integer)

The port to start the selenium server on, if protractor is to start the server.

chromeDriver (string)

The path of the chromedriver to start as the `webdriver.chrome.driver` when starting the selenium server. Protractor will attempt to find the chromedriver using the `PATH` environment variable if this is null.

seleniumArgs (array)

This array of strings are additional arguments that we can manually pass to the selenium server on boot.

sauceUser / sauceKey (string)

If the `sauceUser` and `sauceKey` are set, then the selenium server will not be started and the tests will be run remotely using [SauceLabs⁹⁴](#), a cloud testing service.

seleniumAddress (string)

This is the address of a running selenium server if we're running our own. For instance, if we start the selenium server using the included script, this would be set to: `http://localhost:4444/wd/hub`.

allScriptsTimeout (integer)

The timeout for each script to run in the browser. That is, if a script takes longer than this time to complete, then they will be killed and reported as failures.

⁹⁴<https://saucelabs.com/>

specs (array)

This array of strings is the location of the specs to run. These can be relative or absolute filepaths as well as patterns to match.

```
1  specs: [
2    'spec/*_spec.js',
3  ],
```

capabilities (object)

This object is the key-value pairs of capabilities to be passed to the webdriver instance.

```
1  capabilities: {
2    'browserName': 'chrome'
3 }
```

baseUrl (string)

This is the base URL for our application. If our tests use relative paths, then these paths will be appended to this string.

```
1 baseUrl: 'http://localhost:9000'
```

rootElement (string)

The selector for the element that houses the angular app. It will default to body, but if we include the angular app as a descendant of the <body> element, then we need to include this option.

onPrepare (string/function)

This function will be run after protractor has been started and is ready to run, but before *all* of the specs are executed. This can also be a file that contains code to be run before the specs are executed.

The onPrepare option is useful for setting up Jasmine or protractor for instance:

```
1  onPrepare: function() {
2    // For example, adding a Jasmine reporter:
3    jasmine.getEnv()
4      .addReporter(new jasmine.JUnitXmlReporter(
5        'outputdir/', true, true)
6      );
7 },
```

params (object)

The `params` object will be passed directly to the protractor instance and we can access this object inside our tests. We can put any arbitrary values inside of this `params` object.

jasmineNodeOpts (object)

The `jasmineNodeOpts` specifies the options to be passed to the jasmine node instance. The full list of options for `jasmineNodeOpts` can be found at <https://github.com/juliemr/minijasminenode⁹⁵>.

Writing tests

Protractor uses Jasmine by default and thus the tests that we'll write are very similar to those that we can write with karma with a few key differences.

Protractor exposes a few global variables that we can use in our tests:

browser

This is a wrapper around the instance of the webdriver. We'll use this for navigation and page information.

element

The `element` function helps us find and interact with elements on the page that we're testing.

The return value of `element` is **not** a DOM element, remember. It's an instance of `ElementFinder`, which is an object that acts over webdriver. We can use this object interact with our objects on the page using methods such as `sendKeys` and `click`.

The full API is extensive and more documentation can be found at the documentation page of <https://github.com/angular/protractor/blob/master/docs/api.md⁹⁶>.

⁹⁵<https://github.com/juliemr/minijasminenode>

⁹⁶<https://github.com/angular/protractor/blob/master/docs/api.md>

by

The collection of element locator strategies. We can use this to find elements by CSS selectors, IDs, or even attributes that are bound via ng-model.

by.binding This allows us to search for elements that are bound either by ng-bind or by the template notation {{ }}

by.model We can use by.model to search for input elements that are bound by ng-model.

by.repeater We can search for elements that contain the directive ng-repeat.

by.id by.id allows us to search for elements by their CSS id.

by.css We can search for elements by their CSS selector using by.css.

Since we'll select by CSS selectors often, protractor binds the global variable of \$ to the function element(by.css) as a convenience.

protractor

The protractor namespace that wraps the webdriver namespace. This contains static variables and classes.

Combining these global variables, we can write efficient end-to-end tests.



Note: Protractor assumes we are testing angular apps and expects angular to be present on the page. If it's not found, then it will throw an error.

We can load non-angular pages by dipping down into the lower level webdriver instance using the browser.driver object.

For instance, let's say we want to test the basics example on the AngularJS homepage:

```
1 describe('angularjs homepage', function() {
2   it('should greet the named user', function() {
3     // Load the AngularJS homepage.
4     browser.get('http://www.angularjs.org');
5
6     element(by.model('yourName'))
7       .sendKeys('Julie');
8
9     var greeting =
10      element(by.binding('yourName'));
11
12    expect(greeting.getText())
13      .toEqual('Hello Julie!');
14  });
15});
```

Page Objects

To make our tests more readable, we can use the webdriver concept of Page Objects. Page Objects are basically classes that allow us to wrap specific page functionality into a clean interaction.

For instance, we can wrap a page like:

```
1 var AngularHomepage = function() {
2   this.nameInput = element(by.model('yourName'));
3   this.greeting = element(by.binding('yourName'));
4
5   this.get = function() {
6     browser.get('http://www.angularjs.org');
7   };
8
9   this.setName = function(name) {
10     this.nameInput.sendKeys(name);
11   };
12};
```

Now we can clean up our tests such as:

```
1 describe('angularjs homepage', function() {
2   it('should greet the named user', function() {
3     var angularHomepage = new AngularHomepage();
4     angularHomepage.get();
5
6     angularHomepage.setName('Julie');
7
8     expect(angularHomepage.greeting.getText()).toEqual('Hello Julie!');
9   });
10});
```

Angular Animation

ngAnimate is a module created by the Angular team that gives our angular apps hooks into providing CSS and javascript.

There are several ways to make animations in an angular app:

- Using CSS3 Animations
- Using JavaScript animations
- Using CSS3 Transitions

We'll discuss these three different methods of animating in this chapter and aim to give you a solid understanding of how to power your own custom animations.

Installation

Since 1.2.0.rc1, animations have been pulled out of the core of angular into it's own module. In order to include animations in our angular app, we'll need to install and reference it in our app.

We can download it from [code.angularjs.org⁹⁷](http://code.angularjs.org) and save it in a place that we can reference it from our HTML, like js/vendor/angular-animate.js.

We can also install it using bower, which will place it in our usual bower directory. For more information about bower, see the [bower chapter](#).

```
1 $ bower install --save angular-animate
```

We'll need to reference this in our HTML *after* we reference angular itself.

```
1 <script src="js/vendor/angular.js"></script>
2 <script src="js/vendor/angular-animate.js"></script>
```

Lastly, we'll need to reference the ngAnimate module as a dependency in our app module:

```
1 angular.module('myApp', ['ngAnimate']);
```

Now we are ready to take on animations with AngularJS.

⁹⁷<http://code.angularjs.org/>

How it works

The `$animate` service itself, by default applies two CSS classes for each animation event to the animated element. The `$animate` service supports several built-in angular directives that automatically support animation without needing any extra configuration. It allows us to build our own animations for our directives.

All of the pre-existing directives that support animation do so through monitoring events provided on the directive. For instance, when a new `ngView` *enters* and brings new content into the browser, this event is called the *enter* event for `ngView`.

The following is a list of directives and the events that they each fire at the different states. When defining our animations within the different states, we will use these events to define how our animations will work.

Directive	Events
<code>ngRepeat</code>	<code>enter</code> , <code>leave</code> , <code>move</code>
<code>ngView</code>	<code>enter</code> , <code>leave</code>
<code>ngInclude</code>	<code>enter</code> , <code>leave</code>
<code>ngSwitch</code>	<code>enter</code> , <code>leave</code>
<code>ngIf</code>	<code>enter</code> , <code>leave</code>
<code>ngClass</code>	<code>add</code> , <code>remove</code>
<code>ngShow</code>	<code>add</code> , <code>remove</code>
<code>ngHide</code>	<code>add</code> , <code>remove</code>

The `$animate` service attaches specific classes based upon the events that the directive emits in the form of `ng- [EVENT]` and `ng- [EVENT] -active`.

Automatically added classes

For the directives that fire the `enter` event, they will get a class of `.ng-enter` when the DOM is being updated. Angular will then add the `ng-enter-active` class which triggers the animation. `ngAnimate` will automatically detect the CSS code to determine when the animation is complete.

When the event is done, angular will remove both classes from the DOM element. This enables us to define animate-able properties to the DOM elements.

If the browser does *not* support CSS transitions or animations, then the animation will start and end immediately and the DOM will end up at it's final state, with no CSS transitions/animation classes applied.

The same convention applies for all of the supported animation actions: `enter`, `leave`, `move`, `add`, and `remove`.

Using CSS3 Transitions

By far the easiest way to include animations in our app and works for all browsers except IE9 and earlier versions. For browsers that do not support CSS3 Transitions, this will gracefully fallback to the non-animated version of the app.

To do **any** CSS animation, we'll need to make sure we include the classes we'll be working with to the DOM element we're interested in animating.

For instance, in the following demo we'll look at animating the following element:

```
1 <div class="fadein"></div>
```

CSS3 Transitions are fully class-based, which means as long as we have classes that define the animation in our HTML the animation will be animated in the browser.

In order for us to achieve animations with classes, we'll need to follow the angular CSS naming conventions to define our CSS transitions.

CSS transitions are effects that let an element gradually change from one style to another style. To define an animation, we must specify the property we want to add an animation to as well as specify the duration of effect.

For instance, this will add a transition effect on the all of the properties on DOM elements with the `.fadein` class for a 2 second duration.

```
1 .fadein {
2   transition: 2s linear all;
3   -webkit-transition: 2s linear all;
4   -moz-transition: 2s linear all;
5   -o-transition: 2s linear all;
6 }
```

With this set, we can define properties on different states of the DOM element.

```
1 .fadein:hover {
2   width: 300px;
3   height: 300px;
4 }
```

With `ngAnimate`, our directives animations are started by angular adding the two classes, the initial `ng-[EVENT]` class and shortly thereafter, the `ng-[EVENT]-active` class.

To automatically allow the DOM elements from above transition with angular animation, we'll modify the initial `.fadein` example from above to include the initial state class:

```
1 .fadein.ng-enter {  
2   opacity: 0;  
3 }  
4 .fadein.ng-enter.ng-enter-active {  
5   opacity: 1;  
6 }
```

To actually *run* the animation, we'll need to include the css definitions.

```
1 .fadein.ng-enter {  
2   transition: 2s linear all;  
3   -webkit-transition: 2s linear all;  
4   -moz-transition: 2s linear all;  
5   -o-transition: 2s linear all;  
6 }
```

Using CSS3 Animations

CSS3 animations are more extensive and more complex than CSS3 transitions. They are supported by all major browsers exception IE9 and earlier versions. With CSS3 animations, we'll use the same initial class ng- [EVENT], but we don't need to define animation states in the ng- [EVENT] -active state as our CSS rules will handle the rest of the block.

The @keyframes rule is where the animation is created. Within the CSS element where we define the @keyframes rule, we'll define the CSS styles that we want to be manipulated.

When we want to animate the DOM element, we'll use the animation: to bind the @keyframe CSS property to apply the animation to the CSS element. When we bind the animation to the CSS element, we'll need to specify both the name of the animation as well as the duration.



Remember to add the animation duration: If we forget to add the duration of the animation, it will not run as the duration will default to 0.

To create our @keyframes rule, we'll need to give our keyframe a name and set the time periods of the where the properties should be throughout the animation.

```
1 @keyframes firstAnimation {
2   0% {
3     color: yellow;
4   }
5   100% {
6     color: black;
7   }
8 }
9 /* For Chrome and Safari */
10 @-webkit-keyframes firstAnimation {
11   /* from is equivalent to 0% */
12   from {
13     color: yellow;
14   }
15   /* to is equivalent to 100% */
16   to {
17     color: black;
18   }
19 }
```



Using the keyword `from` is equivalent to setting the percentage to `0%`. Using the keyword `to` is equivalent to setting the percentage to `100%`.

We are not limited to `0%` and `100%`, we can provide animations in steps, such as at `10%`, `15%`, etc.

To assign this `@keyframe` property to the classes we want to animate, we'll use the `animation` keyword that will apply the animation to the elements targeted by the CSS selector.

```
1 .fadein:hover {
2   -webkit-animation: 2s firstAnimation;
3   animation: 2s firstAnimation;
4 }
```

With `ngAnimate`, we'll bind the `firstAnimation` to any elements that are targeted with the `.fadein` class. Angular will apply and remove the `.ng-enter` class for us automatically, so we can simply attach our event to the `.fadein.ng-enter` class:

```
1 .fadein.ng-enter {  
2   -moz-animation: 2s firstAnimation;  
3   -o-animation: 2s firstAnimation;  
4   -webkit-animation: 2s firstAnimation;  
5   animation: 2s firstAnimation;  
6 }
```

Using JavaScript animations

JavaScript animation is different than the previous two ways to animate using angular in that we'll set properties on the DOM element directly using javascript.

JavaScript animation is supported in all major browsers that enable javascript, so it's a good choice if we want to offer animations on browsers that do not support CSS transitions/animations.

Instead of manipulating our CSS to animate elements, we'll update our javascript to handle running animations for us.

The `ngAnimate` module adds the `.animation` method to our the module API that presents an interface to create our animations on top.

The `animation()` method takes two parameters:

- `classname` (string)

This is the classname that will match the class of the element to animate. For our examples thus far, the animation should be named: `.fadein`.

- `animateFun` (function)

The `animate` function is expected to return an object that includes functions for the different events that are fired by the directive where it's used.

See the [\\$animate API](#) docs for detailed documentation on these functions.

```
1 angular.module('myApp', ['ngAnimate'])
2 .animation('.fadein', function() {
3   return {
4     enter: function(element, done) {
5       // Run animation
6
7       return function(cancelled) {
8         // Cancel animation
9       }
10    }
11  }
12});
```

The functions will all be called with the element and the callback function (`done()`). Inside these functions, it's a free-for-all in terms of what we do with the element. The only requirement is that we call `done()` when we are done with the animation.

Inside these functions, we can return an `end` function that will be called when the animation is complete OR the animation has been canceled.

When the animation is triggered, `$animate` will look for the matching animation function for the event. If it finds a function that matches the event, then it will execute it.

Animating built-in directives

Animating ngRepeat

The `ngRepeat` directive fires the events:

Action	Event name
An item was inserted to the list of items	enter
An item was removed to the list of items	leave
An item was moved in the list of items	move

For the three examples, we'll work with the HTML as follows:

```
1 <div ng-controller="HomeCtrl">
2   <ul>
3     <li class="fadein" ng-repeat="r in roommates">
4       {{ r }}
5     </li>
6   </div>
```

We'll assume that the HomeCtrl is defined as such:

```
1 angular.module('myApp', ['ngAnimate'])
2 .controller('HomeCtrl', function($scope) {
3   $scope.roommates = [
4     'Ari', 'Q', 'Sean', 'Anand'
5   ];
6   setTimeout(function() {
7     $scope.roommates.push('Ginger');
8     $scope.$apply(); // Trigger a digest
9
10    setTimeout(function() {
11      $scope.roommates.shift();
12      $scope.$apply(); // Trigger digest
13    }, 2000);
14  }, 1000);
15});
```

In these examples, we have a list of roommates that consists of four elements. After a second, we'll have a fifth added. Two seconds later, we'll remove the first element.

CSS3 Transitions

To animate items in the ngRepeat list, we'll need to make sure to add the CSS class that will present the initial state of the element and the class that will define the final state for both of the enter and edit states.

We'll start by defining the animation properties on the initial class(es):

```
1 .fadein.ng-enter,
2 .fadein.ng-leave {
3   transition: 2s linear all;
4   -webkit-transition: 2s linear all;
5   -moz-transition: 2s linear all;
6   -o-transition: 2s linear all;
7 }
```

Now, we can simply define the stages of the initial and final css properties in the animation. Here, we'll fade the element in with green text and turn the text black at the final stage of the enter animation. In the leave (item removal) animation, we'll reverse the properties:

```
1 .fadein.ng-enter {
2   opacity: 0;
3   color: green;
4 }
5 .fadein.ng-enter.ng-enter-active {
6   opacity: 1;
7   color: black;
8 }
9 .fadein.ng-leave {}
10 .fadein.ng-leave.ng-leave-active {
11   opacity: 0;
12 }
```

CSS3 Keyframe animation

When using keyframe animation, we don't need to define a start and an end class, instead we'll define only a single selector that includes the animation CSS key.

We can start by defining the animation properties for the keyframes:

```
1 @keyframes animateView-enter {
2   from {opacity:0;}
3   to {opacity:1;}
4 }
5 @-webkit-keyframes animateView-enter {
6   from {opacity:0;}
7   to {opacity:1;}
8 }
9 @keyframes animateView-leave {
10   from {opacity: 1;}
```

```
11   to {opacity: 0;}
```

```
12 }
```

```
13 @-webkit-keyframes animateView-leave {
```

```
14   from {opacity: 1;}
```

```
15   to {opacity: 0;}
```

```
16 }
```

With the keyframe set, we can simply attach the animation to the CSS classes added by `ngAnimate`:

```
1 .fadein.ng-enter {
```

```
2   -webkit-animation: 2s fadein-enter-animation;
```

```
3   -moz-animation: 2s fadein-enter-animation;
```

```
4   -o-animation: 2s fadein-enter-animation;
```

```
5   animation: 2s fadein-enter-animation;
```

```
6 }
```

```
7 .fadein.ng-leave {
```

```
8   -webkit-animation: 2s fadein-leave-animation;
```

```
9   -moz-animation: 2s fadein-leave-animation;
```

```
10  -o-animation: 2s fadein-leave-animation;
```

```
11  animation: 2s fadein-leave-animation;
```

```
12 }
```

JavaScript animation

When animating with javascript, we'll need to define the `enter` and `leave` properties on our animation description object.

```
1 angular.module('myApp')
```

```
2 .animation('.fadein', function() {
```

```
3   return {
```

```
4     enter: function(element, done) {
```

```
5       // Raw animation without jQuery
```

```
6       // This is much simpler with jQuery
```

```
7       var op = 0, timeout,
```

```
8       animateFn = function() {
```

```
9         op += 10;
```

```
10        element.css('opacity', op/100);
```

```
11        if (op >= 100) {
```

```
12          clearInterval(timeout);
```

```
13          done();
```

```
14        }
```

```
15      };
```

```

16
17     // Set initial opacity to 0
18     element.css('opacity', 0);
19     timeout = setInterval/animateFn, 100);
20 },
21 leave: function(element, done) {
22     var op = 100,
23         timeout,
24         animateFn = function() {
25             op-=10;
26             element.css('opacity', op/100);
27             if (op <= 0) {
28                 clearInterval(timeout);
29                 done();
30             }
31         };
32     element.css('opacity', 100);
33     timeout = setInterval/animateFn, 100);
34 }
35 }
36 });

```

Animating ngView

The `ngView` directive fires the events:

Action	Event name
New content is ready for the view	enter
Existing content is ready to be hidden	leave

For the three examples, we'll work with the HTML as follows:

```

1 <a href="#">Home</a>
2 <a href="#/two">Second view</a>
3 <a href="#/three">Third view</a>
4 <div class="animateView" ng-view></div>

```

As we're working with the `ng-view` directive, we're working with routes inside of Angular. For more information about routes, check out the [routing chapter](#). The download for angular-routing is [here](#)⁹⁸.

⁹⁸<http://code.angularjs.org/1.2.0-rc.2/angular-route.js>

For the following examples, we'll set our routes to be defined as:

```
1 angular.module('myApp', ['ngAnimate', 'ngRoute'])
2 .config(function($routeProvider) {
3   $routeProvider.when('/', {
4     template: '<h2>One</h2>'
5   }).when('/two', {
6     template: '<h2>Two</h2>'
7   }).when('/three', {
8     template: '<h2>Three</h2>'
9   });
10 })
```

These examples have three routes that show a different view.

CSS3 Transitions

To animate items in the `ngView` list, we'll need to add the CSS class that will define the initial state of the element and the class that will define the final state for both of the enter and edit states.

```
1 .animateView.ng-enter,
2 .animateView.ng-leave {
3   transition: 2s linear all;
4   -webkit-transition: 2s linear all;
5   -moz-transition: 2s linear all;
6   -o-transition: 2s linear all;
7 }
```

Now, we can simply define the stages of the initial and final css properties in the animation. Here, we'll fade the element in with green text and turn the text black at the final stage of the enter animation. In the leave (item removal) animation, we'll reverse the properties:

```
1 .animateView.ng-enter {
2   opacity: 0;
3   color: green;
4 }
5 .animateView.ng-enter.ng-enter-active {
6   opacity: 1;
7   color: black;
8 }
9 .animateView.ng-leave {}
10 .animateView.ng-leave.ng-leave-active {
11   opacity: 0;
12 }
```

CSS3 Keyframe animation

We'll start by adding the @keyframe animations that we'll define for the animation.

```
1 @keyframes animateView-enter {
2   from {opacity:0;}
3   to {opacity:1;}
4 }
5 @-webkit-keyframes animateView-enter {
6   from {opacity:0;}
7   to {opacity:1;}
8 }
9 @keyframes animateView-leave {
10  from {opacity: 1;}
11  to {opacity: 0;}
12 }
13 @-webkit-keyframes animateView-leave {
14  from {opacity: 1;}
15  to {opacity: 0;}
16 }
```

All we need to do to apply the animation is include the animation CSS style on our class:

```
1 .animateView.ng-enter {
2   -webkit-animation: 2s animateView-enter;
3   -moz-animation: 2s animateView-enter;
4   -o-animation: 2s animateView-enter;
5   animation: 2s animateView-enter;
6 }
7 .animateView.ng-leave {
8   -webkit-animation: 2s animateView-leave;
9   -moz-animation: 2s animateView-leave;
10  -o-animation: 2s animateView-leave;
11  animation: 2s animateView-leave;
12 }
```

JavaScript animation

First [download⁹⁹](#) and include jQuery in the head of the document.

When animating with javascript, we'll need to define the enter and leave properties on our animation description object.

⁹⁹<http://jquery.com/download/>

```
1 angular.module('myApp')
2   .animation('.animateView', function() {
3     return {
4       enter: function(element, done) {
5         // Example to show how to animate
6         // with jQuery. Note, this requires
7         // jQuery to be included in the HTML
8         $(element).css({
9           opacity: 0
10        });
11        $(element).animate({
12          opacity: 1
13        }, done);
14      },
15      leave: function(element, done) {
16        done();
17      }
18    }
19  });
```

Animating ngInclude

The `ngInclude` directive fires the events:

Action	Event name
New content is ready for the view	enter
Existing content is ready to be hidden	leave

For the three examples, we'll work with the HTML as follows:

We'll also include the inline templates (for demo purposes) in our page. Alternatively, we could set these views to be fetched from a remote server.

```
1 <script type="text/ng-template" id="/home.html">
2   Home Template
3 </script>
4 <script type="text/ng-template" id="/second.html">
5   Second Template
6 </script>
7 <script type="text/ng-template" id="/third.html">
8   Third Template
9 </script>
```

CSS3 Transitions

To animate items in the `ngInclude`, we'll need to add the CSS class that will define the initial state of the element and the class that will define the final state for both of the enter and edit states.

```
1 .animateInclude.ng-enter,
2 .animateInclude.ng-leave {
3   transition: 2s linear all;
4   -webkit-transition: 2s linear all;
5   -moz-transition: 2s linear all;
6   -o-transition: 2s linear all;
7 }
```

Now, we can simply define the stages of the initial and final css properties in the animation. Here, we'll fade the element in with green text and turn the text black at the final stage of the enter animation. In the leave (item removal) animation, we'll reverse the properties:

```
1 .animateInclude.ng-enter {
2   opacity: 0;
3   color: green;
4 }
5 .animateInclude.ng-enter.ng-enter-active {
6   opacity: 1;
7   color: black;
8 }
9 .animateInclude.ng-leave {}
10 .animateInclude.ng-leave.ng-leave-active {
11   opacity: 0;
12 }
```

CSS3 Animations

We'll start by adding the `@keyframe` animations that we'll define for the animation.

```
1 @keyframes animateInclude-enter {
2   from {opacity:0;}
3   to {opacity:1; color: green}
4 }
5 @-webkit-keyframes animateInclude-enter {
6   from {opacity:0;}
7   to {opacity:1; color: green}
8 }
9 @keyframes animateInclude-leave {
10  from {opacity: 1;}
11  to {opacity: 0; color: black}
12 }
13 @-webkit-keyframes animateInclude-leave {
14  from {opacity: 1;}
15  to {opacity: 0; color: black}
16 }
```

All we need to do to apply the animation is include the animation CSS style on our classes:

```
1 .animateInclude.ng-enter {
2   -webkit-animation: 2s animateInclude-enter;
3   -moz-animation: 2s animateInclude-enter;
4   -o-animation: 2s animateInclude-enter;
5   animation: 2s animateInclude-enter;
6 }
7 .animateInclude.ng-leave {
8   -webkit-animation: 2s animateInclude-leave;
9   -moz-animation: 2s animateInclude-leave;
10  -o-animation: 2s animateInclude-leave;
11  animation: 2s animateInclude-leave;
12 }
```

JavaScript animation

When animating with javascript, we'll need to define the enter and leave properties on our animation description object.

```
1 angular.module('myApp')
2   .animation('.animateInclude', function() {
3     return {
4       enter: function(element, done) {
5         // Example to show how to animate
6         // with jQuery. Note, this requires
7         // jQuery to be included in the HTML
8         $(element).css({
9           opacity: 0
10        });
11        $(element).animate({
12          opacity: 1
13        }, done);
14      },
15      leave: function(element, done) {
16        done();
17      }
18    }
19  });
```

Animating ngSwitch

The `ngSwitch` directive fires the events:

Action	Event name
New content is ready for the view	enter
Existing content is ready to be hidden	leave

The `ngSwitch` directive is similar to our previous examples for these examples, we'll work with the HTML as follows that uses the `ng-switch` directive:

CSS3 Transitions

To animate items in the `ngSwitch`, we'll need to add the CSS class that will define the initial state of the element and the class that will define the final state for both of the enter and edit states.

```
1 .animateSwitch.ng-enter,
2 .animateSwitch.ng-leave {
3   transition: 2s linear all;
4   -webkit-transition: 2s linear all;
5   -moz-transition: 2s linear all;
6   -o-transition: 2s linear all;
7 }
```

Now, we can simply define the stages of the initial and final CSS properties in the animation. Here, we'll fade the element in with green text and turn the text black at the final stage of the enter animation. In the leave (item removal) animation, we'll reverse the properties:

```
1 .animateSwitch.ng-enter {
2   opacity: 0;
3   color: green;
4 }
5 .animateSwitch.ng-enter.ng-enter-active {
6   opacity: 1;
7   color: black;
8 }
9 .animateSwitch.ng-leave {}
10 .animateSwitch.ng-leave.ng-leave-active {
11   opacity: 0;
12 }
```

CSS3 Animations

We'll start by adding the @keyframe animations that we'll define for the animation.

```
1 @keyframes animateSwitch-enter {
2   from {opacity:0;}
3   to {opacity:1; color: green}
4 }
5 @-webkit-keyframes animateSwitch-enter {
6   from {opacity:0;}
7   to {opacity:1; color: green}
8 }
9 @keyframes animateSwitch-leave {
10  from {opacity: 1;}
11  to {opacity: 0; color: black}
12 }
13 @-webkit-keyframes animateSwitch-leave {
```

```
14   from {opacity: 1;}  
15   to {opacity: 0; color: black}  
16 }
```

All we need to do to apply the animation is include the animation CSS style on our classes:

```
1 .animateSwitch.ng-enter {  
2   -webkit-animation: 2s animateSwitch-enter;  
3   -moz-animation: 2s animateSwitch-enter;  
4   -o-animation: 2s animateSwitch-enter;  
5   animation: 2s animateSwitch-enter;  
6 }  
7 .animateSwitch.ng-leave {  
8   -webkit-animation: 2s animateSwitch-leave;  
9   -moz-animation: 2s animateSwitch-leave;  
10  -o-animation: 2s animateSwitch-leave;  
11  animation: 2s animateSwitch-leave;  
12 }
```

JavaScript animation

When animating with javascript, we'll need to define the enter and leave properties on our animation description object.

```
1 angular.module('myApp')  
2   .animation('.animateSwitch', function() {  
3     return {  
4       enter: function(element, done) {  
5         // Example to show how to animate  
6         // with jQuery. Note, this requires  
7         // jQuery to be included in the HTML  
8         $(element).css({  
9           opacity: 0  
10        });  
11        $(element).animate({  
12          opacity: 1  
13        }, done);  
14      },  
15      leave: function(element, done) {  
16        done();  
17      }  
18    }  
19  });
```

Animating ngIf

The `ngSwitch` directive fires the events:

Action	Event name
Fired after <code>ngIf</code> contents change and the new DOM element is injected in	enter
Fired just before the <code>ngIf</code> contents are removed	leave

For the following `ngIf` examples, we're going to work with the HTML:

CSS3 Transitions

To animate items in the `ngIf`, we'll need to add the CSS class that will define the initial state of the element and the class that will define the final state for both of the enter and edit states.

```

1 .animateNgIf.ng-enter,
2 .animateNgIf.ng-leave {
3   transition: 2s linear all;
4   -webkit-transition: 2s linear all;
5   -moz-transition: 2s linear all;
6   -o-transition: 2s linear all;
7 }
```

Now, we can simply define the stages of the initial and final css properties in the animation. Here, we'll fade the element in with green text and turn the text black at the final stage of the enter animation. In the `leave` (item removal) animation, we'll reverse the properties:

```

1 .animateNgIf.ng-enter {
2   opacity: 0;
3   color: green;
4 }
5 .animateNgIf.ng-enter.ng-enter-active {
6   opacity: 1;
7   color: black;
8 }
9 .animateNgIf.ng-leave {}
10 .animateNgIf.ng-leave.ng-leave-active {
11   opacity: 0;
12 }
```

CSS3 Animations

We'll start by adding the @keyframe animations that we'll define for the animation.

```
1 @keyframes animateNgIf-enter {
2   from {opacity:0;}
3   to {opacity:1;}
4 }
5 @-webkit-keyframes animateNgIf-enter {
6   from {opacity:0;}
7   to {opacity:1;}
8 }
9 @keyframes animateNgIf-leave {
10  from {opacity: 1;}
11  to {opacity: 0;}
12 }
13 @-webkit-keyframes animateNgIf-leave {
14  from {opacity: 1;}
15  to {opacity: 0;}
16 }
```

All we need to do to apply the animation is include the animation CSS style on our classes:

```
1 .animateNgIf.ng-enter {
2   -webkit-animation: 2s animateNgIf-enter;
3   -moz-animation: 2s animateNgIf-enter;
4   -o-animation: 2s animateNgIf-enter;
5   animation: 2s animateNgIf-enter;
6 }
7 .animateNgIf.ng-leave {
8   -webkit-animation: 2s animateNgIf-leave;
9   -moz-animation: 2s animateNgIf-leave;
10  -o-animation: 2s animateNgIf-leave;
11  animation: 2s animateNgIf-leave;
12 }
```

JavaScript animation

When animating with javascript, we'll need to define the enter and leave properties on our animation description object.

```

1 angular.module('myApp')
2 .animation('.animateNgIf', function() {
3   return {
4     enter: function(element, done) {
5       // Example to show how to animate
6       // with jQuery. Note, this requires
7       // jQuery to be included in the HTML
8       $(element).css({
9         opacity: 0
10      });
11      $(element).animate({
12        opacity: 1
13      }, done);
14    },
15    leave: function(element, done) {
16      done();
17    }
18  }
19 });

```

Animating ngClass

It's possible to animate the behavior that happens when classes are changed in the view. When a CSS class changes (such as in the `ngShow` and `ngHide` directives), `$animate` notices and triggers animations for both when the classes are added and old ones are removed.

Instead of using the naming convention for entering, we'll use a new CSS convention for `ngClass` where we postfix the new class as: `[CLASSNAME]-add` and `[CLASSNAME]-remove`.

Similar to the `enter` events above, the `[CLASSNAME]-add-active` and the `[CLASSNAME]-remove-active` will get added at the appropriate times for the specific event.

When we are animating on the classes, the animation is fired `first` and when it's complete, the final class is added. When a class is removed, the class will remain on the element until the animation is complete.

The `ngClass` directive fires the events:

Action	Event name
After <code>ngClass</code> evaluates to a truthy value and before the class has been added	<code>add</code>
Fired just before the class is removed	<code>remove</code>

For the following `ngClass` examples, we're going to work with the HTML:

CSS3 Transitions

To animate items in the `ngClass`, we'll need to add the CSS class that will define the initial state of the element and the class that will define the final state for both of the enter and edit states.

```
1 .animateMe.grown-add,
2 .animateMe.grown-remove {
3   transition: 2s linear all;
4   -webkit-transition: 2s linear all;
5   -moz-transition: 2s linear all;
6   -o-transition: 2s linear all;
7 }
```

Now, we can simply define the stages of the initial and final css properties in the animation.

```
1 .grown {font-size: 50px;}
2 .animateMe.grown-add {
3   font-size: 16px;
4 }
5 .animateMe.grown-add.grown-add-active {
6   font-size: 50px;
7 }
8 .animateMe.grown-remove {}
9 .animateMe.grown-remove.grown-remove-active {
10  font-size: 16px;
11 }
```

CSS3 Animations

We'll start by adding the `@keyframe` animations that we'll define for the animation. Here, we'll add a background color to highlight the text. We'll also define the `remove` animation for when we pull the highlight away from the text:

```
1 @keyframes animateMe-add {
2   from {font-size: 16px;}
3   to {font-size: 50px;}
4 }
5 @-webkit-keyframes animateMe-add {
6   from {font-size: 16px;}
7   to {font-size: 50px;}
8 }
9 @keyframes animateMe-remove {
10  to {font-size: 50px;}
11  from {font-size: 16px;}
12 }
13 @-webkit-keyframes animateMe-remove {
14  to {font-size: 50px;}
15  from {font-size: 16px;}
16 }
```

All we need to do to apply the animation is include the animation CSS style on our classes:

```
1 .animateMe.grown-add {
2   -webkit-animation: 2s animateMe-add;
3   -moz-animation: 2s animateMe-add;
4   -o-animation: 2s animateMe-add;
5   animation: 2s animateMe-add;
6 }
7 .animateMe.grown-remove {
8   -webkit-animation: 2s animateMe-remove;
9   -moz-animation: 2s animateMe-remove;
10  -o-animation: 2s animateMe-remove;
11  animation: 2s animateMe-remove;
12 }
```

JavaScript animation

When animating with javascript, we'll need to define the `addClass` and `removeClass` properties on our animation description object.

```

1 angular.module('myApp')
2 .animation('.animateMe', function() {
3   return {
4     addClass: function(ele, clsName, done) {
5       {
6         // Example to show how to animate
7         // with jQuery. Note, this requires
8         // jQuery to be included in the HTML
9         if (clsName === 'grown') {
10           $(ele).animate({
11             'font-size': '50px'
12           }, 2000, done);
13           } else { done(); }
14         },
15         removeClass: function(ele, clsName, done) {
16           {
17             if (clsName === 'grown') {
18               $(ele).animate({
19                 'font-size': '16'
20               }, 2000, done);
21               } else { done(); }
22             }
23           }
24         });

```

Animating ngShow/ngHide

The `ngShow` and `ngHide` directives use the `.ng-hide` class when showing or hiding elements. It's possible to add animations for between the showing and hiding of the DOM elements.

When we are animating on the classes, the animation is fired **first** and when it's complete, the final `.ng-hide` is added to the DOM element.

Because the `ng-hide` directive will still be applied to the DOM element when we're removing the `ng-hide` class, we'll never see our animation until it's complete. Thus, we'll need to tell the CSS to display our class and not cascade.

The `ngShow` and `ngHide` directives fire the events:

Action	Event name
After <code>ngClass</code> evaluates to a truthy value and before the class has been added	add
Fired just before the class	remove

Action	Event name
is removed	

For the following `ngHide` examples, we're going to work with the HTML:

CSS3 Transitions

To animate items in the `ngHide`, we'll need to add the CSS class that will define the initial state of the element and the class that will define the final state for both of the enter and edit states.

```

1 .animateMe.ng-hide-add,
2 .animateMe.ng-hide-remove {
3   transition: 2s linear all;
4   -webkit-transition: 2s linear all;
5   -moz-transition: 2s linear all;
6   -o-transition: 2s linear all;
7   display: block !important;
8 }
```

Notice the last line in the CSS block. This tells the CSS to render this class and no other fallback class property for the `display` property. With out this, the element won't show.

Now, we can simply define the stages of the initial and final css properties in the animation.

```

1 .animateMe.ng-hide-add {
2   opacity: 1;
3 }
4 .animateMe.ng-hide-add.ng-hide-add-active
5 {
6   opacity: 0;
7 }
8 .animateMe.ng-hide-remove {
9   opacity: 0;
10}
11 .animateMe.ng-hide-remove.ng-hide-remove-active {
12   opacity: 1;
13}
```

CSS3 Animations

We'll start by adding the `@keyframe` animations that we'll define for the animation. Here, we'll add a background color to highlight the text. We'll also define the `remove` animation for when we pull the highlight away from the text:

```
1 @keyframes animateMe-add {
2   from {opacity: 1;}
3   to {opacity: 0;}
4 }
5 @-webkit-keyframes animateMe-add {
6   from {opacity: 1;}
7   to {opacity: 0;}
8 }
9 @keyframes animateMe-remove {
10  from {opacity:0;}
11  to {opacity:1;}
12 }
13 @-webkit-keyframes animateMe-remove {
14  from {opacity:0;}
15  to {opacity:1;}
16 }
```

All we need to do to apply the animation is include the animation CSS style on our classes:

```
1 .animateMe.ng-hide-add {
2   -webkit-animation: 2s animateMe-add;
3   -moz-animation: 2s animateMe-add;
4   -o-animation: 2s animateMe-add;
5   animation: 2s animateMe-add;
6 }
7 .animateMe.ng-hide-remove {
8   -webkit-animation: 2s animateMe-remove;
9   -moz-animation: 2s animateMe-remove;
10  -o-animation: 2s animateMe-remove;
11  animation: 2s animateMe-remove;
12  display: block !important;
13 }
```

JavaScript animation

When animating with javascript, we'll need to define the `addClass` and `removeClass` properties on our animation description object.

```
1 angular.module('myApp')
2   .animation('.animateMe', function() {
3     return {
4       addClass: function(ele, clsName, done) {
5         {
6           // Example to show how to animate
7           // with jQuery. Note, this requires
8           // jQuery to be included in the HTML
9           if (clsName === 'ng-hide') {
10             $(ele).animate({
11               'opacity': 0
12             }, 2000, done);
13             } else { done(); }
14           },
15           removeClass: function(ele, clsName, done) {
16             {
17               if (clsName === 'ng-hide') {
18                 $(ele).css('opacity', 0);
19                 // Force the removal of the ng-hide
20                 // class so we can actually show the
21                 // animation
22                 $(ele).removeClass('ng-hide');
23                 $(ele).animate({
24                   'opacity': 1
25                   }, 2000, done);
26                   } else { done(); }
27                 }
28               }
29     });
});
```

Building custom animations

The `$animate` service provides hooks for us to implement our own custom animation inside of our own directives. After injecting the `$animate` service in our own apps, we can use the exposed events to trigger associated functions on the `$animate` object for each event.

To get started with animations in our own directives, we'll need to *inject* the `$animate` service in our directives.

```
1 angular.module('myApp', ['ngAnimate'])
2 .directive('myDirective', function($animate) {
3   return {
4     template: '<div class="myDirective"></div>',
5     link: function(scope, ele, attrs) {
6       // Add animations here
7       // for instance:
8       $animate['addClass'](element, 'ng-hide');
9     }
10   }
11 });

```

Now that we have our directive with `$animation` injected, we can bind events to the directive and start showing our animations.

With our directive set up, we can now create an animation that we can correspond with our directive calling the `$animate` function.

```
1 angular.module('myApp')
2 .animation('.scrollerAnimation', function() {
3   return {
4     animateFun: function(element, done) {
5       // We are free to do what we want inside
6       // this function, but we must call
7       // done to let angular know we're done
8       // animating
9     }
10   }
11 });

```

The `$animate` service exposes several methods to provide *hooks* into the animation events for the built-in directives. These events that are exposed as hooks in the `$animate` service are:

- enter
- leave
- move
- addClass
- removeClass

The `$animate` service provides these events as functions that enable us to control how we work with them from within our own directives.

addClass()

The `addClass()` method triggers a custom animation event based off the ‘`className`’ variable and attaches the ‘`className`’ value to the element as a CSS class. When adding a class to a DOM element, the `$animate` service will add a suffix to the `className` with `-add` to allow for us to set up animation.

Note: If there are no CSS transitions or keyframe animations defined on the CSS selector: `[className]-add`, then it won’t trigger the animation, it will only add the class.

The `addClass()` method takes three parameters:

- `element` (jQuery/jqLite element)

This is the element that will be animated.

- `className` (string)

This is the CSS class that will be animated and attached to the element.

- `done` (function)

This is the callback function that will be called when the animation has completed.

```
1 angular.module('myApp', ['ngAnimate'])
2 .directive('myDirective', function($animate) {
3   return {
4     template: '<div class="myDirective"></div>',
5     link: function(scope, ele, attrs) {
6       ele.bind('click', function() {
7         $animate.addClass(ele, 'greenlight');
8       });
9     }
10   }
11 });
```

Calling the `addClass()` method will run through the following steps:

1. Run any javascript-defined animations on the element
2. The `[className]-add` class is added to the element
3. `$animate` scans the CSS styles for the transition/animation duration and delay properties
4. The `[className]-add-active` class is added to the element’s `classList` (triggering the CSS animation)

5. \$animate waits for the defined duration to complete
6. The animation ends and \$animate removes the two added classes [className]-add and [className]-add-active
7. The className class is added to the element
8. The done() callback function is fired (if defined)

removeClass()

The `removeClass()` method triggers a custom animation event based on the `className` and then removes the CSS class provided by the CSS `className` value. When removing a class to a DOM element, the `$animate` service will add a suffix to the `className` with `-remove` to allow for us to set up animation.

Note: If there are no CSS transitions or keyframe animations defined on the CSS selector: `[className]-remove`, then it won't trigger the animation, it will only add the class.

The `removeClass()` method takes three parameters:

- `element` (jQuery/jqLite element)

This is the element that will be animated.

- `className` (string)

This is the CSS class that will be animated and removed from the element.

- `done` (function)

This is the callback function that will be called when the animation has completed.

```
1 angular.module('myApp', ['ngAnimate'])
2 .directive('myDirective', function($animate) {
3     return {
4         template: '<div class="myDirective"></div>',
5         link: function(scope, ele, attrs) {
6             ele.bind('click', function() {
7                 $animate.removeClass(ele, 'greenlight');
8             });
9         }
10    }
11});
```

A call to the `removeClass()` animation function will cause the following steps:

1. Run any javascript-defined animations on the element
2. The `[className]-remove` class is added to the element
3. `$animate` scans the CSS styles for the transition/animation duration and delay properties
4. The `[className]-remove-active` class is added to the element's `classList` (triggering the CSS animation)
5. `$animate` waits for the defined duration to complete
6. The animation ends and `$animate` removes the three classes `[className]`, `[className]-add` and `[className]-add-active`
7. The `done()` callback function is fired (if defined)

enter()

The `enter()` method appends the element to the parent element in the DOM and then runs the enter animation. When the animation has started, `$animation` service will add the classes `ng-enter` and `ng-enter-active`. This will give the directive a chance to set up the animation.

The `enter()` method takes 4 parameters:

- `element` (jQuery/jqLite element)

This is the element that will be animated.

- `parent` (jQuery/jqLite element)

This is the parent element of the element that will be the focus of the enter animation.

- `after` (jQuery/jqLite element)

This is the sibling element of the element that will be the focus of the enter animation. This is the *previous* element.

- `done` (function)

This is the callback function that will be called when the animation is complete, if defined.

```
1 angular.module('myApp', ['ngAnimate'])
2 .directive('myDirective', function($animate) {
3   return {
4     template: '<div class="myDirective">' +
5       '<h2>Hi</h2></div>',
6     link: function(scope, ele, attrs) {
7       ele.bind('click', function() {
8         $animate.enter(ele, ele.parent());
9       });
10    }
11  }
12});
```

A call to the `enter()` animation function will cause the following steps to run:

1. The element is inserted into the parent element or beside the after element
2. `$animate` runs any JavaScript-defined animations on the element.
3. The `.ng-enter` class is added to the element's `classList`.
4. `$animate` scans the CSS styles for the transition/animation duration and delay properties
5. The `.ng-enter-active` class is added to the element's `classList` (triggers the animation).
6. `$animate` waits for the defined duration to complete
7. The animation ends and `$animate` removes both of the classes `.ng-enter` and `.ng-enter-active` from the element.
8. The `done()` callback function is fired (if defined)

leave()

The `leave()` method runs the leave animation. When it's done running, it removes the element from the DOM. When the animation has started, it will add the `.ng-leave` and `.ng-leave-active` classes to the element.

The `leave()` method takes 2 parameters:

- element (jQuery/jqLite element)

This is the element that will be animated.

- done (function)

This is the callback function that will be called when the animation is complete, if defined.

```
1 angular.module('myApp', ['ngAnimate'])
2 .directive('myDirective', function($animate) {
3   return {
4     template: '<div class="myDirective">' +
5       '<h2>Hi</h2></div>',
6     link: function(scope, ele, attrs) {
7       ele.bind('click', function() {
8         $animate.leave(ele);
9       });
10    }
11  }
12});
```

A call to the `leave()` animation function will cause the following steps to run:

1. `$animate` runs any JavaScript-defined animations on the element.
2. The `.ng-leave` class is added to the element's `classList`.
3. `$animate` scans the CSS styles for the transition/animation duration and delay properties
4. The `.ng-leave-active` class is added to the element's `classList` (triggers the animation).
5. `$animate` waits for the defined duration to complete
6. The animation ends and `$animate` removes both of the classes `.ng-leave` and `.ng-leave-active` from the element.
7. The element is removed from the DOM
8. The `done()` callback function is fired (if defined)

move()

The `move()` function fires the move DOM animation. Before the animation starts, the `$animate` service will either append it into the parent container or add the element directive after the element, if present. Once the animation has started, the `.ng-move` and `.ng-move-active` will be added for the duration of the animation.

The `move()` method takes 4 parameters:

- element (jQuery/jqLite element)

This is the element that will be animated.

- parent (jQuery/jqLite element)

This is the parent element of the element that will be the focus of the move animation.

- after (jQuery/jqLite element)

This is the sibling element of the element that will be the focus of the enter animation. This is the *previous* element.

- done (function)

This is the callback function that will be called when the animation is complete, if defined.

```
1 angular.module('myApp', ['ngAnimate'])
2 .directive('myDirective', function($animate) {
3   return {
4     template: '<div class="myDirective">' +
5       '<h2>Hi</h2></div>',
6     link: function(scope, ele, attrs) {
7       ele.bind('click', function() {
8         $animate.move(ele, ele.parent());
9       });
10    }
11  }
12});
```

A call to the `move()` animation function will cause the following steps to run:

1. The element is moved into the parent element or beside the after element
2. `$animate` runs any JavaScript-defined animations on the element.
3. The `.ng-move` class is added to the element's `classList`.
4. `$animate` scans the CSS styles for the transition/animation duration and delay properties
5. The `.ng-move-active` class is added to the element's `classList` (triggers the animation).
6. `$animate` waits for the defined duration to complete
7. The animation ends and `$animate` removes both of the classes `.ng-move` and `.ng-move-active` from the element.
8. The `done()` callback function is fired (if defined)

Integrating with third-party libraries

Animate.css

The `Animate.css` library provides a bunch of cool, fun, and cross-browser animations out of the box. It's a great library that gives a lot of power without needing to do much work.

Luckily, the Angular community has provided a slick method of including the Animate.css classes into our angular app. To use the Animate.css shim, download the `animate.css` and `animate.js` from [https://github.com/yearofmoo/ngAnimate-animate.css¹⁰⁰](https://github.com/yearofmoo/ngAnimate-animate.css). Reference both of them in our HTML, like so:

```
1 <!-- In the HEAD of our HTML -->
2 <link rel="stylesheet" type="text/css" href="css/animate.css">
3 <!-- In the BODY of our HTML -->
4 <script type="text/javascript" src="js/vendor/animate.js"></script>
```

Now, instead of requiring `ngAnimate` as a dependency for our app, we can simply include `ngAnimate-animate.css` as a dependency. This is because the `ngAnimate-animate.css` module requires the `ngAnimate` module by default.

With this set, we can simply reference the animate classes with the `ng-class` directive. For instance:

```
1 <div class="animateMe"
2   ng-class="{ 'dn-fade' : dn_fade }">
3 </div>
```

For a list of every animation possibility, check out the [README¹⁰¹](#).

TweenMax/TweenLite

TweenLite and TweenMax are fantastic libraries. They are a slick library that was modeled after the ActionScript animation properties. To use the library, we'll need to make sure we download the Greensock library.

Download the library from [Greensock¹⁰²](#) and store it in a place accessible to your `index.html`. We recommend storing it in `js/vendor/TweenMax.min.js`. We'll then need to make sure to reference the TweetMax library in our page:

```
1 <script type="text/javascript" src="js/vendor/TweenMax.min.js"></script>
```

With that set, we're all ready to go. To include the Greensock animations in our app, we'll need to set up our animations to use JavaScript. In this way, there is little to no integration code necessary beyond simply animating with javascript:

¹⁰⁰<https://github.com/yearofmoo/ngAnimate-animate.css>

¹⁰¹<https://github.com/yearofmoo/ngAnimate-animate.css/blob/master/README.md>

¹⁰²<http://www.greensock.com/>

```
1 angular.module('myApp', ['ngAnimate'])
2 .animation('scrollAside', function($window) {
3   return {
4     enter: function(element, done) {
5       TweenMax.set(element, {
6         position: 'relative'
7       });
8       TweenMax.to(element, 1, {
9         opacity: 0,
10        width: 0
11      });
12      $window.setTimeout(done, 2000);
13    }
14  }
15});
```

The digest loop and \$apply

Let's take a peek and look at how Angular works a bit underneath the hood. How do we get this *magical* data-binding to work in only a few lines of code?

Probably one of the most important pieces of angular to know is how the \$digest loop works and how to use the \$apply() method.

In the *normal* browser flow, a browser will execute callbacks that are registered with an event that occurs, like clicking on a link.

Events are fired when the page is loaded, when an \$http request comes back, when the mouse moves or a button is clicked, etc.

When an event is fired/triggered, javascript creates an event object and executes any functions listening for the specific events with this event object. This callback method then runs inside the javascript function, which then returns back to the browser which may or may not update the DOM.



No two events can run at the same time; the browser waits until one event handler finishes before the next handler is called.

In vanilla javascript, sans-angular we can attach a function callback to the click event to a div. Anytime that a click event is found on an element, the function callback gets run:

```
1 var div = document.getElementById("clickDiv");
2 div.addEventListener("click",
3   function(evt) {
4     console.log("evt", evt);
5   });

```



Open the Chrome dev tools and run that inside of any webpage.

Anytime that the browser detects a *click*, the browser will call the function registered with the addEventListener on the document.

When we mix Angular into the flow, it extends this normal browser flow to create an *angular context*. The Angular context refers specifically to code that runs *inside* the Angular event loop, the \$digest loop.

To really understand the *angular context*, we'll need to look at exactly what goes on inside of it. There are two major components of the \$digest loop:

- The \$watch list
- The \$evalAsync list

\$watch list

Every time that we track an event in the view, we are registering a callback function that we intend to get called when an event happens in the page. Recall our first example:

```
1 <!DOCTYPE html>
2 <html ng-app>
3 <head>
4   <title>Simple app</title>
5   <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.0rc1/angular.js"></script>
6 </head>
7 <body>
8   <input ng-model="name" type="text" placeholder="Your name">
9   <h1>Hello {{ name }}</h1>
10 </body>
11 </html>
```

Anytime that the input field is updated, `{{ name }}` changes in the UI. This happens because we bind the input field in the UI to the `$scope.name` property. In order to update the view, Angular needs to *track* the change. It does this by adding a *watch* function to the `$watch` list.

Properties that are on the `$scope` object are *only* bound if they are used in the view. In the case above, we added a single function to the `$watch` list.

Remember: for *all* UI elements that are bound to a `$scope` object, a `$watch` will be added to the `$watch` list.

These `$watch` lists are resolved in the `$digest` loop through a process called `dirty checking`.

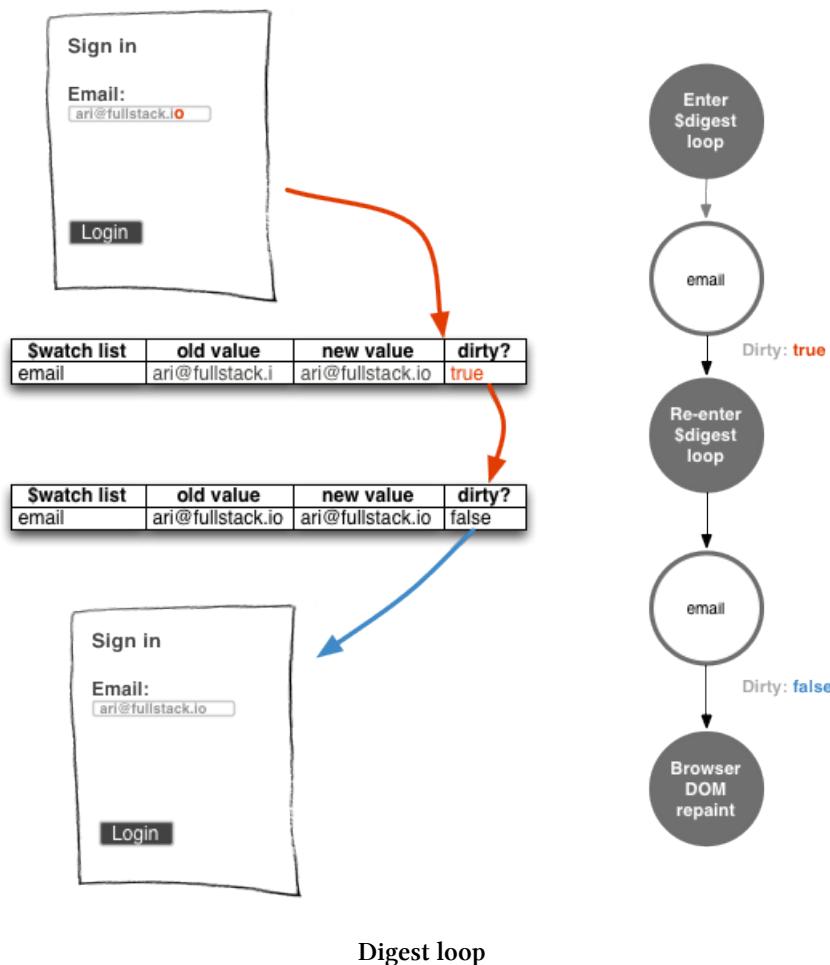
Dirty checking

Dirty checking is a simple process that boils down to a very basic concept to check if a value has changed that hasn't been synchronized across the app.



The dirty checking strategy is commonly used in a lot of different applications, beyond Angular. Game engines, database engines, and Object Relational Mappers (ORMs) are some examples of such systems.

Our angular app will keep track of the values of the current watches (in the watch object, for the curious). As angular walks down the `$watch` list, if the updated value has *not* changed from the old value, then it will continue down the `$watch` list. If the value has changed, then the app will record the new value and continue down the `$watch` list.



Digest loop

After the `$watch` list has been run through entirely, if any value changed, then the app will fall back into the `$watch` loop until nothing has changed.

Why run the loop all over again? If we update a value in the `$watch` list that updates another value angular won't detect the update unless we rerun the loop.

If the loop runs 10 times or more, our angular app will throw an exception and the app will die. Without this exception getting thrown, our app could get thrown into an infinite loop and bad things will happen.

Note: in future versions of Angular, the framework will use the native browser specification `Object.observe()` which will give the *dirty checking* process quite a speed-up.

\$watch

The `$watch` method on the `$scope` object sets up a `dirty check` on every call to `$digest` inside the Angular event loop. The angular `$digest` loop will always return if it detects changes on the expression.

The `$watch` function itself takes two required arguments and a third optional one:

- `watchExpression`

The `watchExpression` can either be a property of a scope object or a function. This runs on **every** call to `$digest` in the `$digest` loop.

If the `watchExpression` is a string, then it is evaluated in the context of the `$scope`. If it is a function, then it is expected to return the value that should be watched.

- `listener/callback`

The callback listener function will only be called when the current value of the `watchExpression` and the previous value of the expression are not equal (except when it's being initialized on the first run).

- `objectEquality`

The `objectEquality` parameter is a comparison boolean that tells Angular to check for equality, rather than by reference.

The `$watch` function returns a `deregistration` function for the listener that we can call to *cancel* angular from watching the value.

```

1 // ...
2 var unregisterWatch =
3   $scope.$watch('newUser.email',
4     function(newVal, oldVal) {
5       if (newVal === oldVal) return; // on init
6     });
7 // ...
8 // later, we can unregister this watcher
9 // by calling
10 unregisterWatch();

```

If we are done watching the `newUser.email` in this example, we can clean up our watcher by calling the `deregistration` function it returns.

For instance, let's say we want to parse an input field value from a full name to split on spaces and find a simple first and last name. Given that our view looks like:

```
1 <input type="text" ng-model="full_name" placeholder="Enter your full name" />
```



Note: we should **never** use `$watch` in a controller as it makes it difficult to test the controller.
We're illustrating this for simplicity and will later move these watches into services.

We'll want to set up a `$watch` listener on the `full_name` property and detect for any changes to the value. We'll set the `$watch` function on the `full_name` property.

```

1 angular.module("myApp")
2 .controller("MyController", ['$scope', function($scope) {
3   $scope.$watch('full_name', function(newVal, oldVal, scope) {
4     // the newVal of the full_name will be available here
5     // while the oldVal is the old value of full_name
6   });
7 }]);

```

In our example, we're setting an AngularJS expression that tells our angular app to "watch the `full_name` property for any potential changes on it and run the function if you detect any changes".

The listener function is called once on initialization, so the first time around, the value of `newVal` and `oldVal` will be `undefined` (and will be equal). This being the case, it's generally good to check inside the expression if we're in the initialization phase or if there is an update to the previous value. We can easily accomplish this inside the function, like so:

```
1 $scope.$watch('full_name',
2   function(newVal, oldVal, scope) {
3     if (newVal === oldVal) {
4       // This will run only on the initialization of the watcher
5     } else {
6       // A change has occurred after initialization
7     }
8   });

```

The `$scope.$watch()` function sets up a `watchExpression` on the `$scope` for 'full_name'.

\$watchCollection

Angular also allows for us to set shallow watches for object properties or elements of an array and fire the listener callback whenever the properties change.

Using `$watchCollection` allows us to detect when there are changes on an object or array, which gives us the ability to determine when items are added, removed, or moved in the object/array. `$watchCollection` works just like normal `$watch` works in the `$digest` loop and we can treat it as a normal `$watch`.

The `$watchCollection()` function takes two parameters:

- `obj` (string/function)

The `obj` to watch. If a string is passed in, then it will be evaluated as an Angular expression. If a function is passed in, it will be called with the current `scope` and will be expected to return the value to watch.

- `listener` (function)

The callback function that will be fired when the collection changes. Similar to the `$watch` function, this function will be called with the new collection fired from the `$watch`, the old collection (a copy of the previous collection), and the `scope` it will be executed within.

The `$watchCollection()` function returns a *deregistration* function that, when called cancels our `$watch` on the collection.

```
1 $scope.$watchCollection('names',
2   function(newNames, oldNames, scope) {
3     // Our names collection has changed
4   });

```

The \$digest loop in a page

Inside of a web page, this process works a bit like this. Assuming we have a (very insecure) login page that features a single name field to log in with a single form validation.



We do not recommend this as an insecure form of authentication.

```
1 <h2>Sign in</h2>
2 <input
3   type="text"
4   placeholder="Your name"
5   ng-model="name"
6   ng-minlength='3' />
7 <input type="submit"
8   ng-click="login()"
9   value="Login" />
```

With the *name* bound in the view through the `ng-model` directive, angular sets up an implicit *watcher* to bind the value of the input field to the current `$scope`.

When the user inputs a character into the form, the angular context kicks in and start iterating through the `$$watchers` (the `$watch` list).

In this case, the `$watch` list consists of a single element: `$scope.name`. Since the user has changed the input field by a single character, the watch function will execute on the `$scope.name` binding. This triggers the validations and formatters to run on the value (because of the `ng-model` binding) before we exit the `$digest` loop.

Since a value *has* changed in the digest loop, angular will need to re-run the loop to confirm that it did not change any other values on the scope.



Why run the digest loop again? If we have a value called `$scope.full_name` which consists of `$scope.first_name + $scope.last_name`, any change to either of these values will change `$scope.full_name` so the loop needs to run again to ensure that nothing else has changed.

Since changing the `$scope.name` attribute does not change any other attribute in the `$scope`, the `$digest` loop will exit and the browser will call repaint on the DOM.

After the user has entered their name in the input field and click on the submit button. This will cause a slightly different flow to happen.

`ng-click` binds the browser's native `click` event to the DOM element. When that DOM element receives the click event, the `ng-click` directive calls `$scope.$apply()` and then we enter the `$digest` loop.

\$evalAsync list

The `$evalAsync()` method is a way to schedule running expressions on the current scope sometime in the future. The second operation that the `$digest` loop runs is executing the `$$asyncQueue`. We can get access to this work queue with the `$evalAsync()` method.

Through the `$digest` loop, the queue will empty between each loop through the dirty checking life-cycle. This means two things for any function called with `$evalAsync`:

- It will execute sometime *after* the function that called it.
- At least *one* `$digest` cycle will be performed after the expression is executed.

The `$evalAsync()` method takes a single argument:

- expression (string/function)

The expression to execute on the current scope. If a string is passed, then it will `$eval` the expression on the scope.

If a function is passed, then it will execute the function with the scope that is passed in as it's executed.

```
1 $scope.$evalAsync('attribute',
2   function(scope) {
3     scope.foo = "Executed"
4   });

```

Some particulars to think about when using `$evalAsync`:

- If a directive calls `$evalAsync()` directly, it will run *after* the DOM has been manipulated by Angular, but *before* the browser renders

- If a controller calls `$evalAsync()`, then it will run *before* the DOM has been manipulated by Angular and before the browser renders (we'll never really want this)

A time when we would want to use the `$evalAsync()` function is any time that we want an action to occur outside of the scope of another action in angular.

We might use this instead of a `setTimeout()` function where it might cause a flicker after the browser re-renders the view.

\$apply

The `$apply()` function is used to execute an expression inside the angular context from outside of the angular framework. For instance, if we implement a `setTimeout()` or are using a third-party library and we want the event to run **inside** the angular context, we must use `$apply()`.

The `$apply()` function takes one optional argument:

- expression (string/function)

The `expression` argument optionally takes a string or a function and executes it inside the current scope.

If a string is passed, then `$apply()` will first call `$eval()` on the string. This will force angular to `$eval()` the string on the local scope context.

If a function is passed, then the function will be executed on the scope passed into the function.

Any exception that is thrown by the `$eval()` method will be caught and handled by the `$exceptionHandler` service. Lastly, the `$apply()` method directly calls the `$digest` loop.

```
1 // Ways to call apply
2 // with a string to be eval'd
3 $scope.$apply('message = "Hello World"');
4 // With a function that is passed a scope
5 $scope.$apply(function(scope) {
6   // Execute this function with the scope
7   scope.message = "Hello World";
8 });
9 // With a function that ignores the scope
10 $scope.$apply(function() {
11   $scope.message = "Hello World";
12 });
13 // Or just force the $digest loop to run
14 // by calling it at the end of our operation
15 $scope.$apply();
```

Simply said, using `$scope.$apply()` is a way to get access to the angular context from outside of it. If we call `$apply()` when an event is fired, we'll run it through the Angular event loop. If we don't call `$apply()`, we won't execute the function in the event loop and it will run outside of the angular context.

When to use \$apply()

We can generally count on any directive that is provided by Angular available in the view to call `$apply()` for us. Any of the `ng-[event]` directives (like `ng-click`, `ng-keypress`, etc) will call `$apply()`.

We can also depend on a lot of built-in angular services to call `$digest()` for us as well. The `$http` service calls `$apply()` after the XHR request is done to trigger updating the return value (promise).

Anytime that we are handling an event manually, using a third-party framework (like jQuery or the Facebook API), or calling `setTimeout()` we'll use the `$apply()` function to tell angular to rerun the `$digest` loop.

It's generally *not* advised to use `$apply()` in a controller as it makes it difficult to test and if we have to use `$apply()` or `$digest()` in a controller, we're probably making things more difficult than they should be.

When integrating jQuery with angular (generally considered *dirty*), we'll **need** to use `$apply()` because it knows nothing about the angular context. For instance, when using a jQuery plugin, like the datepicker we'll need to use apply to get the value from jQuery into our angular app.

Here, we'll build a simple directive (which we'll dive deep into in the [directives chapter](#)), and use the datepicker jQuery plugin function on the element.

The datepicker plugin exposes an event `onSelect` that will get fired when the user picks a date. In order to get access to the date that the user picks *inside* of our angular app, we'll need to run this inside the `$apply()` function.



The `ele.datepicker()` function is a property function made available on the DOM element by the jQuery datepicker plugin. In order to make this work, we'll need to make sure we include jQuery and the jQuery datepicker plugin on the page.



The `ctrl.$setViewValue()` function is a function that is made available when using `ng-model` on a DOM element. For more information, read the [controllers chapter](#).

```
1 app.directive('myDatepicker', function() {
2   return function(scope, ele, attrs, ctrl) {
3     $(function() {
4       // call datepicker on the element
5       ele.datepicker({
6         dateFormat: 'mm/dd/yy',
7         onSelect: function(date) {
8           scope.$apply(function() {
9             ctrl.$setViewValue(date);
10            });
11          }
12        })
13      });
14    }
15  });
});
```

Demystifying angular

At its core, AngularJS webapps load in the browser the same as non-AngularJS apps, however they do run a little differently. The browser loads the AngularJS library as it is building the DOM (as any javascript libraries normally are loaded).

When the browser fires the `DOMContentLoaded` event, Angular goes to work. It starts by looking for the `ng-app` directive (more on this in the next chapter).



AngularJS will also start initializing when the `angular.js` script is loaded *if* the `document.readyState` is set to `complete`. This is useful if we want to dynamically link the `angularjs` script.

If the `ng-app` directive is found in the DOM, then the app will be *automatically* bootstrapped for us. If it's not found, Angular will expect us to bootstrap the app manually.

To manually bootstrap an AngularJS app, we can use the `angular` method `bootstrap()`. It makes sense to manually bootstrap an application in some relatively rare cases, for instance, let's say you want to run an AngularJS app after some other library code runs or you are dynamically creating an element on the fly.

To manually bootstrap an app, we can bootstrap it, like so:

```
1 var newElement = document.createElement("div");
2 angular.bootstrap(newElement, ['myApp']);
```



If there is no `ng-app` directive found in the DOM *and* we have not manually bootstrapped our app, AngularJS will not run. This can definitely cause some issues if you forget to use `ng-app`.

If there is no application specified in the `ng-app` attribute, then angular will load the app without a specific module. If one is specified, then angular will load the module associated with the directive.

Using `ng-app` without specifying a module:

```
1 <html ng-app>
2 </html>
```

Using `ng-app` with a specified module:

```
1 <html ng-app="moduleName">  
2 </html>
```

Angular will use it to configure the `$injector` service (which we will talk about in the [dependency injection chapter](#) in-depth).

Once the application has been loaded, the `$injector` will create the `$compile` service along with the app's `$rootScope`.

Following the `$rootScope`'s creation, the `$compile` service takes over. It links the `$rootScope` with the existing DOM and will start to compile the DOM beginning from where the `ng-app` directive is set as the root.

How the view works

When the browser gets it's HTML in the normal web flow, it will receive the HTML and parse it into a DOM tree. Each element in the DOM tree, called a DOM element will build a bunch of nodes. The browser is then responsible for laying it out, from the parent elements then down to it's children.

When the browser fetches scripts (from the `<script>` tag), it will pause the parsing and wait until the script is retrieved (it's possible to modify this behavior, but this is the default).

When the `angular.js` script is retrieved, it is executed and it sets up an event listener to listen for the `DOMContentLoaded` event to get fired.



The `DOMContentLoaded` event is fired when the HTML document has been completed loaded and parsed.

When the event is detected, angular will look for the `ng-app` directive and creates the several necessary components it needs to run (the `$injector`, the `$compile` service, and the `$rootScope`) and then it will start parsing the DOM tree.

Compilation phase

The `$compile` service traverses the DOM and collects all of the directives that it finds. It then combines all of their linking functions into a single linking function.

This *linking* function is then used to link the compiled template to the `$rootScope` (the scope attached to the `ng-app` DOM element).

It *finds* the directives in the DOM by looking through the attributes, comments, the classes, and the DOM element name.

```

1 <span my-directive></span>
2 <span class="my-directive"></span>
3 <my-directive></my-directive>
4 <!-- directive: my-directive -->
```



For more in-depth directive coverage, check out the [directives](#) chapter.

The `$compile` service walks the DOM tree looking for DOM elements with *directives* declared. When it encounters a DOM element with one or more directives, it will order the directives (based upon their priority), use the `$injector` service to find and collect the directive's `compile` function and execute it.

The `compile` function in directives is the appropriate time to do any DOM transformations or inline templating as it will create a *clone* of the template.

```

1 // Returns a linking function
2 var linkFunction = $compile(appElement);
3 // Calls the linking function, attaching
4 // the $rootScope to the domElement
5 linkFunction($rootScope);
```

After the `compile` method runs per node, the `$compile` service will call the linking functions. The linking function set up `watches` on the directives that are bound to the enclosing scope. This is responsible for creating the *live-view*.

Finally, after the `$compile` service is complete, the AngularJS runtime is ready to go.

Runtime

In a normal browser flow, the event loop waits for events (like the mouse moving, clicking, a keypress, etc). As these events occur, they are placed into the browser's event queue. If any function handlers are set to react to the event, then they are called with the event as a parameter.

```
1 ele.addEventListener('click', function(event) {});
```

With AngularJS, the event loop is augmented a bit as Angular provides its own event-loop. Directives themselves register event listeners so that when events are fired, the directive function is run in AngularJS `$digest` loop.



The angular event loop is called the `$digest` loop. The `$digest` loop is composed of two small loops, the `evalAsync` loop and the `$watch` list.

When the event is fired, it is called within the context of the directive, which is in the AngularJS context. Functionally, AngularJS calls the directive within an `$apply()` method on the containing scope. This entire process is kicked off by Angular starting its `$digest` cycle on the `$rootScope`, which propagates through to all of the child scopes.

When Angular falls into the `$digest` loop, it waits for the `$evalAsync` queue to empty before it hands back the callback execution context to the browser. The `$evalAsync` is used to schedule any work that needs to be run outside of the current stack frame, but before the browser renders.

Additionally, it waits for the `$watch` expression list, an array of potential expressions that *may* change during the previous iteration. *If* a change is detected, then the `$watch` function is called and the `$watch` list is run over again to ensure that nothing has changed.



Note that for any changes detected in the `$watch` list, AngularJS will run through the list again to ensure that nothing has changed.

Once the `$digest` loop settles down and no potential changes are detected, the execution leaves the Angular context and it passes normally back to the browser where the DOM will be rendered.

This entire flow happens between every browser event, which is why Angular can be so powerful. It is also possible to inject events from the browser into the AngularJS flow.

Essential AngularJS extensions

One of the most popular and supported AngularJS plugins is the the angular-ui framework.

AngularUI

AngularJS itself is packed full of features out-of-the-box that we can use to build an expressive AngularJS app without relying on separate libraries. However, the **very active** AngularJS community has built some great libraries that we can take advantage of to maximize the power of our apps.

In this section, we're covering several different and most commonly used components of the [AngularUI¹⁰³](#) library.

The AngularUI library has been broken out into several modules so that rather than including the entire suite, we can pick and choose the components that we're interested in using.

As we walk through the different components, we'll need to ensure that we install each of the different components we are going to work with.

¹⁰³<http://angular-ui.github.io/>

Installation

For each component, we can either download the individual javascript library and place it in our application path *or* we can use [bower¹⁰⁴](#) to install them. We at [ng-newsletter.com¹⁰⁵](#) recommend using bower. For the purposes of this chapter, we'll cover installing each module only with bower.

ui-router

One of the most useful libraries that the AngularUI library gives us is the ui-router. It's a routing framework that allows us to organize our interface by a state machine, rather than a simple url route.

Note that at time of this writing, the ui-router library may change, although there is a lot of support for the API.

Installation

To install the ui-router library, we can either download the [release¹⁰⁶](#) or use bower to install the library.

Make sure you have bower installed globally:

```
1 $ npm install bower -g
```

User bower to install the angular-ui library:

```
1 $ bower install angular-ui-router --save
```

We'll need to make sure we link the library to our view:

```
1 <script type="text/javascript" src="app/bower_components/angular-ui-router/releas\ 
2 e/angular-ui-router.js"></script>
```

And we'll need to inject the ui.router as a dependency in our app:

```
angular.module('myApp', ['ui.router']) ``
```

Now, unlike the built-in ngRoute service, the ui-router can nest views as it works based off states, rather than simply a url.

Instead of using the ng-view directive as we do with the ngRoute service, we'll use the ui-view directive with ngRoute.

When dealing with routes and states inside of ui-router, we're mainly concerned with which state the application is in along with what route the webapp is at.

¹⁰⁴<http://bower.io/>

¹⁰⁵<http://ng-newsletter.com>

¹⁰⁶<http://angular-ui.github.io/ui-router/release/angular-ui-router.js>

```
1 <div ng-controller="DemoController">
2   <div ui-view></div>
3 </div>
```

Just like `ngRoute`, the templates we define at any given state will be placed inside of the `<div ui-view></div>` element. Each of these templates can include their own `ui-view` as well. This is how we can have *nested* views inside our routes.

To define a route, we'll use the `.config` method, just like normal, but instead of setting our routes on `$routeProvider`, we'll set our *states* on the `$stateProvider`.

```
.config(function($stateProvider, $urlRouterProvider) { $stateProvider .state('start', { url: '/start', templateUrl: 'partials/start.html' })});"
```

This assigns the *state* named `start` to the state configuration object. The state configuration object, or the `stateConfig`, has similar options to the route configuration object which is how we can configure our states.

template, templateUrl, templateProvider

The three ways to set up templates on each of the views are with one of the following options:

- `template` - String of HTML content or a function that returns HTML
- `templateUrl` - A string URL path to a template or a function that returns a URL path string
- `templateProvider` - a function that returns HTML content string

For instance:

```
$stateProvider.state('home', { template: '<h1>Hello {{ name }}</h1>' });"
```

controller

Just like in `ngRoute`, we can either associate an already registered controller with a url (via a string) or we can create a controller function that operates as the controller for the state.

If there is no template defined (in one of the previous options), then the controller will **not** be created.

resolve

We can resolve a list of dependencies that we can inject into our controller using the `resolve` functionality. In `ngRoute`, the `resolve` option allows us to resolve promises before the route is actually rendered. Inside `angular-route`, we have a bit more freedom as to how we can use this option.

The `resolve` option takes an object where the keys are the names of the dependency to inject into the controller and the values are the factories that are to be resolved.

If a string is passed, then `angular-route` will try to match an existing registered service. If a function is passed then the function is injected and the return value of the function is the dependency. If the function returns a promise, it is resolved *before* the controller is instantitated and the value (just like `ngRoute`) is injected into the controller.

```
$stateProvider.state('home', { resolve: { // This will return immediately as the // result is not a promise
  person: function() { return { name: "Ari", email: "ari@fullstack.io" } }, // This function returns a
  promise, therefore // it will be resolved before the controller // is instantiated
  currentDetails: function($http) { return $http({ method: 'JSONP', url: '/current_details' }); }, // We can use the resulting
  promise in another // resolution
  facebookId: function($http, currentDetails) { $http({ method: 'GET',
  url: 'http://facebook.com/api/current_user', params: { email: currentDetails.data.emails[0] } }) } },
  controller: function($scope, person, currentDetails, facebookId) { $scope.person = person; } })"
```

url

The `url` option will assign a url that the application is in to a specific state inside our app. This way we can get the same features of deep linking, while navigating around the app by state, rather than simply by URL.

This is similar to the `ngRoute` `url`, but can be thought of as a major upgrade, as we'll see in a moment.

The basic route can be specified like so:

```
$stateProvider .state('inbox', { url: '/inbox', template: '<h1>Welcome to your inbox</h1>' })";"
```

When the user navigates to `/inbox`, then the `inbox` state will be transitioned into and the main `ui-view` directive will be filled with the contents of the template (`<h1>Welcome to your inbox</h1>`).

The `url` can take several different options that makes it incredibly powerful. We can set the basic parameters in the `url` like in `ngRoute`:

```
$stateProvider .state('inbox', { url: '/inbox/:inboxId', template: '<h1>Welcome to your inbox</h1>',
  controller: function($scope, $stateParams) { $scope.inboxId = $stateParams.inboxId; } })";"
```

The `:inboxId` will then be captured as the second component in the URL. For instance, if the user is transitioned to `/inbox/1`, then `$stateParams.inboxId` would become 1 (as `$stateParams` will be `{inboxId: 1}`).

We can also use a different syntax if you prefer:

```
url: '/inbox/{inboxId}'";"
```

The path must match exactly to the `url`. Unlike `ngRoute`, if the user navigates to `/inbox/`, the above path *will* work, however `/inbox` the above state will not be activated.

The path will also enable us to use regex inside of parameters so we can set a rule to match against our route. For instance:

```
// Match only inbox ids that contain // 6 hexadecimal digits
url: '/inbox/{inboxId:[0-9a-fA-F]{6}}', //
Or // match every url at the end of /inbox // to inboxId (a catch-all)
url: '/inbox/{inboxId:.*}'";"
```

Note, we cannot use capture groups inside the route as the route resolver will not be able to resolve the route.

We can even specify query parameters in our route:

```
// will match a route such as // /inbox?sort=ascending url: '/inbox?sort' ""
```

Nested routing

We can use the `url` parameter to use appended routes to provide for nested routes. This enables us to support having multiple `ui-views` inside our page and our templates. For instance, we can nest individual routes inside of our `/inbox` route above.

```
$stateProvider .state('inbox', { url: '/inbox/:inboxId', template: '<div><h1>Welcome to your inbox</h1>\<div ui-view></div>\</div>', controller: function($scope, $stateParams) { $scope.inboxId = $stateParams.inboxId; } }) .state('inbox.priority', { url: '/priority', template: '<h2>Your priority inbox</h2>' }); ""
```

Our first route will match as we expect from above. We now will have a second route, a child route that will match underneath the `inbox` route (because we specified it as a child using the `.` syntax).

`/inbox/1` will match the first state, and `/inbox/1/priority` will match the second state. With this syntax, we can support a nested url inside of the parent route. The `ui-view` inside of the parent view will resolve to the `priority` inbox.

params

The `params` option is an array of parameter names or regexes. This option **cannot** be combined with the `url` option. When the state becomes active, these parameters will be populated in the `$stateParams` service.

views

We can set multiple different *named* views inside of a state. This is a particularly powerful feature of `ui-router`. Inside of a single view, we can define multiple views that we can reference inside of a single template.

If we set the `views` parameter, then the state's `templateUrl`, `template`, and `templateProvider` will be ignored. If we want to include a parent template in our routes, we'll need to create an abstract template the contains a template.

Let's say we have a view that looks like:

```

1 <div>
2   <div ui-view="filters"></div>
3   <div ui-view="mailbox"></div>
4   <div ui-view="priority"></div>
5 </div>

```

We can now create *named* views that will fill each of these individual templates. Each of the subviews can contain their own templates, controllers, and resolve data.

```
$stateProvider .state('inbox', { views: { 'filters': { template: '<h4>Filter inbox</h4>', controller: function($scope) {} }, 'mailbox': { templateUrl: 'partials/mailbox.html' }, 'priority': { template: '<h4>Priority inbox</h4>', resolve: { facebook: function() { return FB.messages(); } } } }]); ""
```

abstract

An abstract template can never be directly activated, but can set up descendants that are activated.

We can use an abstract template to provide a template wrapper around multiple named views, or to pass \$scope objects to descendant children. We can use them to pass around resolved dependencies or custom data or simply to nest several routes under the same 'url' (like have all routes under the /admin url).

Setting up an abstract template is just like setting up a regular state, except that we'll set the `abstract` property:

```
$stateProvider .state('admin', { abstract: true, url: '/admin', template: '<div ui-view></div>' })
.state('admin.index', { url: '/index', template: '<h3>Admin index</h3>' })
.state('admin.users', { url: '/users', template: '<ul>...</ul>' }); ""
```

onEnter, onExit

These callbacks are called during the life cycle of our views as to when they are transitioned into and out of. For both options, we can set a function to be called. These functions have access to the resolved data.

These callbacks give us the ability to trigger an action on a new view or before we head out to another state. This is a good way to launch an "Are you sure?" modal view or request user log in before they head into this state.

data

We can attach arbitrary data to our state configObject. This is similar to the `resolve` property, except that this data will **not** be injected into the controller, nor will promises be resolved.

This is particularly useful when passing data to child states from a parent state.

Events!

Like the `ngRoute` service, the `angular-route` service fires events at different times during the state lifecycle.

We can attach onto these different events inside of our application by listening on the `$scope`. All of the following events are fired on the `$rootScope`, so we can listen to these events on any of our `$scope` objects.

State change events

We can listen to the events as follows:

```
$scope.$on('$stateChangeStart', function(evt, toState, toParams, fromState, fromParams), { // We can prevent this state from completing evt.preventDefault(); }); ""
```

The events that get fired as follows:

\$stateChangeStart This is fired when the transition from one state to another **begins**.

\$stateChangeSuccess This is fired when the transition from one state to the next is **completed**.

\$stateChangeError This is fired when an error happens during the transition. This is usually caused by either a template that cannot be resolved or a resolve promise failing to resolve.

View load events

The `ui-router` provides events at the view loading stage.

\$viewContentLoaded This event is fired when the view begins loading and happens before the DOM is rendered.

We can listen to this event like so:

```
$scope.$on('$viewContentLoaded', function(event, viewConfig){ // Access to all the view config properties. // and one special property 'targetView' // viewConfig.targetView}); ""
```

\$viewContentLoaded This event is fired after the view has been loaded and after the DOM is rendered.

\$stateParams

Above, we used the `$stateParams` to pick out the different params options from the url parameters. The service is how we'll handle data from different components of our url.

For instance, if we have a url in our `inbox` state that looks like:

```
url: '/inbox/:inboxId/messages/{sorted}?from&to' ``
```

And our user finds their way to the route:

```
/inbox/123/messages/ascending?from=10&to=20 ``
```

Then our `$stateParams` object will result in:

```

1  #### $urlRouterProvider
2
3  Just like `ngRoute`, we have access to a route provider that we can build rules a\
4  round what happens when a particular url is activated.
5
6  The states that we create are responsible for handling activating themselves at d\
7  ifferent urls, so the `'$urlRouterProvider` is not necessary to manage activating \
8  and loading states. It does come in handy when we want to manage events that happ\
9  en outside of the scope of our states, such as with redirection or authentication\
10 .
11
12 > We can use the `'$urlRouterProvider` in our module's config function.
13
14 ##### `when()`
15
16 The when function takes two parameters, the incoming path that we want to match o\
17 n and the path that we want to redirect to or a function that gets invoked when t\
18 he path is matched.
19
20 To set up redirection, we'll set the `when` method to take a string.
21
22 For instance, if we want to redirect an empty route to our `/inbox` route:
23
24 {lang="javascript"}
25 .config(function($urlRouterProvider) {
26   $urlRouterProvider.when('', '/inbox');
27 });

```

If we pass a function, then it will get invoked when the path has matched. The handler can return one of three values:

- **falsy** - this tells the \$urlRouter that the rule didn't match and it should try finding a different state that does match. This can be useful if we want to ensure a user has valid access to a url.
- **a string** - The \$urlRouter will treat a string value as a redirect url.
- **truthy or undefined** - this lets the \$urlRouter know that we've handled the url.

`otherwise()` Just like the `otherwise()` method in `ngRoute`, the `otherwise()` method will redirect a user if no other routes are matched. This is a good method of creating a *default* url, for instance.

The `otherwise()` method takes a single parameter, either a string or a function.

If we pass in a string, then any invalid or unmatched routes will be redirected to the string as a specified url.

If we pass in a function, it will get invoked if no other route is matched and we'll be responsible for handling the return.

```
.config(function($urlRouterProvider) { $urlRouterProvider.otherwise('/'); // or $urlRouterProvider.otherwise(  
function($injector, $location) { $location.path('/');});}); ""
```

`rule()` If we want to bypass any of the url matching or want to do some route manipulation before other routes, we can use the `rule()` function or manipulating our routes.

We must return a valid path as a string when using the `rule()` function.

```
app.config(function($urlRouterProvider){ $urlRouterProvider.rule( function($injector, $location) {  
return '/index';});}); ""
```

Create a wizard

Why does it matter if we use a new, more powerful router than the built-in `ngRoute` provider?

A useful case for using the `ui-router` is when we want to create a sign-up wizard for our users to walk them through the process of signing up for our service.

Using the `ui-router`, we'll create a quick signup service with a single controller than can handle the sign-up.

First, we'll create a view for the app:

```
1 <div ng-controller="WizardSignupController">  
2   <h2>Signup wizard</h2>  
3   <div ui-view></div>  
4 </div>
```

Inside of this view, we'll house our signup views. Next, in our signup wizard we'll need three stages:

- **start** - at this stage, we'll take the user's name and introduce them to our signup wizard
- **email** - here, we'll take the user's email in the second step
- **finish** - and at this state, the user will have completed our signup process and we'll simply show them a complete page.

In a *real* app, the **finish** stage would likely send the registration data back to a server and handle real registration. Here, we have no backend, so we'll just show the view.

Our signup process will depend on a `wizardapp.controllers` module that houses our `WizardSignupController`.

```
angular.module('wizardApp', [ 'ui.router', 'wizardapp.controllers' ]); ``
```

Our `WizardSignupController` will simply house the `$scope.user` object that we'll carry with us through the signup process as well as the signup action.

```
angular.module('wizardapp.controllers', []) .controller('WizardSignupCtrl', ['$scope', '$state', function($scope, $state) { $scope.user = {}; $scope.signup = function() {} }]); ``
```

Now, the the wizard process logic will house the majority of the work. We'll set this up in our `config()` function of our app:

```
angular.module('wizardApp', [ 'ui.router', 'wizardapp.controllers' ]) .config([ '$stateProvider', '$urlRouterProvider', function($stateProvider, $urlRouterProvider) { $stateProvider .state('start', { url: '/step1', templateUrl: 'partials/wizard/step1.html' }) .state('email', { url: '/step2', templateUrl: 'partials/wizard/step2.html' }) .state('finish', { url: '/finish', templateUrl: 'partials/wizard/step_3.html' }); }]); ``
```

With these options set up, we have our basic flow all done. Now, if the user navigates to the route `/step_1`, they'll be navigated to the beginning of the flow. Although it might make sense that our entire flow takes place at the root url (i.e. `/step_1`), we might want to house that in a sublocation (`/wizard/step_1`, for instance).

To do this, we'll set up an abstract state that will house the rest of our steps.

```
.config([ '$stateProvider', '$urlRouterProvider', function($stateProvider, $urlRouterProvider) { $stateProvider .state('wizard', { abstract: true, url: '/wizard', template: '<div><div ui-view></div></div>' }) .state('wizard.start', { url: '/step1', templateUrl: 'partials/wizard/step1.html' }) .state('wizard.email', { url: '/step2', templateUrl: 'partials/wizard/step2.html' }) .state('wizard.finish', { url: '/finish', templateUrl: 'partials/wizard/step_3.html' }); }]); ``
```

Now, instead of having our routes at a top level, we can nest them safely inside our `/wizard` url.

We'll also want to attach an action that happens at the end of the signup process that calls our `signup` function on our parent controller `WizardSignupController`. We'll set a controller on the final step of the wizard process that will simply call the function on the `$scope`. Because our entire wizard is encapsulated in our `WizardSignupController`, we'll be able to use the nested scope property of scopes like normal.

```
.state('wizard.finish', { url: '/finish', templateUrl: 'partials/wizard/step_3.html', controller: function($scope) { $scope.signup(); } }); ``
```

ui-utils

The ui-utils library is a powerful utility package that makes a lot of custom extensions available to your project without needing to [reinvent the wheel](#)¹⁰⁷.

We'll cover a few notable features that the ui-utils library gives us:

Installation

```
1 $ bower install --save angular-ui-utils
```

We'll need to ensure that we include the library in our html template. As each component of the ui-utils library is built as an individual module, we'll need to include each one independently.

mask

When we want to take a credit-card or a phone number (or any other input that requires a specific format), we can present a clean UI that tells our users that they are giving us clean input.

We'll need to ensure we include the `mask.js` library in our html:

```
1 <script type="text/javascript" src="app/bower_components/angular-ui-utils/modules\  
2 /mask/mask.js"></script>
```

And set the `ui-mask` as a dependency for our app:

```
angular.module('myApp', ['ui.mask']) ``
```

Now, we can specify input masks using the `ui-mask` directive. The `ui-mask` directive takes a single format string that follows the formatting rules:

- A - any letter
- 9 - any number
- * - any alphanumeric character.

For instance, to format a credit-card number in an input, we might set the `ui-mask` directive to look like:

¹⁰⁷http://en.wikipedia.org/wiki/Reinventing_the_wheel

```

1 <input name="ccnum" ui-mask="9999 9999 9999 9999" ng-model="user.cc" placeholder=\n
2 "Credit card number" />

```

Note that similar to how Angular input works as the input will not be valid until it matches the mask.

Note that this input only supports credit cards whose input mask matches 9999-9999-9999-9999. With a bit more work, we can support other card types.

In the same sense, we can format an input field with characters or any alphanumeric character.

ui-event

Just like the other modules, we'll need to include the event.js library in our html:

```

1 <script type="text/javascript" src="app/bower_components/angular-ui-utils/modules\
2 /event/event.js"></script>

```

And we need to include the ui.event as a dependency for our app as well:

```
angular.module('myApp', ['ui.event']) ``
```

When we want to handle events that are not natively supported by AngularJS. For instance, if we want the user to double click on a element or handle a blur event, we'd have to write a wrapper around the native browser events.

This ui-event module is simply a wrapper around native events, so we can use it to respond to any event that is fired by the browser on any element.

For instance, let's say we want to reveal an image after the user double clicks on another image. We simply set the ui-event directive to a key-value pair with the event name and the action to take when the element catches the event.

For instance, in our HTML, we can set a double click dblclick event to call a showImage() function on our controller:

```

1 

```

And in our controller, we can write our the method on the scope like normal:

```
.controller('DemoCtrl', ['$scope', function($scope) { $scope.showImage = function() { $scope.shouldshowImage
= !$scope.shouldshowImage; } }]); ``
```

Since the `ui-event` directive is simply a wrapper around native browser events, we can use it to mimic any browser event on any element.

For instance, if we want to capture a blur or focus event on an element, we can use the `ui-event` directive to capture these events for us.

For instance, let's say we want to provide some helpful tips for filling out a form. We can set actions on the `focus` event and the `blur` event to show and reveal these help tips.

For instance, if we have a form that includes the input fields of name and email, we can attach functions to our `blur` and `focus` events to show help events.

```
1 <form name="form">
2   <input type="text" name="name" placeholder="Your name"
3     ui-event="{focus: 'showNameHelp=true',
4               blur: 'showNameHelp=false'}"
5   />
6   <input type="email" name="email" placeholder="Your email"
7     ui-event="{focus: 'showEmailHelp=true',
8               blur: 'showEmailHelp=false'}"
9   />
10  </form>
```

With these events set on the input fields, we can show our help fields depending which field the user is focused on in the angular way (using `ng-show` and `ng-hide`).

ui-format

We'll need to ensure we include the `format.js` library in our html:

```
1 <script type="text/javascript" src="app/bower_components/angular-ui-utils/modules\
2 /format/format.js"></script>
```

And set `ui.format` as a dependency for our app:

```
angular.module('myApp', ['ui.format'])"
```

The `format` library is a wrapper around different ways to handle working with string tokens. It enables us to work directly with tokens we expect in our app that are variable.

Token replacement with the `format` library works either with an array or a key-value/javascript object. For instance:

```
1 {{ "Hello $0" | format: 'Ari' }}
```

Alternatively, we can bind the name to a variable on our scope and use the `format` library to present it in a clean format. Let's say we have a controller like so:

```
angular.module('myApp', ['ui.format']).controller('FormatCtrl', ['$scope', function($scope) { $scope.name = 'Ari'; }]); ``
```

Then we can format the input against that bound variable on the `$scope`:

Although this isn't particularly interesting (and this functionality comes out-of-the-box with Angular), it becomes particularly interesting when we want to manipulate text on a key-value basis.

For instance, we can format a string based on the keys of an object we have. Let's say we have an object with a `name` and an `email` attribute:

```
.controller('FormatCtrl', ['$scope', function($scope) { $scope.person = { name: 'Ari', email: 'ari@fullstack.io' } }]); ``
```

Then we can change our markup to include the keys as tokens to replace our markup to include `name` and `email` as tokens:

```
1 {{ "Hello :name. Your email is :email" | format: person }}
```

The `format` module is particularly useful when working with translations or `i18n` support (for more on translations, check out our [translations chapter](#) on `i18n` support).

Mobile apps

Mobile apps are **not** the *next frontier* for software developers, they're already here. There are already 1.2 billion mobile web app users and that number is growing rapidly ([Wikipedia¹⁰⁸](#)). Soon, the number of mobile devices will exceed the number of people on the planet. At the rate at which the number of mobile devices is growing, it's estimated that 5.1 billion people will be using mobile phones by 2017.

For us as app developers, it's important that we develop for mobile technology if we want to stay relevant. With AngularJS, we have some great support for mobile, written by both the Angular team and the community.

In this section, we're going to work through these two different ways to give our users a mobile experience for our app:

- Responsive web apps
- Native with Cordova

Responsive web apps

The easiest way to support mobile with Angular is by using the tools we already know and love to create a mobile-ready angular app: HTML and CSS. Since angular apps are already based on HTML, making our designs and interaction responsive are only a matter of building the architecture to support the different devices.

Interaction

For the desktop, the ability to create interactions are already available to us through the `ng-click` and family directives.

Starting from Angular version 1.2.0 and on, we now have the ability to use touchevents using the new `ngTouch` module. Since `ngTouch` is not built-in to the core Angular library, we'll need to install the library.

¹⁰⁸http://en.wikipedia.org/wiki/List_of_countries_by_number_of_mobile_phones_in_use

Installation

Installing ngTouch can be done in several ways. The simplest way to install the ngTouch module is by downloading the source from angularjs.org¹⁰⁹.

Find the `extras` in the download section and we'll download and store the `ng-touch.js` file into an accessible location in our app.

Alternatively, we can use `bower` to install `angular-touch`:

```
1 $ bower install angular-touch --save
```

Either way, we'll need to reference the library in our `index.html` as a script:

```
1 <script src="/bower_components/angular-touch/angular-touch.js"></script>
```

And finally, include ngTouch as a dependency in our app:

```
1 angular.module('myApp', ['ngTouch']);
```

Now we're ready to take advantage of the ngTouch library.

ngTouch

Mobile browsers work slightly differently than desktop browsers when dealing with click events. Mobile browsers detect a *tap* event and then wait about 300 ms or so to detect any other taps, for instance if we're double-tapping the device. After this delay, then the browser fires a click event.

This delay can make our apps feel incredibly unresponsive. Instead of dealing with the `click` event, we can detect `touch` events instead.

The ngTouch library handles this for us, seamlessly through the `ng-click` directive and will take care of calling the *right* click event for us. This so-called fast click event will be called.

After the *fast* click has been called, the browser's delayed click will then be called, causing a 'double' action. ngTouch takes care of this for us.

Using the `ngClick` directive on mobile devices works the exact same way on mobile browsers as it does on desktop browsers:

¹⁰⁹<http://angularjs.org/>

```
1 <button ng-click="save()">Save</button>
```

ngTouch also introduces two new directives: *swipe* directives. These swipe directives allow us to capture user swipes, either left or right across the screen. These are useful for situations where we want the user to be able to swipe a photo gallery photo or to a new portion of our app.

The `ngSwipeLeft` directive detects when an element is swiped from the right to the left, while the `ngSwipeRight` directive detects when an element is swiped from the left to the right.

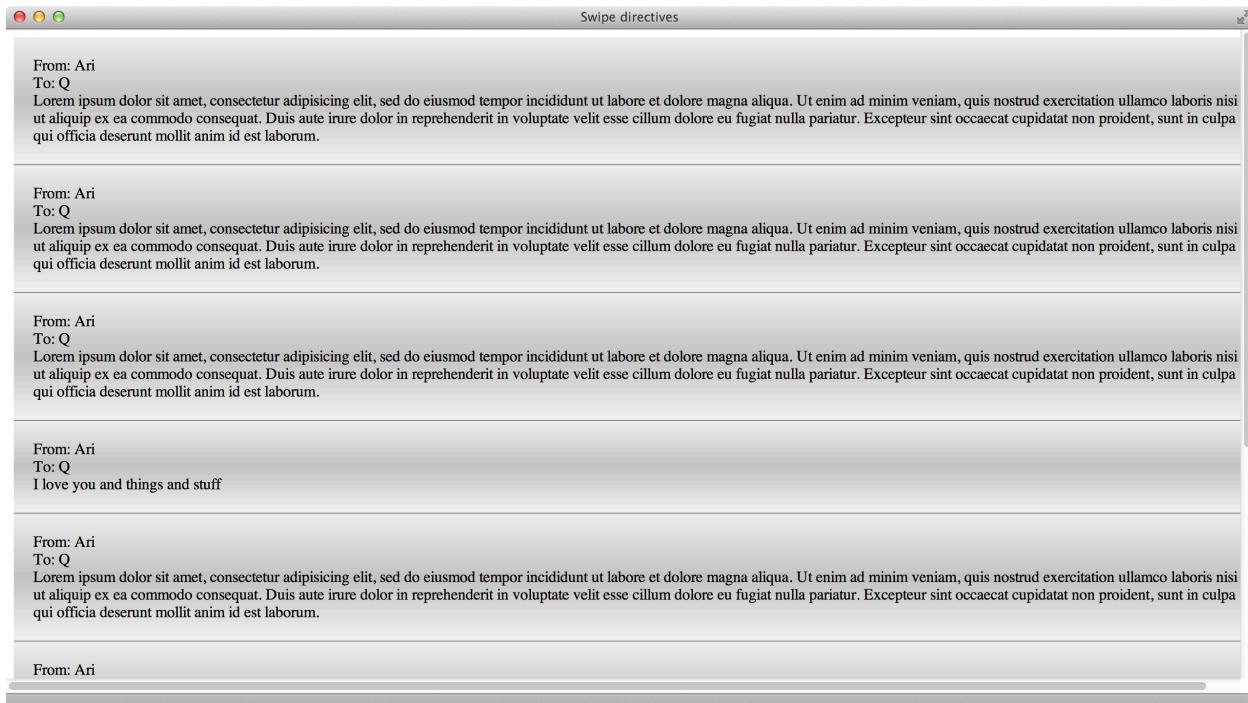
One of the nice features the `ngSwipe*` directives give us is that they work both with touch-based devices as well as mouse clicking and dragging.

Using the `ngSwipe*` directives is easy. For instance, let's say we have a list of emails and we want to reveal actions for each email, like the popular mobile email client MailboxApp.

We can easily implement it using these swipe directives on our list of elements. When we are showing our list of emails, we'll enable swiping in one direction to show the actions we can take on that particular mail item.

When we are showing actions for the mail item, we'll enable swiping in the other direction to hide the actions we can take.

```
1 <ul>
2   <li ng-repeat="mail in emails">
3     <div
4       ng-show="!mail.showActions"
5       ng-swipe-left="mail.showActions=true">
6       <div class="from">
7         From: <span>{{ mail.from }}</span>
8       </div>
9       <div class="body">
10      {{ mail.body }}
11    </div>
12  </div>
13  <div
14    ng-show="mail.showActions"
15    ng-swipe-right="mail.showActions=false">
16    <ul class="actions">
17      <li><button>Archive</button></li>
18      <li><button>Trash</button></li>
19    </ul>
20  </div>
21  </li>
22 </ul>
```



Swipe directives example

\$swipe service

For more *custom* touch-based animations, we can use the `$swipe` service directly. The `$swipe` service is a service that abstracts the details of hold-and-drag swiping behavior.

The `$swipe` service has a single method called `bind()`. This `bind()` method takes an element that it will bind the swipe actions on as well as an object with four event handlers.

These event handlers get called with an object that contains the coordinates object, like so: `{ x: 200, y: 300 }`.

The four events handlers are handlers to handle the following events:

- start

The `start` event is fired on either a `mousedown` or a `touchstart` event. After this event, the `$swipe` service sets up watches for `touchmove` and `mousemove` events. These events are only fired until the **total distance moved** exceeds a small distance to prevent accidental swipes.

Once this distance has been passed, then one of two events happen:

- If the vertical delta is greater than the horizontal, then the browser takes over as a scroll event.
- If the horizontal delta is greater than the vertical delta, then this is viewed as a swipe and our `move` and `end` events are set to follow the swipe.

- move

The move event is called on mousemove and touchmove events only after the \$swipe service has determined that a swipe is in fact in progress.

- end

The end event is fired when we've finished with a touchend or a mouseup event after the move event has been fired.

- cancel

The cancel event is called on either a touchcancel or we begin scrolling after the start event instead.

For instance, we can create a directive that enables swiping between slides that might control a projector screen. To handle swiping on the mobile control, we'll use the \$swipe service to handle our custom logic for how to display the UI layer.

```
1 angular.module('myApp')
2 .directive('mySlideController', ['$swipe',
3   function($swipe) {
4
5     return {
6       restrict: 'EA',
7       link: function(scope, ele, attrs, ctrl) {
8         var startX, pointX;
9
10        $swipe.bind(ele, {
11          'start': function(coords) {
12            startX = coords.x;
13            pointX = coords.y;
14          },
15          'move': function(coords) {
16            var delta = coords.x - pointX;
17            // ...
18          },
19          'end': function(coords) {
20            // ...
21          },
22          'cancel': function(coords) {
23            // ...
24          }
25        });
26      }
27    };
28  }
29
```

```
24          }
25      });
26  }
27 }
28 }]);
```

angular-gestures and multi-touch gestures

Angular gestures is an angular module that gives us the ability to handle multi-touch in our angular apps. It is based on the very popular and well-tested [Hammer.js¹¹⁰](#) library.

The Hammer.js library gives us a bunch of events common to touchscreen events:

- Tap
- DoubleTap
- Swipe
- Drag
- Pinch
- Rotate

The angular-gestures library enables us to use these events using angular directives. For instance, all of these directives are available to us:

- hmDoubleTap : ‘doubletap’,
- hmDragStart : ‘dragstart’,
- hmDrag : ‘drag’,
- hmDragUp : ‘dragup’,
- hmDragDown : ‘dragdown’,
- hmDragLeft : ‘dragleft’,
- hmDragRight : ‘dragright’,
- hmDragEnd : ‘dragend’,
- hmHold : ‘hold’,
- hmPinch : ‘pinch’,
- hmPinchIn : ‘pinchin’,
- hmPinchOut : ‘pinchout’,
- hmRelease : ‘release’,
- hmRotate : ‘rotate’,
- hmSwipe : ‘swipe’,
- hmSwipeUp : ‘swipeup’,

¹¹⁰<http://eightmedia.github.io/hammer.js/>

- hmSwipeDown : 'swipedown',
- hmSwipeLeft : 'swipeleft',
- hmSwipeRight : 'swiperight',
- hmTap : 'tap',
- hmTouch : 'touch',
- hmTransformStart : 'transformstart',
- hmTransform : 'transform',
- hmTransformEnd : 'transformend'

angular-gestures installation

To install the `angular-gestures` library in our app, we'll need to include the `gestures.js` (or `gestures.min.js`) library in our page.

We can either download the `gestures.js` files directly from the github page at <https://github.com/wzr1337/angular-gestures>¹¹¹ or we can use bower to install it.

To install `angular-gestures` using bower, install it with the following command:

```
1 $ bower install --save angular-gestures
```

Lastly, we'll need to set `angular-gestures` as a dependency for our angular app:

```
1 angular.module('myApp', ['angular-gestures']);
```

Using angular-gestures

From here, angular gestures are really easy to use. Gestures are just angular directives, so we'll use them the same way we use any other directives in our app.

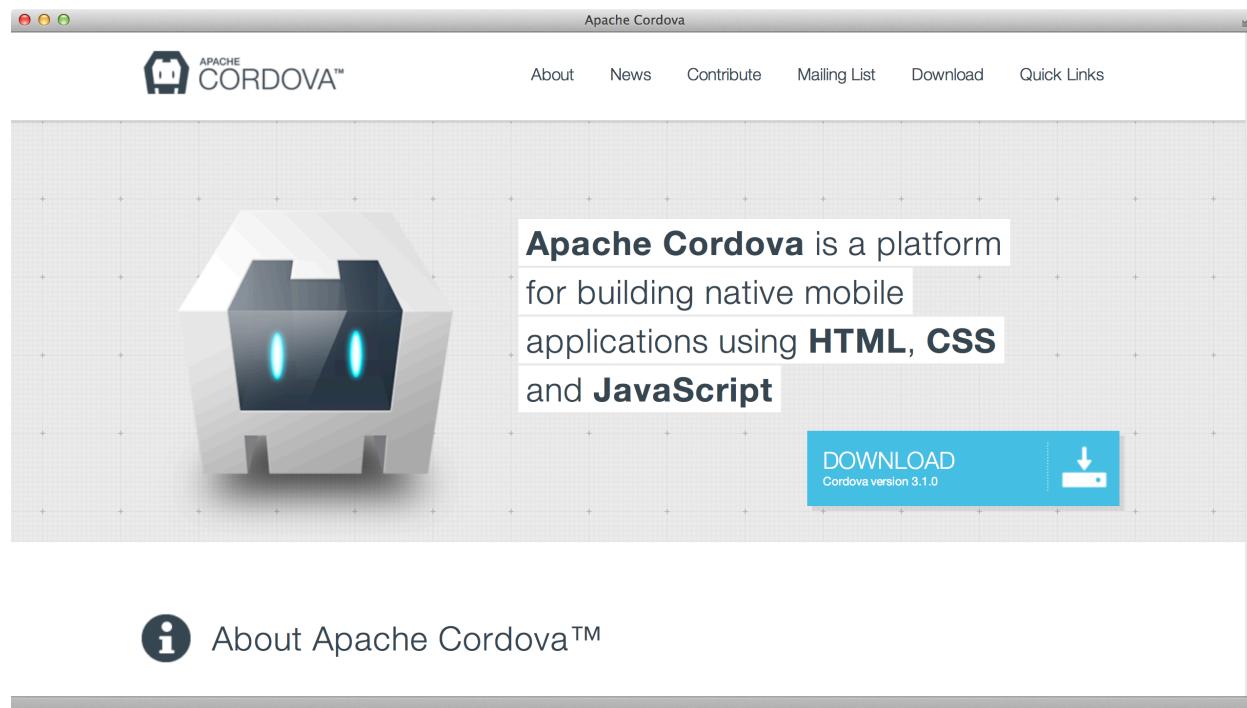
Let's say that we want to allow users to rotate, pinch, and zoom photos in a photo gallery. We can use

FINISH::::

¹¹¹<https://github.com/wzr1337/angular-gestures>

Native applications with Cordova

Cordova is a free, open-source framework that allows us to create mobile apps using standard web APIs, instead of native code. It enables us to write mobile applications using HTML, JavaScript, CSS, and AngularJS instead of needing to write Objective-C or Java (for iOS or Android, respectively).



About Apache Cordova™

Cordova

Cordova exposes native device access through JavaScript APIs that allow us to run device-specific operations, such as getting the native location or using the camera. It is built to support the plugin architecture so we can take advantage of Cordova community-built plugins, such as native audio access or barcode scanning plugins.

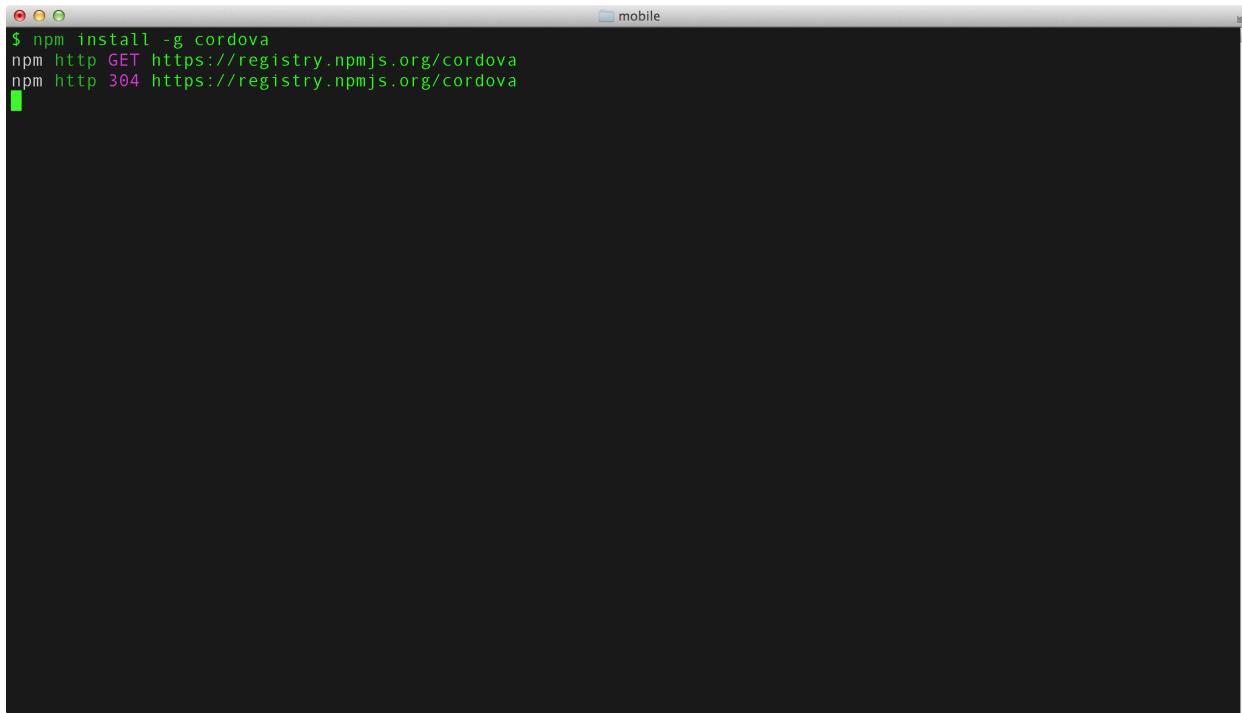
One of the benefits of using Cordova is that we can reuse our Angular app code to support the mobile environment. Of course, there are a few issues that we'll deal with, such as performance and native component access.

Installation

Cordova itself is distributed as an npm package, so we'll use npm to install it.

If you do not have npm installed, make sure you have node installed. For information on installing NodeJS, read the [Next Steps](#) chapter.

```
1 $ npm install -g cordova
```



```
$ npm install -g cordova
npm http GET https://registry.npmjs.org/cordova
npm http 304 https://registry.npmjs.org/cordova
```

Installing Cordova

The Cordova package includes a generator that will create our app and make it Cordova-ready.

Getting started with Cordova

Getting started with Cordova is simple. We'll use the generator to create the starting point of our Cordova app. We'll call the app `GapApp`.

The generator takes up to three parameters:

- project-directory (required)

The directory where we'll create the app

- package-id

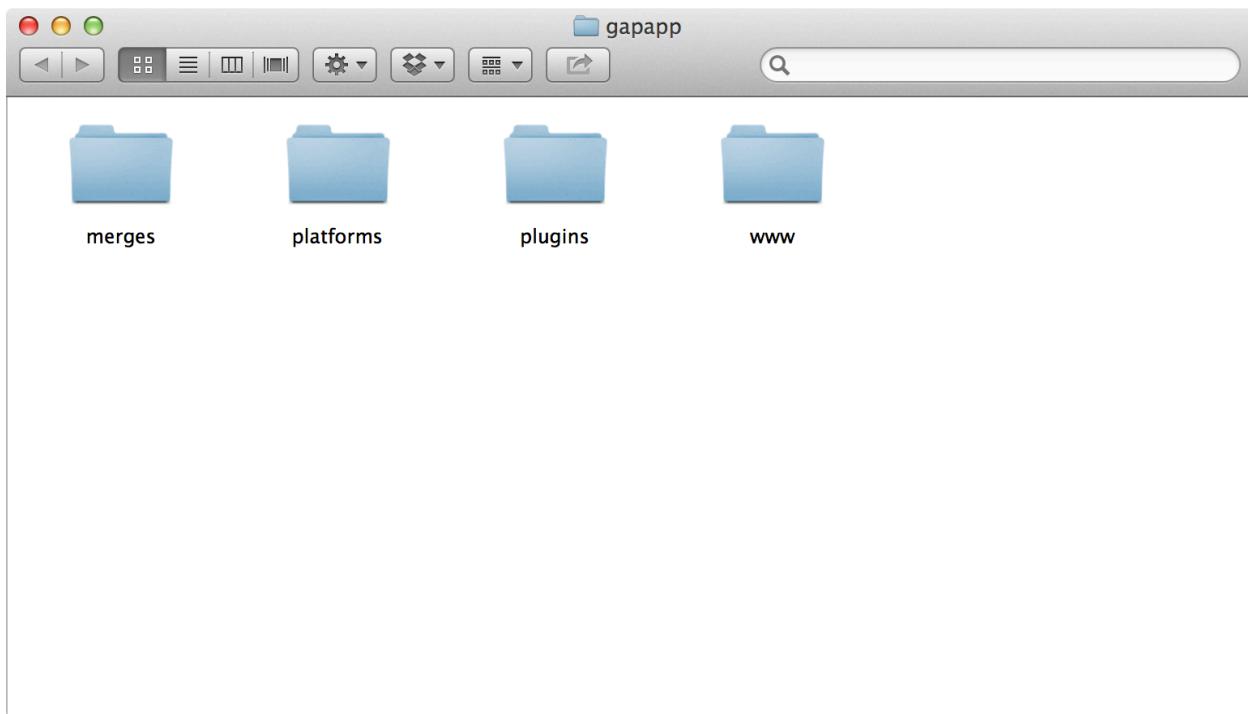
The ID of the project (the package name in reverse-domain style)

- name

The package name (name of the application)

```
1 $ cordova create gapapp io.fullstack.gapapp "GapApp"
```

This line will set up a directory called `gapapp` (identified by the first parameter) with a package ID `io.fullstack.gapapp` and the project name `GapApp`.



Cordova file structure

The Cordova team has broken Cordova into plugins so that we don't need to include platforms we won't be building for (and thus making it easier to develop support for other platforms). That means that we'll need to add to our project any platforms for which we're interested in developing.

For this project, we'll assume the rest of these commands are run from *inside* the project directory:

```
1 $ cd gapapp/
```

We'll be building for iOS (although the process is the same for other platforms). To add iOS as a platform, simply add it to the project using the following Cordova command:

```
1 $ cordova platform add ios
```

For this command to work, we'll need to ensure we have the iOS SDK installed using XCode. Download the iOS SDK and XCode at developer.apple.com¹¹².

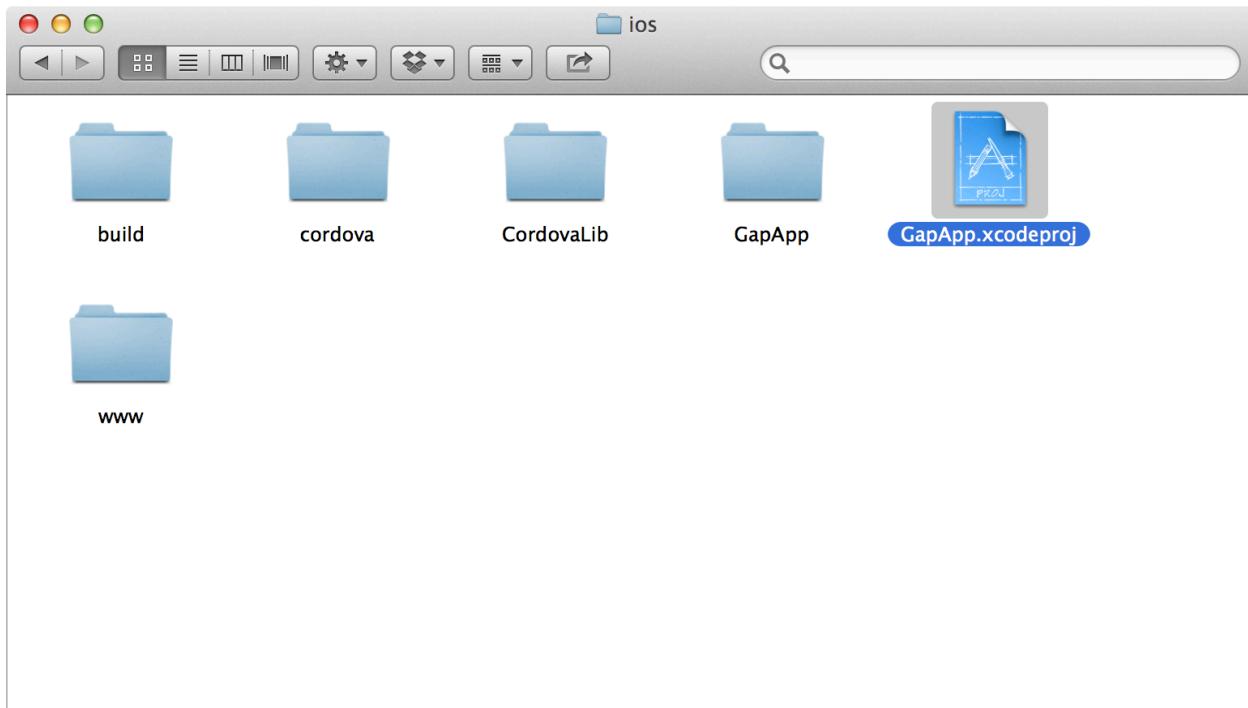
Once you have that set, build the basic app:

¹¹²<https://developer.apple.com/>

```
1 $ cordova build ios
```

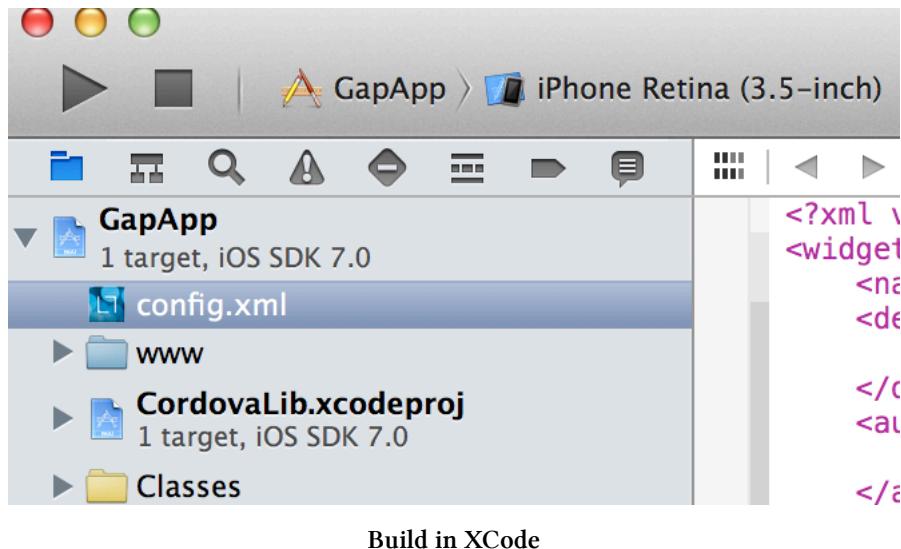
Now, due to some intricacies with Apple's developer tools, we will have to build the app ourselves to get it to run on our local iOS simulator.

Let's navigate to our app directory, where we'll find the platforms directory. Inside of it, we'll find the `ios/` directory that was created for us by the `platform add` command above.



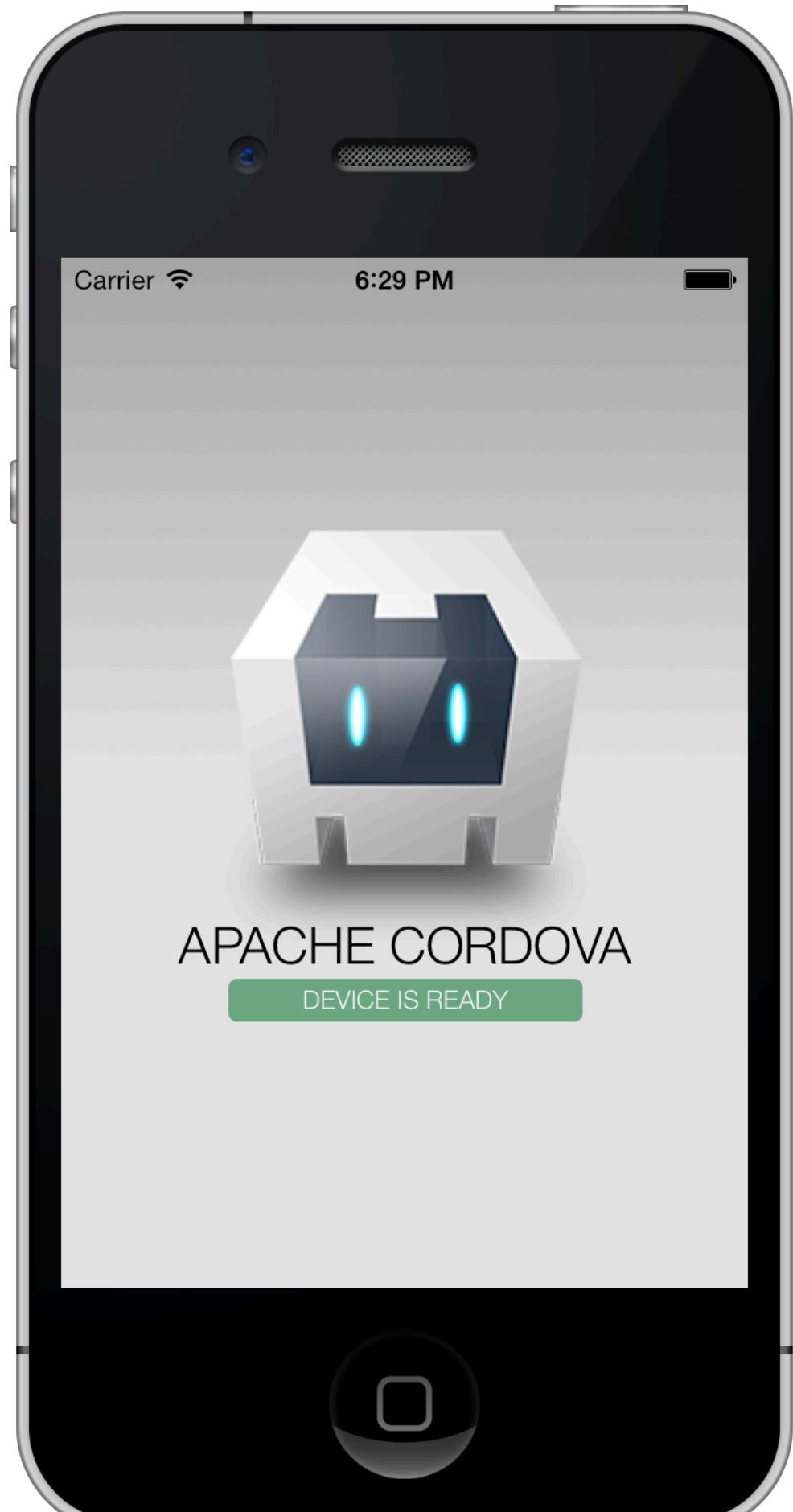
Generated project

In XCode, open the project that we created with said command. Make sure the simulator is shown in the platform identifier at the top of XCode.



Click run.

Once you have done so, we should see the basic Cordova app start to run in our simulator.



Development workflow with Cordova

Cordova powers the PhoneGap project, which has been accepted into the Apache Foundation. The project itself includes a command-line tool that we'll use to interact with our native app, from creation to deployment.

Platforms

At this point, we've created our app and added a platform (in this case, iOS).

Available platforms for the Cordova app vary depending on which development environment we're using. On a Mac, the available platforms are:

- iOS
- Android
- Blackberry10
- Firefox OS

For a Windows machine, we can develop for the following platforms:

- Android
- Windows Phone 7
- Windows Phone 8
- Windows8
- Blackberry10
- Firefox OS

If we forget which platforms are available, we can run the `platforms` command to check which are available and installed:

```
1 $ cordova platforms ls
```

To add a platform, we can use the `platform add` command (as we've done above):

```
1 $ cordova platform add android
```

To remove one, we can use the `rm` or `remove` command:

```
1 $ cordova platform rm blackberry10
```

Plugins

Cordova is built to be incredibly modular, with the expectation that we will install all of the non-core components with the plugin system. To add a plugin to our project, we'll use the `plugin add` command:

```
1 $ cordova plugin add https://git-wip-us.apache.org/repos/asf/cordova-plugin-geolocation.git
```

We can list the current plugins that we have installed using the `plugins ls` command:

```
1 $ cordova plugins ls
2 [ 'org.apache.cordova.geolocation' ]
```

Finally, we can remove a plugin using the `plugin rm` command:

```
1 $ cordova plugins rm org.apache.cordova.geolocation
```

Building

By default, Cordova creates a skeleton project that houses the web files in `www/` directory in the project directory. When Cordova builds the project, it copies these files and places them in their platform-specific directories.

To build the app, we'll use another Cordova command, the `build` command:

```
1 $ cordova build
```

Without specifying any platform to build for, this command will build for all of the platforms we've listed in our project.

We can limit the scope by building only for specific platforms, such as:

```
1 $ cordova build ios
2 $ cordova build android
```

The `build` command will ensure that the necessary platform-specific code is set so our app can be compiled. In effect, we're doing the same thing as calling `cordova prepare` & `cordova compile`.

Emulating and running

Cordova also makes it possible to run an emulator in order to simulate running the app on a device. Doing so is, of course, only possible if an emulator is installed and set up on our local development environment.

Assuming our emulator is set up in our development environment, we can tell Cordova to launch and install our app in our emulator:

```
1 $ cordova emulate ios
```

For iOS, we may have to build the project (as we did above) using XCode if the emulator environment is not set up on our machine.

It's also possible to run the application on a particular device by using the `run` command instead. The `run` command will launch the application on a device or on the emulator if no device is found and available.

```
1 $ cordova run ios
```

In development

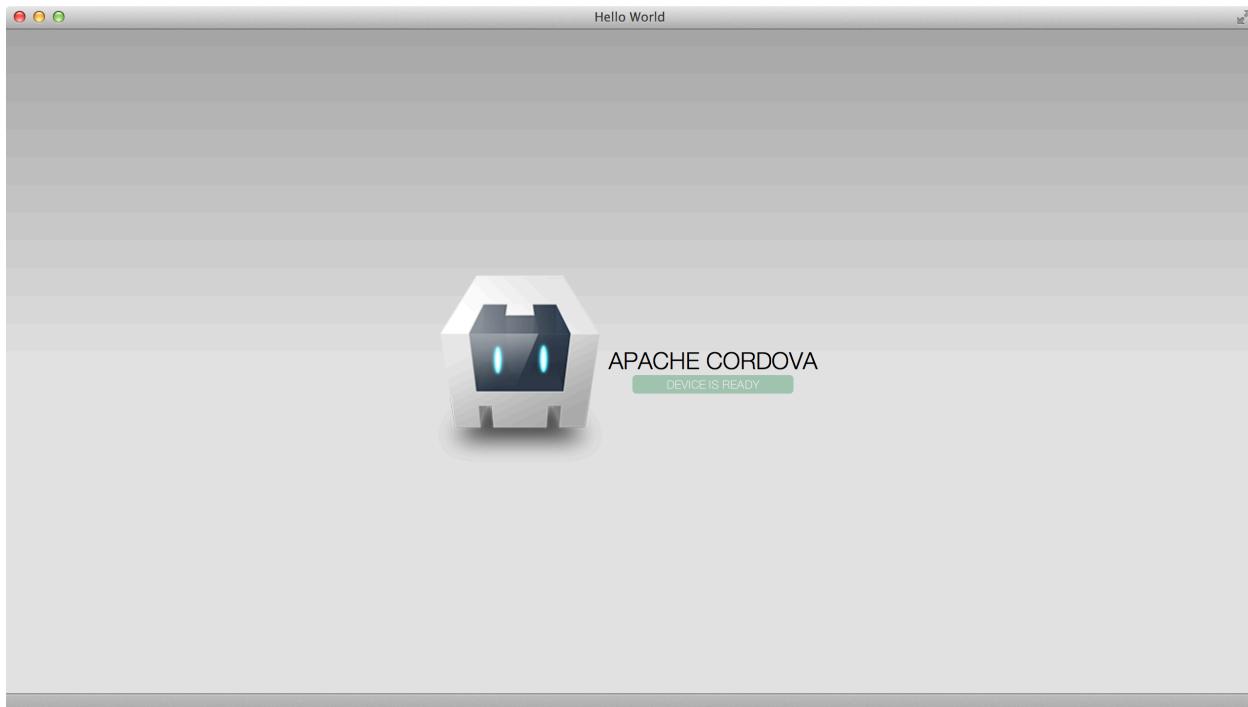
It can be cumbersome to make a change to one part of our app and need to recompile the app to see the changes reflected in our app. To help speed the process of developing the web app side of the app, we can use the `serve` command to serve a local version of our `www/` folder to a web browser.

```
1 $ cordova serve ios
2 Static file server running at
3   => http://0.0.0.0:8000/
4 CTRL + C to shutdown
```

Now, we can use our web browser and navigate to the URL: `http://localhost:8000/ios/www/index.html`. Our app's `www/` folder is being served through HTTP, so we can build it and watch it change as we make changes to the app.

When we make changes, we'll need to make sure we rebuild the app:

```
1 $ cordova build ios
```



Building using Safari

Angular Cordova Service

When our Cordova app is ready, the device has connected, and everything is ready to go, Cordova will fire the browser event called `deviceready`.

With Angular, we can either bootstrap the app *after* this event has been fired or we can use promises to handle our logic after the `deviceready` event has been fired.

To bootstrap the app after we've received the `deviceready` event, we'll need to set an event listener for the event and then manually call `bootstrap` on our app:

```
1 angular.module('myApp', []);
2
3 var onDeviceReady = function() {
4     angular.bootstrap( document, [ 'myApp' ]);
5 }
6 document.addEventListener('deviceready',
7 onDeviceReady);
```

We prefer to use an alternative method of listening for the `deviceready` event that uses promises to set up execution bindings for after the `deviceready` event has been fired.

We'll set up an Angular module that will listen for the deviceready event. We'll use a service that will listen for the deviceready event and resolve our promises depending on whether the event has been fired.

```
1 angular.module('fsCordova', [])
2 .service('CordovaService', ['$document', '$q',
3   function($document, $q) {
4
5     var d = $q.defer(),
6         resolved = false;
7
8     var self = this;
9     this.ready = d.promise;
10
11    document.addEventListener('deviceready', function() {
12      resolved = true;
13      d.resolve(window.cordova);
14    });
15
16    // Check to make sure we didn't miss the
17    // event (just in case)
18    setTimeout(function() {
19      if (!resolved) {
20        if (window.cordova) d.resolve(window.cordova);
21      }
22    }, 3000);
23  }]);
24 ]);
```

Now, we'll set the fsCordova as a dependency for our module:

```
1 angular.module('myApp', ['fsCordova'])
2 // ...
```

We can use the CordovaService to determine if Cordova is, in fact, ready, and we can set our logic to depend upon the service being ready:

```
1 angular.module('myApp', ['fsCordova'])
2 .controller('MyController',
3   function($scope, CordovaService) {
4     CordovaService.ready.then(function() {
5       // Cordova is ready
6     });
7   });

```

Including Angular

With the bare Cordova app, we only have a bare JavaScript app that hides and displays the JavaScript view in `js/index.js`.

We can introduce Angular into the workflow very simply. As we are building a native app, including Angular from a CDN is not ideal; instead, we'll include the necessary components directly into the app.

We can use [Bower](#) for more complex setups, but for the time being, we'll keep it simple.

To get our Angular app building, we'll need to download Angular from angularjs.org¹¹³ and store it in a directory accessible by our `index.html`. We recommend `www/js/vendor/angular.js`.

Once that's set, we can start building our Angular app. We'll need to include the JavaScript file in our `www/index.html`.

```
1 <script type="text/javascript" src="js/vendor/angular.js"></script>
```

Now, we can replace all of the contents of the `js/index.js` file with our Angular app and develop our app like normal.

Development workflow

When building our app, we'll use the following workflow:

- Start our local server (Cordova serve [platform])
- Edit our app
- Rebuild our app (Cordova build [platform])

This flow, although somewhat cumbersome, is how we'll edit our app.

If our app doesn't rely on the Cordova platform, we can edit outside of the simulator and in our web browser (e.g., Chrome). In this case, we can work specifically with building our app, instead of needing to rebuild and redeploy the app.

¹¹³<http://angularjs.org>

Building with Yeoman

We can use [Yeoman¹¹⁴](#) to build a production-ready version of our app. Yeoman is a collection of build scripts that is the officially supported build process for Angular apps. For more information on Yeoman, check out the Yeoman [section](#) in the [Next Steps](#) chapter.

To install Yeoman, the Angular generator, and the Cordova generator:

```
1 $ npm install -g yo  
2 $ npm install -g generator-angular  
3 $ npm install -g cordova
```

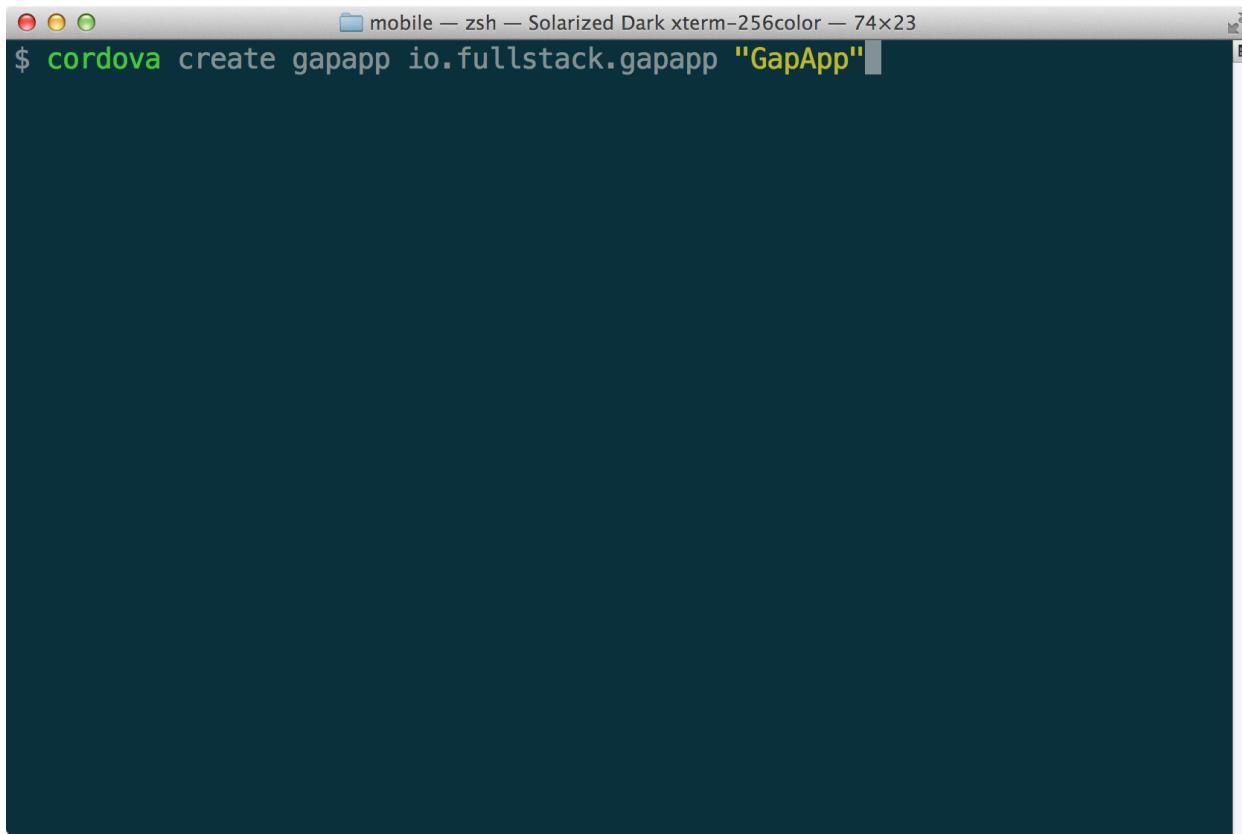
In order to get Yeoman working with Cordova, we'll need to make a few adjustments to the flow we describe above.

We'll first create a cordova app like normal.

```
1 $ cordova create gapapp io.fullstack.gapapp "GapApp"
```

This will create the folder `gapapp/` in our local directory like normal.

¹¹⁴<http://yeoman.io>

A screenshot of a Mac OS X terminal window titled "mobile — zsh — Solarized Dark xterm-256color — 74x23". The window shows a single line of text: "\$ cordova create gapapp io.fullstack.gapapp \"GapApp\"". The terminal has its characteristic dark blue background with light blue text and red cursor.

Generating our app

Next, let's change into the directory and add our platform:

```
1 $ cd gapapp/  
2 $ cordova platform add ios
```

This will create a platform folder that we'll work with in a minute to get the actual gapapp working locally in both our emulator and our device.

The first thing that we need to do is set up our yeoman app. in our directory and make a few minor changes to the default configuration:

```
1 $ yo angular
```

We'll go through the normal yeoman questions and let the process complete and build us our directory like normal.

When this process is done, we will have our yeoman app in the `app/` directory. This where we'll do all of our work when it comes to building our mobile app.

In our toolchain, we'll use the following workflow to build our app:

- Write code
- Test our code (angular testing)
- Run our code in the emulator (optional)
- Test our code on a running device (optional)

The first two tasks are already taken care of using the fantastic yeoman build tool. The second two are the tasks that we'll build now.

Cordova works by including the `www/` directory into the compiled apps, so any modifications we make to the files in the `www/` directory will be wrapped into the compiled application after we build it.

Modifying yeoman to work with cordova

Yeoman assumes a different structure by default where it will build our application into a folder called `dist/`. We'll modify this build directory to build into the `www/` directory.

First, we must save the `www/config.xml` file that comes built by the cordova create command. Copy or move this file from the `www/` into the `app/` directory:

```
1 $ cp www/config.xml app/
```

Now we'll want this `config.xml` to be copied back over to our `www/` directory when yeoman builds it.

To change the default directory in which yeoman builds our app, we'll find the yeoman section in our `Gruntfile.js` and change the `dist:` key from `dist` to be `www`:

```
1 // ...
2 grunt.initConfig({
3   yeoman: {
4     app: require('./bower.json').appPath || 'app',
5     dist: 'www' // <~ Change this to www
6   },
7   watch: {
8     // ...
9   }
10 }
11 module.exports = grunt;
```

Next, we'll need to tell yeoman to copy over the `config.xml` file along with the rest of the files that get copied over by yeoman. Luckily this process is incredibly easy as we'll only need to add a single string to the list of paths in the `copy` task.

Inside the `copy:dist` configuration, add the extension `.xml` to the glob string that lists the files to copy over:

```
1 // ...
2 },
3 copy: {
4   dist: {
5     files: [
6       expand: true,
7       dot: true,
8       cwd: '<%= yeoman.app %>',
9       dest: '<%= yeoman.dist %>',
10      src: [
11        '*.{ico,png,txt,xml}', // ~ Add xml
12        '.htaccess',
13        // ...
14      ]
15    }
16  }
17}
```

With these two commands in place, we can now build our app with yeoman and our application inside the app/ directory will be built into the www/ directory.

```
1 $ grunt build
```

This sets up our basic app to work with yeoman, but not with any of the dev tools to actually build our mobile app.

Fixing the yeoman build

Note that when we're developing our applications, we cannot fetch remote resources from a CDN, for instance. The default yeoman build sets up our scripts to load from the google CDN.

We must modify the `index.html` template slightly so that we don't load jquery and angular from the CDN by surrounding the script tags inside a build script, like so:

```
1 <!--  
2   Add the following line and the endbuild line  
3   after the script tags  
4 -->  
5 <!-- build:js scripts/library.js -->  
6 <script src="bower_components/jquery/jquery.js"></script>  
7 <script src="bower_components/angular/angular.js"></script>  
8 <!-- endbuild -->
```

Building the mobile part

To build our mobile app with our yeoman tools, we can add a few more task definitions to grunt to make the processes of building, testing, and deploying to our devices a breeze.

Cordova has two different binaries to build our app, a local builder inside the `platforms/` directory to actually compile the native application and a global binary to build the our application from the root directory created by the cordova create command.

To build our application *normally*, we can simply run the `cordova build` command from the root directory and cordova will handle copying the `www/` directory into the appropriate location for the different platforms.

```
1 $ cordova build
```

We can create a task to do the same for us so we can simply use grunt like normal. To support this, we'll need to install a grunt library called `grunt-shell`. We can use `npm` to handle installing this library for us:

```
1 $ npm install --save-dev grunt-shell
```

Next we'll need to define our configuration for the shell commands. We're going to create two commands, one to emulate our app on our computer's mobile device simulator and one to our actual device.

```
1 uglify: {
2   // ...
3 },
4 shell: {
5   build: {
6     command: 'cordova build'
7   },
8   emulate: {
9     command: 'cordova build'
10  }
11 }
12 // ...
```

These commands are very similar right now as in both commands we'll want to tell cordova to actually build the application first.

Next, we'll need to use the *local* `cordova` command to emulate or run our application. The *local* `cordova` command can be found in our location `platforms` path for each different type of platform.

For instance, we'll have a local `emulate` command in the `ios` directory as we've added the `ios` platform in `platforms/ios/cordova/emulate`. If we add an android platform, we'll find the `cordova` command inside the android platform directory at `platforms/android/cordova/emulate`.

We'll build a helper function to help us find these local `cordova` commands. At the top of the Gruntfile, add the following command:

```

1 module.exports = function (grunt) {
2     var path = require('path'),
3         cordova = require('cordova');
4
5     var cordova_cmd = function(cmd) {
6         var target = grunt.option('target') || "ios";
7         return path.join(
8             __dirname, "platforms",
9             target, "cordova", cmd);
10    }

```

With this, we can use the `cordova_cmd()` function to find these local `cordova` commands. With these local commands we can modify our shell tasks from above to include these custom tasks:

```

1 shell: {
2     build: {
3         command: 'cordova build && ' +
4             cordova_cmd('emulate')
5     },
6     run: {
7         command: 'cordova build &&' +
8             cordova_cmd("run")
9     }
10 }

```

We can use these commands simply by running them directly in the shell to test them out:

```
1 $ grunt shell:build
```



Depending upon the platform we're developing with, we may need to install dependencies. For instance, with `ios`, we'll need to make sure we have `ios-sim` installed. Check the [official cordova docs¹¹⁵](#) for information on dependencies for your platform.

¹¹⁵<http://docs.phonegap.com/en/3.1.0/index.html>

Now these commands are semi-useless without being wrapped into another command that *actually* builds the app from our `app/` directory into the `www/` directory.

Grunt makes this an easy process where we can simply register a new task that composes of multiple grunt tasks. In this case, we'll simply wrap our `build` and `run` commands into a new task that calls `build` first:

```
1      // task configuration above here
2
3  });
4 // ...
5 grunt.registerTask('devemulate', [
6   'build',
7   'shell:build',
8 ]);
9
10 grunt.registerTask('devrun', [
11   'build',
12   'shell:run'
13 ]);
14
15 grunt.registerTask('server', function (target) {
16   // ...
```

Now we have the command `devemulate` available for us to run our app in the simulator environment and on our device.

```
1 $ grunt devemulate
```



Note, if the command is not working as expected, using the `--verbose` flag with `grunt` will often reveal issues, such as missing dependencies.



We can also run the application on a mobile device that is set up to accept development applications by using the `devrun` task:

```
1 $ grunt devrun
```

Handling the navigator

Lastly, the cordova platform uses the `deviceready` event fired on the DOM to indicate that the device itself is ready for action.

Now we run into a timing issue to tell us if the cordova app is ready to go or if our angular app has loaded. We can get around this issue by creating a service that captures the `deviceready` event and resolves to a variable on the outset that we can work with.

The service is very simple:

```
1 angular.module('gapappApp.services')
2 .factory('Cordova', function($q) {
3     var d = $q.defer();
4
5     if (window.navigator) {
6         d.resolve(window.navigator);
7     } else {
8         document
9             .addEventListener('deviceready', function(evt) {
10                 d.resolve(navigator);
11             });
12     }
13
14     return {
15         navigator: function() {
16             return d.promise;
17         }
18     }
19 })
```

Now, when we want to use cordova's `navigator`, we'll just use the following syntax that will resolve once the device is ready:

```
1 angular.module('gapappApp')
2   .controller('MainCtrl',
3     function($scope, Cordova) {
4       Cordova.navigator().then(function(navigator) {
5         navigator.notification.vibrate();
6       });
7     });

```

Localization

As worldwide access to the web increases, we as developers are constantly pressed to make our apps internationally and locally accessible. When a user visits our apps, he or she should be able to switch languages on the fly at runtime.

Given that we are building AngularJS client-side apps, we don't particularly want the user to have to refresh the page or visit an entirely different URL. Of course, AngularJS could easily accommodate your international audience natively, perhaps by generating different templates for different languages and serving those within the app.

This process can become cumbersome, and what happens when you want to change the layout of the app? Every single template needs to be rebuilt and redeployed. This process should just be *easy*.

angular-translate

Instead of creating new templates, we'll use **angular-translate**, an AngularJS module that brings i18n (internationalization) to your Angular app. **angularjs-translate** requires us to create a JSON file that represents translation data per language. It lazy-loads the language-specific translation data from the server only when necessary.

The library **angular-translate** comes with built-in directives and filters that make the process of internationalizing apps simple. Let's get started.

Installation

To use **angular-translate**, we need to load the angular-translate library. We can install it in several different ways, but we prefer using Bower.

Bower is a front-end package manager. It handles not only JavaScript libraries, but also HTML, CSS, and image packages. A package is simply encapsulated, third-party code that is typically publicly accessible in a repository.

- Using Bower

We install angular-translate using the normal Bower process:

```
1 $ bower install angular-translate
```

Alternatively, we can download the minified version of angular-translate from github.

Once we've installed the latest stable version of angular-translate, we can simply embed it in our HTML document. Just make sure it's embedded after Angular itself, as it depends on the core angular library.

```
1 <script src="path/to/angular.js"></script>
2 <script src="path/to/angular-translate.js"></script>
```

Last but not least, our app has to declare angular-translate as a load dependency:

```
var app = angular.module('myApp', ['pascalprecht.translate']);“
```

Great, we're now ready to use angular-translate's components to translate our app!

Teaching your app a new language

Now our app depends upon angular-translate as installed, and our app declares it as a dependency, so we can use it to translate our app's contents.

First, we need to provide translation material for our app to actually *speak* a new language. This step actually entails configuring the `$translate` service through our fresh `$translateProvider` service.

Training our app to use a new language is simple. Using the `config` function on our app, we provide the different language translations for our app, i.e. English, German, Hebrew, etc.. First, we inject our `$translateProvider` in the `config` function, like so:

```
angular.module('angularTranslateApp', ['pascalprecht.translate']).config(['$translateProvider', function($translateProvider) { // Our translations will go in here }]);“
```

To add a language, we have to make `$translateProvider` aware of a **translation table**, which is a JSON object containing our messages (keys) that are to be translated into (values). Using a **translation table** enables us to write our translations as simple JSON for loading remotely or setting at compile-time, such as:

```
{ 'MESSAGE': 'Hello world', } “
```

In a translation table, the key represents a translation id, whereas the value represents the concrete translation for a certain language. Now add a translation table to your app. `$translateProvider` provides a method called `translations()`, which takes care of that.

```
app.config(function ['$translateProvider', ($translateProvider) { $translateProvider.translations({ HEADLINE: 'Hello there, This is my awesome app!', INTRO_TEXT: 'And it has i18n support!' }); }]);“
```

With this translation table in place, our app is set to use angular-translate. Since we're adding the translation table at configuration time, angular-translate's components are able to access it as soon as they are instantiated.

Let's switch over to our app template. Adding translations in the view layer is as simple as binding our *key* to the view. Using the `translate` filter, we don't even have to engage our controller or services or worry about the view layer: We're able to decouple the translate logic from any controller or service and make our view replaceable without touching business logic code.

Basically, the `translate` filter works like so:

```
1 <h2>{{ 'TRANSLATION_ID' | translate }}</h2>
```

To update our example app, we make use of the `translate` filter:

```
1 <h2>{{ 'HEADLINE' | translate }}</h2>
2 <p>{{ 'INTRO_TEXT' | translate }}</p>
```

Great! We're now able to translate our content within the view layer without polluting your controllers logic with translation logic; however, we could achieve the same result without using angular-translate at all, since our app only knows about one language.

Let's see angular-translate's real power and learn how to teach our app more than one language.

Multi-language support

You've already learned how to add a translation table to your app using `$translateProvider.translations()`.

The `$translateProvider` knows one language, as we set it with the `$translateProvider.translations()` method. Now, we can add an additional language in the same way by providing a second **translation table**.

When we set our first translation table, we can provide it a key (a language key) that specifies the language we're translating. We can simply add another translation key with another language key.

Let's update our app to include a second language:

```
app.config(function ['$translateProvider', ($translateProvider) { $translateProvider.translations('en-US', { HEADLINE: 'Hello there, This is my awesome app!', INTRO_TEXT: 'And it has i18n support!' }); }]); ""
```

To add a second translation table for another language, let's say German, just do the same with a different language key:

```
app.config(function ['$translateProvider', ($translateProvider) { $translateProvider.translations('en', { HEADLINE: 'Hello there, This is my awesome app!', INTRO_TEXT: 'And it has i18n support!' }]); }]); ""
```

```
}); .translations('de', { HEADLINE: 'Hey, das ist meine großartige App!', INTRO_TEXT: 'Und sie unterstützt mehrere Sprachen!' });}); ""
```

Now our app knows about two different languages. We can add as many languages as needed, there's no limit; however, since there are now two languages available, how does our app know which language should to use? `angular-translate` doesn't prefer any language until you tell it to do so.

To set a *preferred* language, we can use the method `$translateProvider.preferredLanguage()`. This method tells `angular-translate` which of the registered languages is the one that our app should use, by default. It expects an argument with the value of the language key, which points to a certain translation table.

Now, let's tell our app that it should use English as its default language:

```
app.config(function ['$translateProvider', ($translateProvider) { $translateProvider.translations('en', { HEADLINE: 'Hello there, This is my awesome app!', INTRO_TEXT: 'And it has i18n support!' }) .translations('de', { HEADLINE: 'Hey, das ist meine großartige App!', INTRO_TEXT: 'Und sie unterstützt mehrere Sprachen!' }); $translateProvider.preferredLanguage('en'); }]); ""
```

Switching the language at runtime

To switch to a new language at runtime, we have to use `angular-translate`'s `$translate` service. It has a method `uses()` that either returns the language key of the currently used language, or, when passing a language key as argument, tells `angular-translate` to use the corresponding language.

To get a feeling for how this capability works in a real app, add two new translation id's that represent translations for buttons you'll add later in your HTML template:

```
app.config(function ['$translateProvider', ($translateProvider) { $translateProvider.translations('en', { HEADLINE: 'Hello there, This is my awesome app!', INTRO_TEXT: 'And it has i18n support!', BUTTON_TEXT_EN: 'english', BUTTON_TEXT_DE: 'german' }).translations('de', { HEADLINE: 'Hey, das ist meine großartige App!', INTRO_TEXT: 'Und sie unterstützt mehrere Sprachen!', BUTTON_TEXT_EN: 'englisch', BUTTON_TEXT_DE: 'deutsch' }); $translateProvider.preferredLanguage('en'); }]); ""
```

Next, implement a function on a controller that uses the `$translate` service and its `uses()` method to change the language at runtime. To do that, we'll inject the `$translate` service in our app's controller and add a function on its `$scope`:

```
app.controller('TranslateCtrl', ['$translate', '$scope', function ($translate, $scope) { $scope.changeLanguage = function (langKey) { $translate.uses(langKey); }; }]); ""
```

Now, let's reflect this change in the HTML template by adding a button for each language. We'll also have to set up an `ng-click` directive on each button, which calls the function that changes the language at runtime:

```
1 <div ng-controller="Ctrl">
2   <button ng-click="changeLanguage( 'de' )" translate="BUTTON_TEXT_DE"></button>
3   <button ng-click="changeLanguage( 'en' )" translate="BUTTON_TEXT_EN"></button>
4 </div>
```

Et voilà! We now have an app with multi-language support!

Loading languages

What fun would it be if we were going to set the languages statically? We can dynamically load languages thanks to Angular's `$http` service, through the `$translateProvider`'s `registerLoader` function.

First, we need to install the `loader-url` extension by setting the `loader-url` service, which expects that there is a back-end server to send back JSON by handling the `lang` parameter. If you do have a back end that handles the route with the `lang` parameter, install the `loader-url` service with Bower like so:

```
1 bower install angular-translate-loader-url
```

If you prefer to have a service that loads static files, we can use the `static-files` loader that loads JSON files from a path with language files. Since this router is simpler, we'll go ahead and install this service through Bower:

```
1 bower install angular-translate-loader-static-files
```

Now, let's make sure this file is loaded in our view through a script tag:

```
1 <script src="/js/angular-translate-loader-url.min.js"></script>
```

To configure our service to use the `static-files` loader, we need to tell our `$translateProvider` to use the loader with a configuration object. The configuration object takes two parameters:

- `prefix` - which specifies the file prefix (including file paths)
- `suffix` - which specifies the file suffix (usually the extension)

The file loader attempts to fetch files at the following URL path: `[prefix]/[langKey]/[suffix]`. For instance, if we set our config object as:

```
$translateProvider.useStaticFilesLoader({ prefix: '/languages/' , suffix: '.json' });
```

The `angular-translate` attempts to load the `en_US` language from `/languages/en_US.json`. Using the `StaticFilesLoader` like so gives us the side benefit of **lazy-loading**. `$translate` will only pull down the language files it needs at runtime.

Of course, using asynchronous loading will cause a flash of untranslated content as the app loads. We can circumvent this side effect by setting a default language that is packaged with the app.

One last cool feature: We can use local storage to store our language files. `angular-translate` provides the ability to use local storage; this capability can be enabled with one function:

```
$translateProvider.useLocalStorage();“
```

We've covered how to use `angular-translate` to bring i18n support to your Angular app using `$translateProvider.translations()` and the `translate` filter. We've also shown how to change the language at runtime using `$translate` service and its `uses()` method.

Try out `angular-translate`; it comes with a lot of really nice built-in features, such as handling pluralization, using custom loaders, and setting translations through a service. The docs are fantastic; we suggest you check them out [here¹¹⁶](#).

There are a lot of examples with which you can play directly on the site! There's also an [API Reference¹¹⁷](#) that shows all available components and the interfaces you can use to build awesome apps with internationalization support!

angular-gettext

Similar to `angular-translate`, `angular-gettext` also provides translation in a completely different method. Rather than relying on us to embed our strings to be translated into the app, we'll abstract out the specific strings and let our library handle them.

`gettext` is an internationalization and localization system that is sponsored by the GNU Project and was first released in 1995. It's a popular system for wrapping new language support as it wraps strings that can later be translated to support new languages.

The `angular-gettext` library works in the same way, by providing string wrapping for strings we'll later want to translate.

Installation

To use `angular-gettext`, we need to load the `angular-gettext` library. We can install it in several different ways, but we prefer using Bower.

- Using Bower

We can install `angular-gettext` using Bower as:

¹¹⁶<http://pascalprecht.github.io/angular-translate>

¹¹⁷<http://pascalprecht.github.io/angular-translate/#/api>

```
1 $ bower install --save angular-gettext
```

Alternatively, we can download the minified version of `angular-gettext` from github.

Once we've installed the latest stable version of `angular-gettext`, we can simply embed it in our HTML document. Just make sure it's embedded after Angular itself, as it depends on the core `angular` library as well as `jquery` (as it's a dependency of `angular-gettext`)

```
1 <script src="path/to/jquery.js"></script>
2 <script src="path/to/angular.js"></script>
3 <script src="path/to/angular-gettext.js"></script>
```

Last but not least, our app has to declare `angular-gettext` as a load dependency:

```
1 var app = angular.module('myApp', ['gettext']);
``
```

Great, we're now ready to use `angular-gettext`'s components to translate our app!

Usage

The `angular-gettext` library includes the `translate` directive, which is simply a directive that gets placed on any of our DOM elements where we want the strings to be replaced by their translated counterparts.

```
1 <h1 translate>Hello!</h1>
```

Our `<h1>` contents will *automatically* be translated using the translated strings we'll define in a few minutes.

The strings to be translated don't need to be any more special than regular strings, giving us the power to have full interpolation support from within inside our app.

```
1 <h1 translate>Hello {{ name }}</h1>
```

We can also support translating plural notation as well. For instance, let's say we want to human-translate apples.

```
1 <h1 translate>One apple</h1>
```

We can add two more directives on to our `<h1>` element that signify the current count and the eventual string to translate.

```
1 <h1 translate
2   translate-n="count"
3   translate-plural="{{ count }} apples"
4   One apple
5 </h1>
```

If the string `translate-n` expression results in more than 1, then `gettext` will use the `translate-plural` string whereas otherwise it uses the value of the `<h1>` DOM element.

The extra `translate-n` directive accepts any angularjs expression including functions. For more information on expressions, check out the [expressions](#) chapter.

The `translate-plural` is simply a string that will replace the inner value of the DOM element the directive is called upon.

Lastly, we can also use the `translate` filter inside of our app. Sometimes we can't use a directive, for instance:

```
1 <input type="text" placeholder="Username" />
```

We *can* use the `translate` filter to substitute the `Username` value in the placeholder:

```
1 <input type="text"
2   placeholder="{{ 'Username' | translate }}" />
```

String extraction

Now, instead of providing the strings up-front that we'll need to translate, we'll *extract* the strings from our templates to build translations. We'll be generating a `.pot` file, the standard `gettext` template.

The *easiest* method for extracting these strings to be translated is by using the `grunt-angular-gettext` utility.



For more information about Grunt, check out the [Grunt](#) section in the [next steps](#) chapter.

In order to use the `grunt-angular-gettext` grunt task, we'll need to install it with `npm`:

```
1 $ npm install grunt-angular-gettext --save-dev
```

Once our grunt task is installed, we'll need to load it in our `Gruntfile`. Using the standard grunt method for referencing grunt tasks, we'll enable it inside of our `Gruntfile`:

```
1 grunt.loadNpmTasks('grunt-angular-gettext');
```

Once this is set, we'll need to extract the strings to be translated from our app. We can do this using the `nggettext_extract` task.

To set up this task, we'll need to provide our configuration.

Effectively, the most important key that matters in our `nggettext_extract` task is the `files` key:

```
1 grunt.initConfig({
2   nggettext_extract: {
3     pot: {
4       files: {
5         'po/template.pot': ['src/views/*.html']
6       }
7     },
8   },
9 });

```

We can also include a few options in our task that set the start and end delimiters. If we've configured angular to use different delimiters, we can set those in our task as options:

```
1 grunt.initConfig({
2   nggettext_extract: {
3     pot: {
4       options: {
5         startDelim: '/*',
6         endDelim: '*/'
7       },
8       files: {
9         'po/template.pot': ['src/views/*.html']
10      }
11    },
12  },
13 });

```

Now, we can run this task using the grunt like so:

```
1 $ grunt nggettext_extract
```

Once we're done running this task, we will have a `po/template.pot` file. For instance, for the template:

```
1 <div ng-controller="HomeCtrl">
2   <h1 translate>Hello {{ user.name }}</h1>
3   <h2 translate translate-n="count"
4     translate-plural="{{ count }} books">
5     {{ count }} books
6   </h2>
7 </div>
```

We'll get a `po/template.pot` file that looks like:

```
1 msgid ""
2 msgstr ""
3 "Content-Type: text/plain; charset=UTF-8\n"
4 "Content-Transfer-Encoding: 8bit\n"
5
6 #: app/index.html
7 msgid "Hello {{ user.name }}"
8 msgstr ""
9
10 #: app/index.html
11 msgid "{{ count }} books"
12 msgid_plural "{{ count }} books"
13 msgstr[0] ""
14 msgstr[1] ""
```

Translating our strings

Now that we have our `.pot` file ready to go, we can start translating it. One of the great reasons to use open-source software is that there are many tools available for us to create translations.

We're going to focus on using the `Poedit` tool, which is an open-source tool that will enable us to edit our `.pot` file.

First, we'll need to download which we can get from their site at www.poedit.net¹¹⁸.

Once we have this installed, let's open the application and select File -> New Catalog from POT File...:

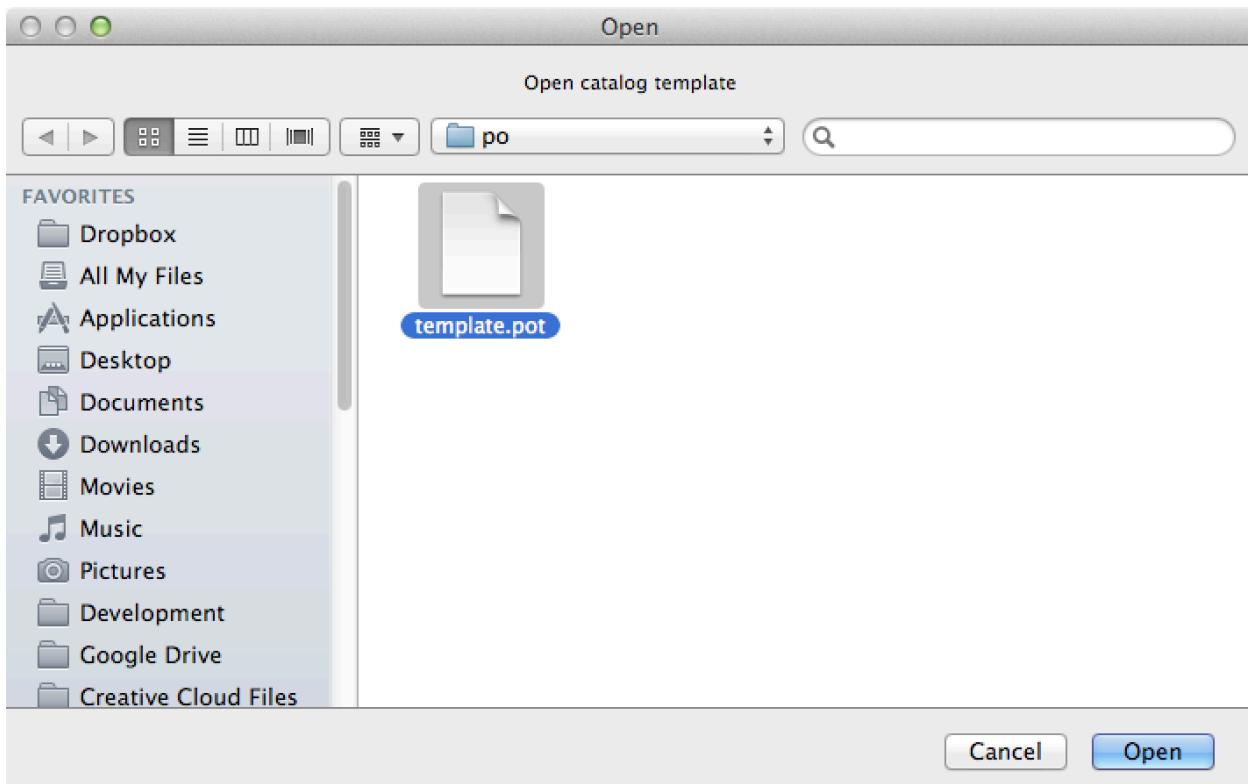
¹¹⁸<http://www.poedit.net/>

Catalogs Manager

- New Catalog...
- New Catalog from POT File...
- Open... ⌘O
- Close ⌘W
- Save ⌘S
- Save As... ⌘⌘S
- Export... ⌘⌘E

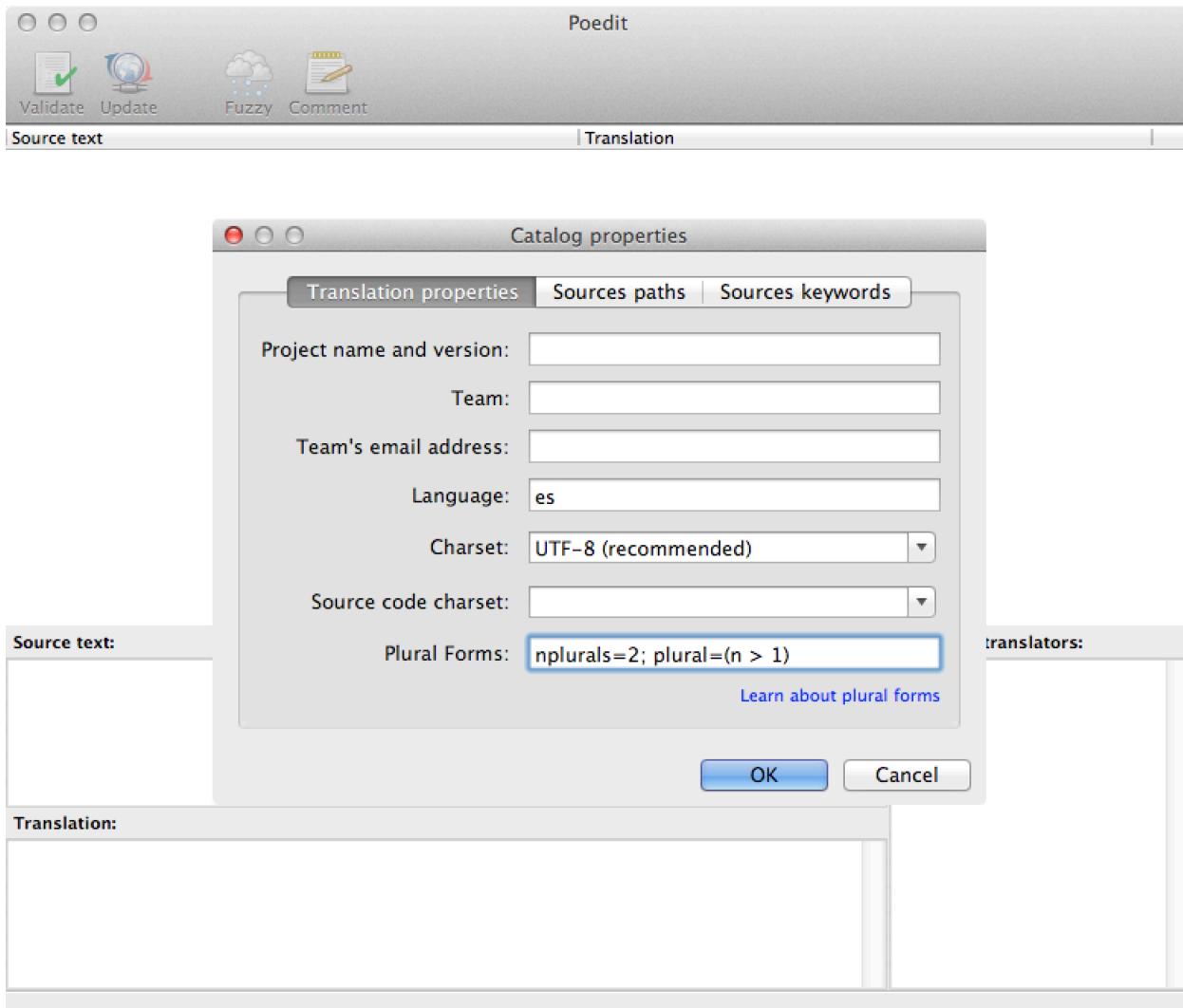
New Catalog from Pot File...

From here, find our file and select it.



Locate our .pot

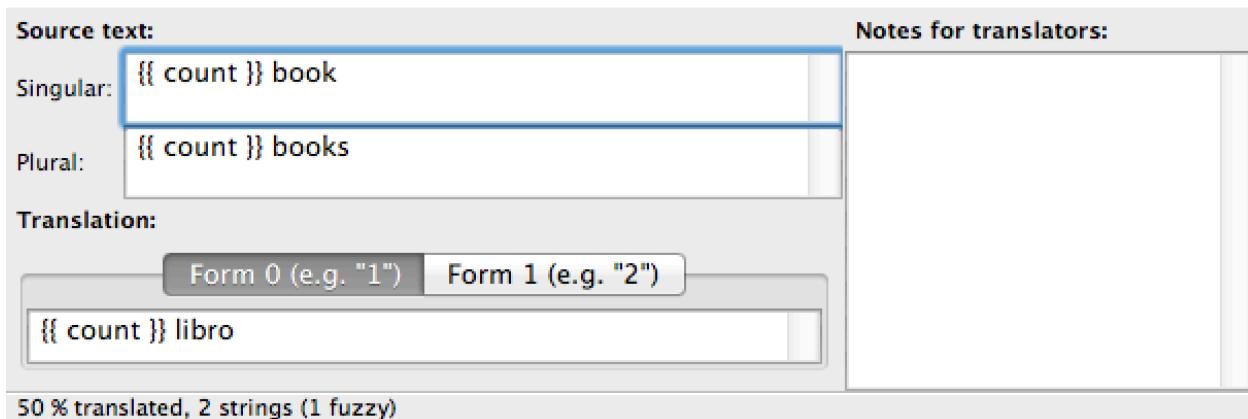
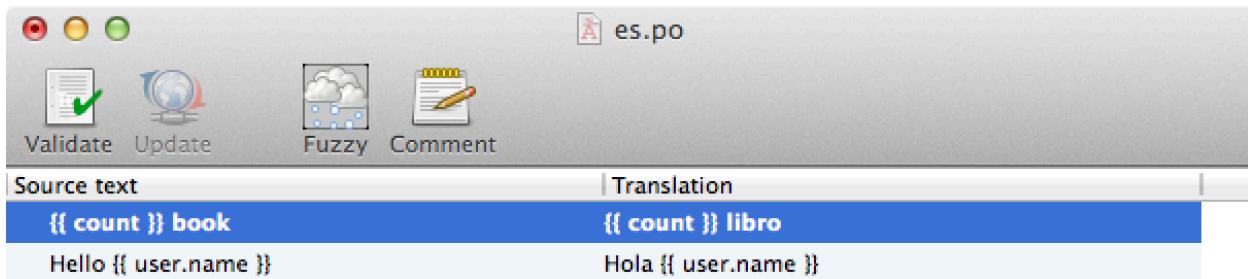
Now, we'll need to make sure that we include the plural form exactly as follows:



Setting plural forms

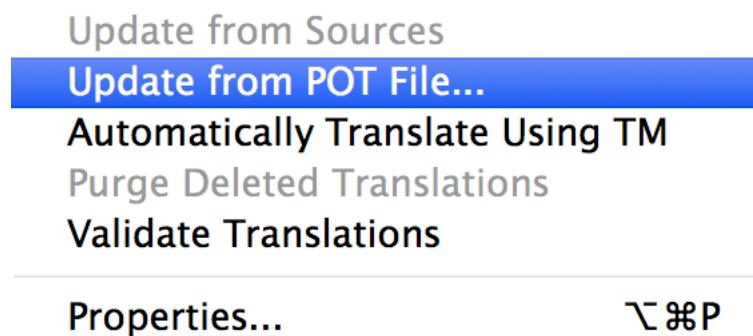
Select “OK” and we should be brought to a screen that shows our to-be translated strings. Fill these strings out for a specific language.

For instance, we’ll translate our app into Spanish, so we’ll use the es language. We’ll save our file as es.po in the same directory as our templates.pot file as is conventional for gettext.



Translating our app

Once we're done editing, we can save this file and continue on our way. Note, if we make changes to our app, we can simply re-run grunt and select “Update from POT File...” in poedit. This will update our new strings, remove old ones and make suggestions for those that have changed.



Translating our app

Compiling our new language

Finally, we can finally use our new compiled language formats to generate our new `translation.js` file.

We'll use `grunt` one more time to compile our new `.po` files into our `translations.js` file that we'll use in the runtime.

We'll need to add a new task, the `nggettext_compile` task which will take our `.po` files and wrap them into the language that we can use in our app.

The basic configuration task looks like this:

```
1  grunt.initConfig({
2      nggettext_compile: {
3          all: {
4              files: {
5                  'app/scripts/translations.js': ['po/*.po']
6              }
7          },
8      },
9  })
```

This configuration will generate the `app/scripts/translations.js` file out of all of the `po/*.po` files.

We can also specify a specific module that we want our translations into, such as:

```
1  grunt.initConfig({
2      nggettext_compile: {
3          all: {
4              options: {
5                  module: 'myApp.translations'
6              },
7              files: {
8                  'app/scripts/translations.js': ['po/*.po']
9              }
10         },
11     },
12  })
```



We suggest setting a Gruntfile task to call both of the `nggettext_*` functions, such as `grunt.registerTask('default', ['nggettext_extract', 'nggettext_compile']);`.

Now, when we run our grunt task, our `translations.js` file will be generated for us. We only need to include this file in our runtime `.html` file and our app will be ready for translation.

Changing languages

Now we're ready to set our language and use our translations to support different languages.

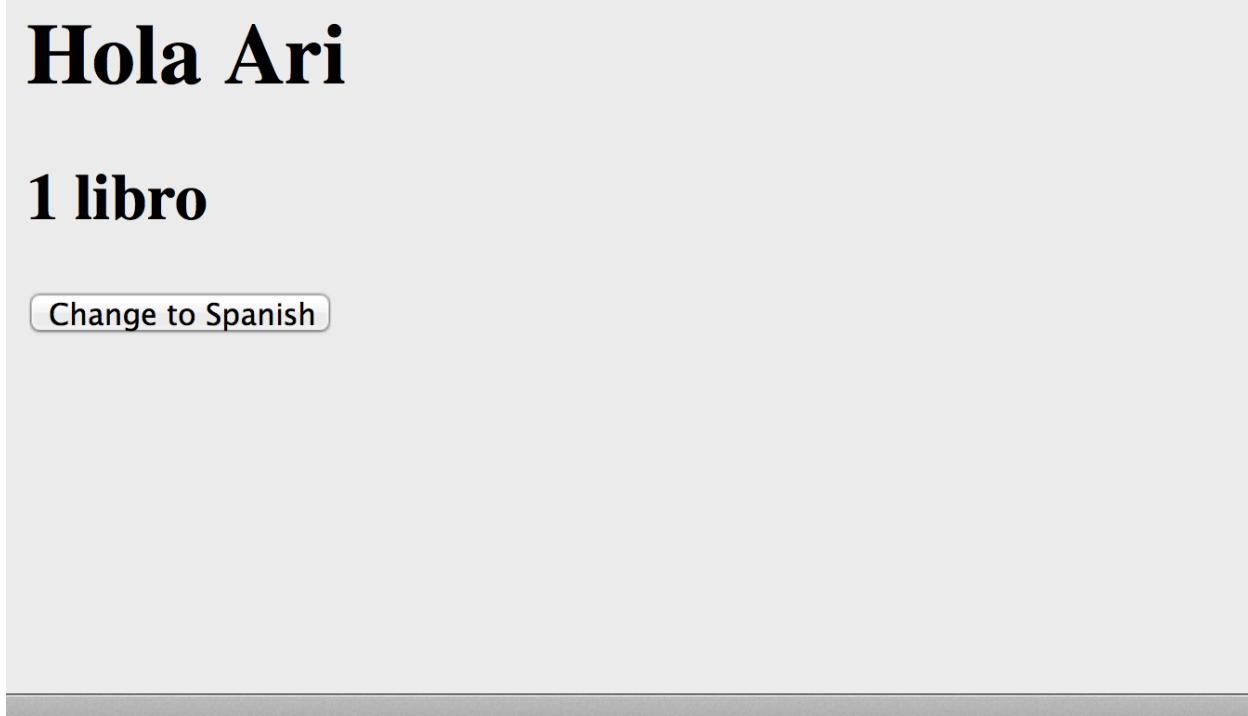
The `gettext` module includes a service called `gettextCatalog` that we can inject in our app to set the current language. To set the language, we can simply call:

```
1 angular.module('myApp')
2   .run(function gettextCatalog) {
3     gettextCatalog.currentLanguage = 'es';
4   });
```

This will load the current language as our Spanish language translations.

Note that we can also do this on-the-fly as we only need to *inject* the `gettextCatalog` into our angular object, like so:

```
1 .controller('HomeCtrl',
2   function($scope, gettextCatalog) {
3     $scope.user = { name: "Ari" }
4     $scope.count = 1;
5     $scope.changeLanguage = function() {
6       gettextCatalog.currentLanguage = 'es';
7     }
8   })
```



Security

With any client-side app, it's always a good idea to think about security at build-time. Additionally, it's relatively tough to deliver 100% protection in any situation and even more difficult to do it when the client can see the entire code.

In this chapter, we're going to take a look at some techniques for keeping our application secure. We'll look at how to master the \$sce service to secure our text input through wrapping authorized requests with tokens (when talking to a protected backend).

Strict Contextual Escaping, the \$sce service

The strict contextual escaping mode (available in Angular version 1.2 and higher include this by default) that tells our app that it *requires* bindgins in certain contexts to result in a value that is marked as safe for use inside the context.

For instance, when we want to bind raw HTML to an element using ng-bind-html, we'll want angular to render the element with HTML, rather than escaped text.

```
1 <textarea ng-model="htmlBody"></textarea>
2 <div ng-bind-html="{{htmlBody}}"></div>
```

\$sce is a fantastic service that allows us to write whitelisted, secure code by default and goes a long way in helping us prevent XSS and other vulnerabilities. With this power, it's important to understand what it is that we're doing so we can use it wisely.

In the above example, the <textarea> is bound to the htmlBody model. In this textarea, the user can input whatever arbitrary code they would like to get rendered into the div. For instance, this might be a live preview for writing a blog post or comments, etc.

If the user can input any arbitrary text into the text field, we have essentially opened ourselves up to a giant security hole.

In order to protect ourselves against malicious users, it's a good idea to run our unsafe text through a sanitizer.

The \$sce service does this for us, by default on all interpolated expressions. No constant literals are ever untrusted. For instance, this is always a trusted value as the value is a string.

```
1 <div ng-html-bind-unsafe="'<h1>Trusted</h1>'"></div>
```

Basically, at the root of embedded directives starting in version 1.2 and on, the values are not bound directly to the value of the binding, but to the result of the `$sce.getTrusted()` method.

Directives use the new `$sce.parseAs()` method instead of the `$parse` service to watch attribute bindings. The `$sce.parseAs()` method calls `$sce.getTrusted()` on all non-constant literals.

In effect, the `ng-bind-html` directive calls `$sce.parseAsHtml()` behind the scenes and binds the value to the DOM element. The `ng-include` directive runs this same behavior as well as any `templateUrl` defined on a directive.

When enabled, the built-in directives will call out to `$sce` automatically. It is possible that we can use this same behavior in our own directives and other custom components.

To set up `$sce` protection, we'll need to *inject* the `$sce` service.

```
1 angular.module('myApp', [])
2   .directive('myDirective', ['$sce',
3     function($sce) {
4       // We have access to the $sce service
5     }])
6   .controller('MyCtrl', [
7     '$scope', '$sce',
8     function($scope, $sce) {
9       // We have access to the $sce service
10    }]);

```

Inside of our directive and our controller, we'll want to give angular the ability to both allow *trusted* content back into the view as well as taking trusted interpolated input.

The `$sce` service has a simple API that gives us the ability to both set and get trusted content of explicitly specific types.

For instance, let's build an email previewer. This email client will allow users to write HTML in their email and we want to give them a live preview of their text.

The HTML we can use might look something like:

```
1 <div ng-app="myApp">
2   <div ng-controller="MyController">
3     <textarea ng-model="email.rawHtml"></textarea>
4     <pre ng-bind-html="email.htmlBody"></pre>
```

Now, notice that we are taking a body of text in a `<textarea></textarea>` on a different property of `email`; `email.rawHtml` vs. `email.htmlBody`. Inside our controller, we'll *parse* this `email.rawHtml` as HTML and output it to the browser.

Inside our controller, we can set up a `$watch` to monitor changes on the `email.rawHtml` and anytime it changes, run a trusted parser on the HTML content.

```
1 .controller('MyCtrl', [
2   '$scope', '$sce',
3   function($scope, $sce) {
4     // set up a watch on the email.rawHtml
5     $scope.$watch('email.rawHtml', function(v) {
6       // so long as we are not in the
7       // $compile phase
8       if (v) {
9         // Render the htmlBody as trusted HTML
10        $scope.email.htmlBody =
11          $sce.trustAsHtml($scope.email.rawHtml);
12      }
13    })
14  }]);
```

Now, anytime the contents of `email.rawHtml` changes, we'll run a parser on the content and get back suitable HTML contents. Note, that the content will be rendered as sanitized HTML that's safe to source in the application.

Now, what if we want to support the user to write custom javascript to execute on the page? For instance, if we want to enable the user to write an ecard that includes custom javascript, we'll want to enable the ability for them to run their javascript as well.

The HTML invocation for this might look like:

```
1 <textarea ng-model="email.rawJs"></textarea>
2 <pre ng-bind="email.jsBody"></pre>
3 <button ng-click="runJs()">Run</button>
```

With this snippet, we're running the same mechanism for parsing our raw text into *safe* text. This time, we also add a third element, a button that calls `runJs()` on our scope.

As we saw with our HTML bindings, we'll watch the javascript snippet

```
1 .controller('MyCtrl', [
2   '$scope', '$sce',
3   function($scope, $sce) {
4     // set up a watch on email.rawJs
5     $scope.$watch('email.rawJs', function(v) {
6       if (v) {
7         $scope.email.jsBody =
8           $sce.trustAsJs($scope.email.rawJs);
9       }
10    });
11  }]);
12]);
```

Notice, this time we did not use `trustAsHtml()`, but we used the `trustAsJs()` method. This tells Angular to parse the text as executable javascript code. At the end of this call, we'll have a safe, parsed javascript snippet we can `eval()` in the context of the application.

We can now enable the `runJs()` method to be filled out and run the javascript snippet supplied by `email.rawJs`.

```
1 // ...
2 $scope.runJs = function() {
3   eval($scope.email.jsBody.toString());
4 }
```

Note, there are more intelligent methods of running `eval` on javascript snippets. For production use, we recommend against using `eval`.

We get built-in protection by Angular as it will only load templates from the same domain and protocol as the app is loaded within. This is enforced by Angular calling the `$sce.getTrustedResourceUrl` on the `templateUrl`.

This does not replace the browser's Same Origin policies and Cross-Origin Resource Sharing, or CORS. These policies will still be in effect to protect the browser.

We can override this value by whitelisting or blacklisting domains with the `$sceDelegateProvider`.

Whitelisting urls

In the module's `config()` function, we can set new whitelist and blacklists.

```
1 angular.module('myApp', [])
2   .config(['$sceDelegateProvider',
3     function($sceDelegateProvider) {
4       // Set a new whitelist
5       $sceDelegateProvider.resourceUrlWhitelist(['self']);
6     }]);

```

To set a new whitelist, we'll use the `resourceUrlWhitelist()` method. The function takes one optional parameter.

- whitelist (array)

If a parameter is **not** passed, then this function serves as a getter and will return the currently set whitelist array.

If the whitelist parameter is passed in, then the array of will replace the `resourceUrlWhitelist` with the new array.

Each element of the array must either be a regex or the string 'self', which refers to all URLs to match against URLs of the same domain as the app. When a regex is used, it is matched against the absolute URL of the resource being tested.

If the array is empty empty, `$sce` will block ALL URLs.

Using 'self' will enable sourcing https resources from html documents.

To enable the every single url, to whitelist every domain:

```
1 angular.module('myApp', [])
2   .config(['$sceDelegateProvider',
3     function($sceDelegateProvider) {
4       // Set a new whitelist
5       $sceDelegateProvider.resourceUrlWhitelist(['.*']);
6     }]);

```

By default, the whitelist is set to `['self']`.

Blacklisting urls

It's also possible to blacklist URLs instead of whitelisting. It's often much safer to depend on whitelisting, but we can use them in combination. It's useful to whitelist a trusted domain and blacklist open redirects served by our domains.

To set a new blacklist, we'll use the `resourceUrlBlacklist()` method. This method takes one optional parameter.

- `blacklist` (array)

If a parameter is **not** passed in, then the function will return the currently set blacklist array.

If the `blacklist` parameter is passed in, then the blacklist is replaced by the new array.

Each element of the array must be either a regex or the string ‘self’, although in the case of the blacklist, it’s not useful. When a regex is used, it is matched against the absolute URL of the resource being tested.

The blacklist always has the *final say* in what is acceptable and what is not acceptable for trusted content.

By default, the blacklist is set to an empty array, `[]`.

\$sce API

The `$sce` library has two *main* functions we’ll be using as well as several helper functions.

getTrusted

To get the trusted version of a value of a specific type, we can call the `getTrusted()` method.

The `getTrusted()` method takes two arguments:

- `type` (string)

The type of context where the value will be used. See [sce types](#) for available types.

- `maybeTrusted`

This is the return value from `$sce.trustAs`. If it is invalid, then it will throw an exception.

The `$sce` library has a few helper methods for the `getTrusted()` method.

The following method calls are functionally equivalent:

<code>getTrustedCss(value)</code>	<code>getTrusted(\$sce.CSS, value)</code>
<code>getTrustedHtml(value)</code>	<code>getTrusted(\$sce.HTML, value)</code>
<code>getTrustedJs(value)</code>	<code>getTrusted(\$sce.JS, value)</code>
<code>getTrustedResourceUrl(value)</code>	<code>getTrusted(\$sce.RESOURCE_URL, value)</code>
<code>getTrustedUrl(value)</code>	<code>getTrusted(\$sce.URL, value)</code>

parse

Similar to the \$parse service, this converts an Angular expression into a function. If the expression is a literal constant, it will call to the \$parse service, otherwise it will call to the \$sce.getTrusted() service.

The parse() method takes two arguments:

- type (string)

The type of \$sce context where the value will be used. See [sce types](#) for available types.

- expression (string)

The Angular expression to compile.

The parse() method returns a function of the form: `function(context, locals)` where:

- context (object)

The object where the expression should be evaluated against. Typically, this will be a \$scope object.

- locals (object)

These are local variables, mostly useful for overriding values in the context.

The \$sce library has a few helper methods for the parse() method.

The following method calls are functionally equivalent:

parseAsCss(expr)	parseAs(\$sce.CSS, expr)
parseAsHtml(expr)	parseAs(\$sce.HTML, expr)
parseAsJs(expr)	parseAs(\$sce.JS, expr)
parseAsResourceUrl(expr)	parseAs(\$sce.RESOURCE_URL, expr)
parseAsUrl(expr)	parseAs(\$sce.URL, expr)

trustAs

The trustAs() method returns an object that is trusted by angular for use in a specific strict contextual escaping context. Bindings such as `ng-bind-html` and `ng-include` use the provided value.

The trustAs() method takes two arguments:

- type (string)

The type of `$sce` context where the value will be safe for us. See [sce types](#) for available types.

- `value`

The value that can be used to stand in for the provided value.

The `trustAs()` method returns a value that can be used where angular expects a `$sce.trustAs()` return value.

The `$sce` library has a few helper methods for the `trustAs()` method.

The following method calls are functionally equivalent:

<code>trustAsHtml(value)</code>	<code>trustAs(\$sce.HTML, value)</code>
<code>trustAsJs(value)</code>	<code>trustAs(\$sce.JS, value)</code>
<code>trustAsResourceUrl(value)</code>	<code>trustAs(\$sce.RESOURCE_URL, value)</code>
<code>trustAsUrl(value)</code>	<code>trustAs(\$sce.URL, value)</code>

isEnabled()

The `isEnabled()` method takes no parameters and returns a boolean that tells us if the `sce` environment is enabled. If it is, then it will return true, otherwise it will return false.

Configuring \$sce

If we want to completely disable the `sce` subsystem from running our app (although we discourage this as it provides security by default), we can do so in the `config()` function of our app:

```
1 angular.module('myApp', [])
2   .config(['$sceProvider',
3     function($sceProvider) {
4       // Turn off SCE
5       $sceProvider.enabled(false);
6     }]);
7 
```

Trusted context types

The `$sce` library has five built-in context types that are supported by default. These context types are what angular uses to parse and determine what is *safe* in one context vs. another.

Context	Description
\$sce.HTML	Tells Angular this is safe HTML to source in the app
\$sce.CSS	Tells Angular that it's safe to source this as CSS in the app
\$sce.URL	Tells Angular that the URL is safe to follow as a link
\$sce.RESOURCE_URL	Tells Angular that the URL is safe to follow as a link and that the contents are safe to include in the app
\$sce.JS	Tells Angular that the contents are safe to execute in the application

AngularJS and Internet Explorer

AngularJS works seamlessly with most modern browsers. Safari, Google Chrome, Google Chrome Canary, and Firefox work great. The notorious Internet Explorer version 8 and earlier can cause us problems.



For more information, read the AngularJS docs [guide on IE¹¹⁹](#).

If we are planning on releasing our applications for Internet Explorer v8.0 or earlier, we will need to pay some extra attention to help support it.

Internet explorer does not like element names that start with a prefix `ng:` as it considers it as an XML namespace. It will ignore these elements unless it has a corresponding namespace declaration:

```
1 <html xmlns:my="ignored">
```



This `xmlns:ng="http://angularjs.org"` makes IE feel more comfortable.

If we use non-standard HTML tags, then we need to create the tags in the head of the document for IE to recognize them. We can do this simply in the `head` element.

```
1 <!doctype html>
2 <html xmlns:ng="http://angularjs.org">
3   <head>
4     <!--[if lte IE 8]
5       <script>
6         document.createElement('ng-view');
7         // Other custom elements
8       </script>
9     <![endif]-->
10   </head>
11   <body>
12     <!-- ... -->
```

It is recommended that we use the attribute directive form as we don't need to create custom elements to support IE:

¹¹⁹<http://docs.angularjs.org/guide/ie>

```
1 <div data-ng-view></div>
```

To make AngularJS work with IE7 and earlier, we'll need to polyfill JSON.stringify. We can use the [JSON3¹²⁰](#) or [JSON2¹²¹](#) implementations.

In our browser, we'll need to conditionally include this file in the head. Download the file, store it in a location relative to the root of our application, and reference it in the head like so:

```
1 <!doctype html>
2 <html xmlns:ng="http://angularjs.org">
3   <head>
4     <!--[if lte IE 8]
5       <script src="lib/json2.js"></script>
6     <![endif]-->
7   </head>
8   <body>
9     <!-- ... -->
```

To use the ng-app directive with IE support, set the element id to ng-app as well.

```
1 <body id="ng-app" ng-app="myApp">
2 <!-- ... -->
```

We can take advantage of the angular-ui-utils library's ie-shiv module to help give us custom elements in our DOM.

In order to use the ui-utils ie-shiv library, we'll need to ensure we have the angular-ui library installed. Installation is easy. Download the ui-utils library and include the module. The ui-utils library can be found on github here: [https://github.com/angular-ui/ui-utils¹²²](https://github.com/angular-ui/ui-utils).

Ensure we've included the ui-utils in an accessible location to your app and include the file just like this:

¹²⁰<http://bestiejs.github.io/json3/>

¹²¹<https://github.com/douglascrockford/JSON-js>

¹²²<https://github.com/angular-ui/ui-utils>

```
1 <!--[if lte IE 8]>
2 <script type="text/javascript">
3   // define our custom directives here
4 </script>
5 <script src="lib/angular-ui-ieshiv.js"></script>
6 <![endif]-->
```

With that in place, we'll only activate the ie-shiv on Internet Explorer versions 8 and earlier. The shiv enables us to add our custom directives onto its global object, which will in turn create the proper declarations for IE.

The shiv library looks for the `window.myCustomTags` object. If it's defined, it will include these tags at load time along with the rest of the angular library directives:

```
1 <!--[if lte IE 8]>
2 <script type="text/javascript">
3   // define our custom directives here
4   window.myCustomTags = ['myDirective'];
5 </script>
6 <script src="lib/angular-ui-ieshiv.js"></script>
7 <![endif]-->
```

Ajax caching

IE is the only major browser that caches XHR requests. An efficient way to avoid this poor behavior is by setting an HTTP response header of `Cache-Control` to be `no-cache` for every request.

This is the default behavior for modern browsers and will help give a better experience for IE users.

We can change the default headers for every single request like so:

```
1 .config(function($httpProvider) {
2   $httpProvider.defaults
3     .headers.common['Cache-Control'] = 'no-cache';
4 });
```

SEO with AngularJS

Search engines, such as Google and Bing are engineered to crawl static web pages, not javascript-heavy, client-side apps. This is typical of a search engine which does not render javascript when the search bot is crawling over web pages.

This is because our javascript-heavy apps need a javascript engine to run, like PhantomJS or v8, for instance. Web crawlers typically load a web page without using a javascript interpreter.

Search engines do not include JS interpreters in their crawlers for good reason, they don't need to and it slows them down and makes them more inefficient for crawling the web.

Getting angular apps indexed

There are several different ways that we can tell Google to handle indexing our app. One, the more common approach is by using a backend to serve our angular app. This has the advantage of being simple to implement without much duplication of code.

The second approach is to render all of the content delivered by our angular app inside a `<noscript>` tag in our javascript, which we will cover lightly as it's mostly dependent upon how we deliver our apps to succinctly cover it without duplicating efforts.

Server-side

Google and other advanced search engines support the hashbang URL format, which is used to identify the current page that's being accessed at a given URL. These search engines transform this URL into a custom URL format that enables them to be accessible by the server.

The search engine visits the URL and expects to get the HTML that our browsers will receive, with the fully rendered HTML content. For instance, Google will turn the hashbang URL from:

1 `http://www.ng-newsletter.com/#!/signup/page`

Into the URL:

1 `http://www.ng-newsletter.com/?_escaped_fragment_=signup/page`

Within our angular app, we will need to tell Google to handle our site slightly differently depending upon which style we handle.

Hashbang syntax

Google's Ajax crawling specification was written and originally intended for delivering URLs with the hashbang syntax, which was an original method of creating permalinks for JS applications.

We'll need to configure our app to use the `hashPrefix` (default) in our routing:

```
1 angular.module('myApp', [])
2 .configure(['$location', function($location) {
3   $location.hashPrefix('!');
4 }]);
```

HTML5 routing mode

The new HTML5 pushState doesn't work the same way as it modifies the browser's URL and history. To get angular apps to "fool" the search bot, we can add a simple element to the header:

```
1 <meta name="fragment" content="!">
```

This tells the Google spider to use the new crawling spec to crawl our site. When it encounters this tag, instead of crawling our site like *normal*, it will revisit the site using the `?_escaped_fragment_=` tag.

This assumes that we're using HTML5 mode with the `$location` service:

```
1 angular.module('myApp', [])
2 .configure(['$routeProvider',
3   function($routeProvider) {
4     $routeProvider.html5Mode(true);
5   }]);

```

With the `_escaped_fragment_` in our query string, we can use our backend server to serve static HTML instead of our client-side app.

Now, our backend can detect if the request has the `_escaped_fragment_` in the request and and we can serve static HTML back instead of our pure angular app.



This can be accomplished using a proxy, like Apache or Nginx or our backend service. Setting these up is out of scope for this book, however we'll discuss how to set this up with a NodeJS app.

Options for handling SEO from the server-side

We have a number of different options available to us to make our site SEO-friendly. We'll walk through three different ways to deliver our apps from the server-side:

- Using node/express middleware
- Use Apache to rewrite URLs
- Use nginx to proxy URLs

Using node/express middleware

Note: although we are using NodeJS in this example, this is simply one implementation of how to serve static HTML snapshots to our backend. This technique works regardless of the backend you're using to deliver your HTML.

To deliver static HTML using NodeJS and Express (the web application framework for NodeJS), we'll add some middleware that will look for the `_escaped_fragment_` in our query parameters:

```
1 // ideas shared by
2 // http://johanneskueber.com/blog/2013/03/optimizing-angularjs-in-html5-mode-for-
3 seo-with-node-js-and-expressjs
4 // In our app.js configuration
5 app.use(function(req, res, next) {
6   var fragment = req.query._escaped_fragment_;
7
8   if (!fragment) return next();
9
10  // If the fragment is empty, serve the
11  // index page
12  if (fragment === "" || fragment === "/")
13    fragment = "/index.html";
14
15  // If fragment does not start with '/'
16  // prepend it to our fragment
17  if (fragment.charAt(0) !== "/")
18    fragment = '/' + fragment;
19
20  // If fragment does not end with '.html'
21  // append it to the fragment
22  if (fragment.indexOf('.html') === -1)
23    fragment += ".html";
24
25  // Serve the static html snapshot
26  try {
27    var file = __dirname + "/snapshots" + fragment;
28    res.sendfile(file);
29  } catch (err) {
30    res.send(404);
31  }
32});
```

This middleware expects our snapshots to exist in a top-level directory called '/snapshots' and serve files based upon the request path.

For instance, it will serve a request to / as index.html, while it will serve a request to /about as about.html in the snapshots directory.

Use Apache to rewrite URLs

If we're using the [apache server¹²³](#) to deliver our angular app, we can add a few lines to our configuration that will serve snapshots instead of our javascript app.

We can use the mod_rewrite mod to detect if the route being requested includes the _escaped_fragment_ query parameter or not. If it **does** include it, then we'll rewrite the request to point to the static version in the /snapshots directory.

In order to set the rewrite in motion, we'll need to enable the appropriate modules:

```
1 $ a2enmod proxy  
2 $ a2enmod proxy_http
```

Then we'll need to reload the apache config:

```
1 $ sudo /etc/init.d/apache2 reload
```

We can set the rewrite rules either in the virtualhost configuration for the site or the .htaccess file that sits at the root of the server directory.

```
1 RewriteEngine On  
2 Options +FollowSymLinks  
3 RewriteCond %{REQUEST_URI} ^/$  
4 RewriteCond %{QUERY_STRING} ^_escaped_fragment_=/(.*)$  
5 RewriteRule ^(.*)$ /snapshots/%1? [NC,L]
```

Use nginx to proxy URLs

If we're using [nginx¹²⁴](#) to serve our angular app, we can add some configuration to serve snapshots of our app if there is an _escaped_fragment_ parameter in the query strings.

Unlike Apache, nginx does not require us to enable a module, so we can simply update our configuration to replace the path with the question file instead.

In our nginx configuration file (For instance, /etc/nginx/nginx.conf), ensure your configuration looks like this:

¹²³<http://httpd.apache.org/>

¹²⁴<http://wiki.nginx.org/Main>

```

1 server {
2   listen 80;
3   server_name example;
4
5   if ($args ~ "_escaped_fragment_=/(.+)") {
6     set $path $1;
7     rewrite ^ /snapshots/$path last;
8   }
9
10  location / {
11    root /web/example/current/;
12    # Comment out if using hash urls
13    if (!-e $request_filename) {
14      rewrite ^(.*)$ /index.html break;
15    }
16    index index.html;
17  }
18 }
```

Once this is complete, we're good to reload our configuration:

```
1 sudo /etc/init.d/nginx reload
```

Taking snapshots

We can take snapshots of our HTML app to deliver our backend app, using a tool like [PhantomJS¹²⁵](#) or [zombie.js¹²⁶](#) to render our pages. When a page is requested by Google using the _escaped_fragment_ query parameter, we can simply return and render this page.

We'll discuss two methods to take snapshots, using zombie.js and using a grunt tool. We're not covering using the fantastic [PhantomJS¹²⁷](#) tool as there are plenty of great resources that demonstrate it.

Using Zombie.js to grab html snapshots

To set up [zombie.js¹²⁸](#), we'll need to install the npm package zombie:

¹²⁵<http://phantomjs.org/>

¹²⁶<http://zombie.labnotes.org/>

¹²⁷<http://phantomjs.org/>

¹²⁸<http://zombie.labnotes.org/>

```
1 $ npm install zombie
```

Now, we'll use NodeJS to save our file using `zombie`. First, a few helper methods we'll use in the process:

```
1 var Browser = require('zombie'),
2     url      = require('url'),
3     fs       = require('fs'),
4     saveDir = __dirname + '/snapshots';
5
6 var scriptTagRegex = /<script\b[^>]*(?:<!-->)<!--[^&gt;]*--&gt;/gi;
7
8 var stripScriptTags = function(html) {
9     return html.replace(scriptTagRegex, '');
10}
11
12 var browserOpts = {
13     waitFor: 2000,
14     loadCSS: false,
15     runScripts: true
16}
17
18 var saveSnapshot = function(uri, body) {
19     var lastIdx = uri.lastIndexOf('#/');
20
21     if (lastIdx &lt; 0) {
22         // If we're using html5mode
23         path = url.parse(uri).pathname;
24     } else {
25         // If we're using hashbang mode
26         path =
27             uri.substring(lastIdx + 1, uri.length);
28     }
29
30     if (path === '/') path = "/index.html";
31
32     if (path.indexOf('.html') === -1)
33         path += ".html";
34
35     var filename = saveDir + path;
36     fs.open(filename, 'w', function(e, fd) {
37         if (e) return;</pre>
```

```
38     fs.write(fd, body);
39   });
40 };
```

Now all we need to do is run through our pages, turn every link from a relative link into an absolute link (so the crawler can follow them), and save the resulting html.

We're setting a relatively high `waitFor` in the browser options above. This will cover 90% of the cases we care about. If we want to get more precise on how and when we take a snapshot, instead of waiting the 2 seconds we'll need to modify our angular app to fire an event and listen for the event in our zombie browser.

Since we like to automate as much as possible and prefer not to muck with our angular code, we prefer to set our timeout relatively high to attempt to let the code settle down.

Our `crawlPage()` function:

```
1 var crawlPage = function(idx, arr) {
2   // location = window.location
3   if (idx < arr.length) {
4     var uri = arr[idx];
5     var browser = new Browser(browserOpts);
6     var promise = browser.visit(uri)
7       .then(function() {
8
9       // Turn links into absolute links
10      // and save them, if we need to
11      // and we haven't already crawled them
12      var links = browser.querySelectorAll('a');
13      links.forEach(function(link) {
14        var href = link.getAttribute('href');
15        var absUrl = url.resolve(uri, href);
16        link.setAttribute('href', absUrl);
17        if (arr.indexOf(absUrl) < 0) {
18          arr.push(absUrl);
19        }
20      });
21
22      // Save
23      saveSnapshot(uri, browser.html());
24      // Call again on the next iteration
25      crawlPage(idx+1, arr);
26    });
27  }
28 }
```

Now we can simply call the method on our first page:

```
1 crawlPage(0, ["http://localhost:9000"]);
```

Using grunt-html-snapshot

Our preferred method of taking snapshots is by using the grunt tool `grunt-html-snapshot`. Since we use [yeoman¹²⁹](#) and grunt is already in our build process, we set up this task to run after we make a release of our apps.

To install `grunt-html-snapshot`, we'll use npm like so:

```
1 npm install grunt-html-snapshot --save-dev
```

If we're not using [yeoman¹³⁰](#), we'll need to include this task as a grunt task in our `Gruntfile.js`:

```
1 grunt.loadNpmTasks('grunt-html-snapshot');
```

Once this is set, we'll set some configuration about our site. To set up configuration, we'll create a new config block in our `Gruntfile.js` that looks like:

```
1 htmlSnapshot: {
2   debug: {
3     options: {}
4   },
5   prod: {
6     options: {}
7 }
8 }
```

Now we simply get to add our different options for the different stages:

¹²⁹<http://yeoman.io>

¹³⁰<http://yeoman.io>

```
1 htmlSnapshot: {
2   debug: {
3     options: {
4       snapshotPath: 'snapshots/',
5       sitePath: 'http://127.0.0.1:9000/',
6       msWaitForPages: 1000,
7       urls: [
8         '/',
9         '/about'
10      ]
11    }
12  },
13  prod: {
14    options: {}
15  }
16 }
```

To see a list of the entire available configuration options, check out the documentation page at <https://github.com/cburgdorf/grunt-html-snapshot¹³¹>.

Prerender.io

Alternatively, we can use an open-source tool such as [Prerender.io¹³²](#), which includes a node server that renders our site on-the-fly and an express middleware that communicates with the backend to prerenderHTML on-the-fly.

Essentially, prerender.io will take a url and returns the rendered HTML (with no script tags). Essentially, the prerender server we'll deploy will be called from our app like so:

```
GET http://our-prerenderserver.com/http://localhost:9000/#/about
```

This GET will return the rendered content of our #/about page.

Setting up a prerender cluster is actually pretty easy to do. We'll also show you how to integrate your own prerender server into your node app. Prerender.io is also avaialble for Ruby on Rails through a gem, but we won't cover how to set it up.

Setting up our own server to run it is pretty easy. Simply run the `npm install` to install the dependencies and run the command through either foreman or node:

¹³¹<https://github.com/cburgdorf/grunt-html-snapshot>

¹³²<http://prerender.io/>

```
1 $ npm install
2 $ node index.js
3 # Or through foreman
4 $ foreman start
```

The prerender library is also convenient to run on heroku:

```
1 $ git clone https://github.com/collectiveip/prerender.git
2 $ heroku create
3 $ git push heroku master
```

We store our rendered HTML in S3, so we recommend you use the built-in s3 cache. Read the docs how to set this up [here](#)¹³³.

After our server is running, we just need to integrate the fetching through our app. In express, this is very easy using the node library prerender-node.

To install prerender-node, we'll use npm:

```
1 $ npm install --save prerender-node
```

After this is installed, we'll tell our express app to use this middleware:

```
1 var prerender =
2   require('prerender-node').set('prerenderServiceUrl', 'http://our-prerenderserve\
3 r.com/');
4 app.use(prerender);
```

And that is it! This tells our express app that if we see a crawler request (defined by having the _escaped_fragment_ or the user agent string), then make a GET request to our prerender service at the appropriate url and get the prerendered HTML for the page.

We can also use the <noscript> tag to render our pages without needing to resort to using a server backend. Unfortunately, this is complex in that for all of our pages, we'll need to copy all of the elements of the page from outside of the <noscript> tag into a noscript tag.

This can become cumbersome and take a lot of work to keep the two in-sync.

¹³³<https://github.com/collectiveip/prerender#s3-html-cache>

Building Angular Chrome apps

The [Chrome¹³⁴](#) web browser is Google's custom browser. Not only is it incredibly speedy and on the bleeding edge of web development, it is at the forefront of delivering web experiences both on and off the web.

Chrome Apps are embedded applications that run within the web browser, but are intended on delivering a native app feel. Since they run within Chrome itself, they are written in HTML5, javascript, CSS3, and have access to native-like capabilities that true web applications do not.

Chrome apps have access to the Chrome API and services and can provide a integrated desktop-like experiences to the user.

One more interesting differentiation between Chrome apps and webapps is that they always load locally, so they show up immediately, rather than waiting for the network to fully download the components. This greatly improves the performance and our user's experience with running our apps.

Understanding the Chrome apps

Let's dive into looking at how Chrome apps actually work and how we can start building our own. Every Chrome application has three core files:

manifest.json

The `manifest.json` file that describes the meta-data about the application, such as the name, description, version, and how to launch our application.

A background script

The background script that sets up how our application responds to system-level events, such as a user installing our app or launching it, etc.

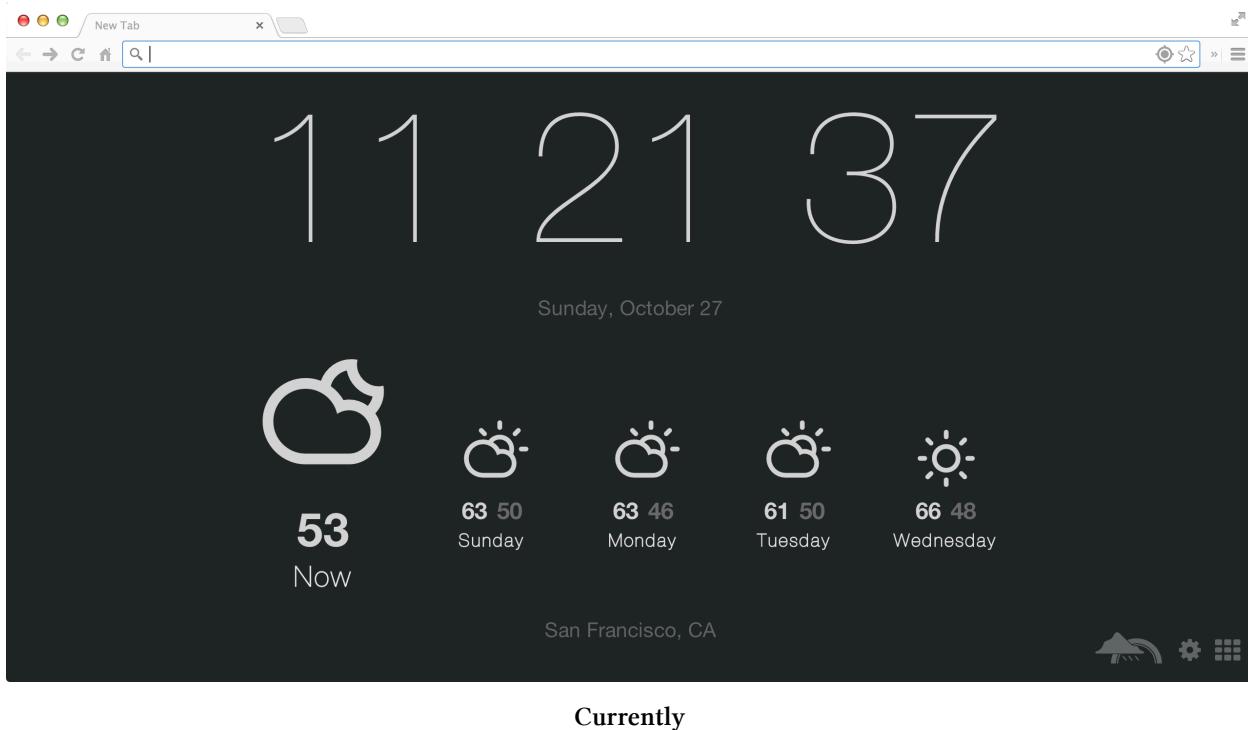
A view

Most Chrome applications have a view. This component is optional, but will most generally always be used for our applications.

¹³⁴<https://www.google.com/intl/en/chrome/browser/>

Building our Chrome app

In this section, we'll walk through how to create an advanced Chrome application using Angular. We're going to create a clone of the *fantastic* chrome webapp *Currently* by the team at Rainfall¹³⁵.



We'll be building a clone that we'll call *Presently*.

Architecting Presently

When we're building Presently, we'll need to take into account the application architecture. This will give us insight into how we'll build the app when we get to code.

Like *Currently*, *Presently* will be a “newtab” app. This means that it will launch every time we open a new tab.

Presently has two main screens:

- The home screen

This is the screen that features the current time and the current weather. It also features several weather icons beside the weather.

¹³⁵<http://blog.rainfalldesign.com/>

- The settings screen

This screen will allow our users to change their location within the app.

In order to support the home screen, we'll need to be able to show a properly formatted date and time as well as fetch weather from a remote API service.

To support the settings screen, we'll integrate with a remote API service to auto-suggest potential locations for an input box.

Finally, we'll use the basic localstorage (session storage) to persist our settings across the app.

Building the skeleton

Building our app, we'll set up a file structure like so:

```
$ tree
.
├── css
│   └── main.css
├── fonts
└── js
    ├── app.js
    └── vendor
        └── angular.min.js
├── manifest.json
└── tab.html
└── templates

5 directories, 5 files
$
```

File structure

We'll place our css files in `css/`, our custom fonts in `fonts/`, and our javascript files in `js/`. The main javascript file will be set in the `js/app.js` file and the HTML for our app will be placed in `tab.html` at the root.



There are great tools to help bootstrap Chrome app extensions such as [yeoman¹³⁶](#).

¹³⁶<http://yeoman.io>

Before we can start up our Chrome extension, we'll need to grab a few dependencies.

We'll grab the *latest* version of [angular.min.js¹³⁷](#) (1.2.0*) as well as [angular-route.min.js¹³⁸](#) from [angularjs.org¹³⁹](#) and save them to the `js/vendor/` directory.

Lastly, we'll use twitter's bootstrap 3 framework to style our app, so we'll need to get the `bootstrap.min.css` and save it to `css/` from [getbootstrap.com¹⁴⁰](#).



In production, it's often more efficient when working with multiple developers to use a tool like [Bower¹⁴¹](#) to manage dependencies. Since we're building a newtab app, however it's important we keep our app lightweight so it launches quickly.

manifest.json

With every Chrome app we'll write, we'll need to set up a `manifest.json`. This manifest tells Chrome how the application should run, what files it should use, what permissions it has, etc. etc.

Our `manifest.json` will need to describe our app as *newtab* app as well as describing the `content_security_policy` (the policies that describe what our application can and cannot do) and the background script (needed by Chrome).

```

1  {
2      "manifest_version": 2,
3      "name": "Presently",
4      "description": "A currently clone",
5      "version": "0.1",
6      "permissions": [],
7      "background": {
8          "scripts": ["js/vendor/angular.min.js"]
9      },
10     "content_security_policy": "script-src 'self'; object-src 'self'",
11     "chrome_url_overrides" : {
12         "newtab": "tab.html"
13     }
14 }
```

The `manifest.json` is relatively straightforward with the name, the `manifest_version`, the `version`, etc. In order to tell Chrome to launch our app as a *newtab* app, we set the app to *override* the *newtab* page.

¹³⁷<http://code.angularjs.org/1.2.0-rc.3/angular.min.js>

¹³⁸<http://code.angularjs.org/1.2.0-rc.3/angular-route.min.js>

¹³⁹<http://angularjs.org/>

¹⁴⁰<http://getbootstrap.com/>

¹⁴¹<http://bower.io/>

tab.html

The main HTML file for our application is the `tab.html` file. This is the file that will be loaded when we open a new tab in Chrome.

We'll set up the basic angular app inside of the `tab.html` file:

```
1 <!doctype html>
2 <html data-ng-app="myApp" data-ng-csp="">
3   <head>
4     <meta charset="UTF-8">
5     <title>Presently</title>
6     <link rel="stylesheet" href="css/bootstrap.min.css">
7     <link rel="stylesheet" href="css/main.css">
8   </head>
9   <body>
10    <div class="container">
11    </div>
12    <script src=".js/vendor/angular.min.js"></script>
13    <script src=".js/vendor/angular-route.min.js"></script>
14    <script src=".js/app.js"></script>
15  </body>
16 </html>
```

This very basic structure of an angular application looks almost identical to any angular app, with one exception: `data-ng-csp=""`.

The `ngCsp` directive enables Content Security Policy (or CSP) support for our angular app. Since Chrome apps prevent the browser from using `eval` or `function(string)` generated functions and Angular uses the `function(string)` generated function for speed, `ngCsp` will cause Angular to evaluate all expressions.

This compatibility mode comes as a cost of performance, however as it will execute operations much slower, but will not throw any security violations in the process.

CSP also forbids javascript files from inlining stylesheet rules, so we'll need to include `angular-csp.css` manually.

The `angular-csp.css` file can be found at <http://code.angularjs.org/snapshot/angular-csp.css>¹⁴².

Lastly, `ngCsp` must be placed alongside the root of our angular apps:

¹⁴²<http://code.angularjs.org/snapshot/angular-csp.css>

```
1 <html ng-app ng-csp>
```



Without the `ng-csp` directive, our Chrome app will **not** run as it will throw a security exception. If you see a security exception being thrown, make sure you check the root element for the directive.

Loading the app in Chrome

With our app in progress, let's load it into Chrome so we can follow our progress along in the browser. To load our app in Chrome, navigate to the url: `chrome://extensions/`.

Once there, click on the button “Load unpacked extension...” and find the root directory (the directory that contains our `manifest.json` file from above).

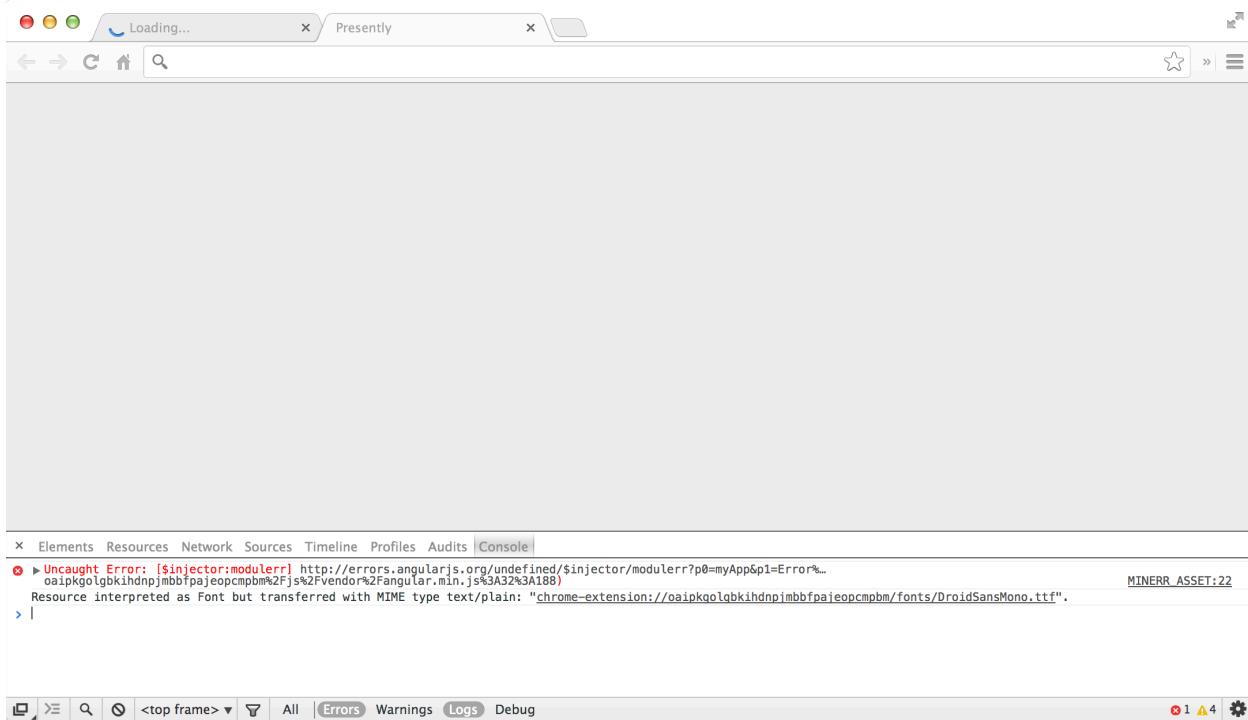
Extensions

The new Apps Developer Tools have been added to the application list.

Load unpacked extension...

Load unpacked extension

Once the application has been *loaded* into the Chrome browser, open a new tab and we should see our empty app with one error (don't worry, we'll fix this shortly):



Load unpacked extension



Anytime that we update or modify our `manifest.json` file, we'll need to click on the `Reload` link underneath our Chrome app in `chrome://extensions`.

The main module

Our entire angular application will be built in the `js/app.js` file. For production versions of our app, we may want to split this functionality into multiple files or use a tool like `grunt`¹⁴³ to compress and concatenate them for us.

Our app is called `myApp`, so we'll create an angular module with the same name:

```
1 angular.module('myApp', [])
```

With this, our app will run in the browser without any issues.

Building the homepage

We'll start by building the home section in our app. In this section, we'll work on putting together components of our app that will make the application run. In the next section, we'll set up the multi-route application.

¹⁴³<http://gruntjs.com/>

Building the clock

The main feature of *Presently* is the large clock that sits right at the top of the application and updates every second. In Angular, we can set this up pretty simply.

We'll first start by building a `MainCtrl` will be responsible for managing the home screen. Inside this `MainCtrl` controller, we'll set up a timeout that will tick every second and update a local scope variable.

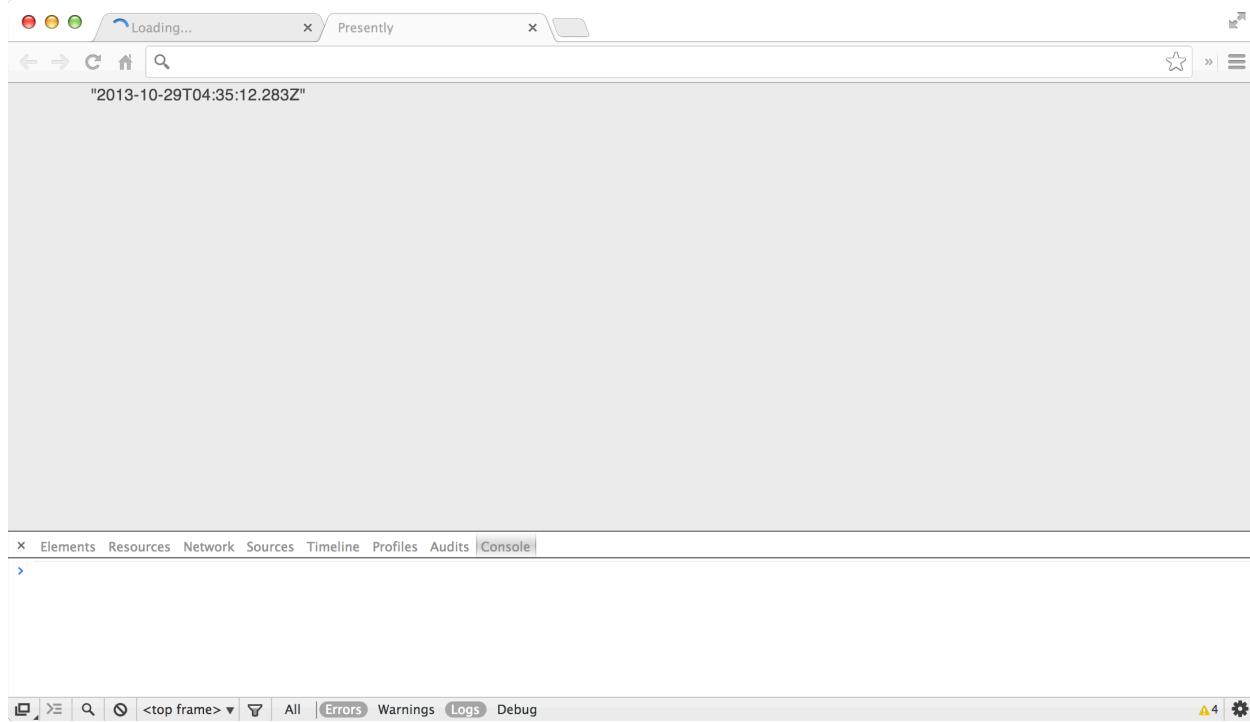
```
1 angular.module('myApp', [])
2   .controller('MainCtrl', function($scope, $timeout) {
3     // Build the date object
4     $scope.date = {};
5
6     // Update function
7     var updateTime = function() {
8       $scope.date.raw = new Date();
9       $timeout(updateTime, 1000);
10    }
11
12    // Kick off the update function
13    updateTime();
14  });
15
```

Every second that our `MainCtrl` is visible, the `updateTime()` function will be ran to update the `$scope.date.raw` timestamp and our view will be updated.

In order for us to see anything in the view load in our Chrome app, we'll need to bind this data to the document. We can set up this binding using the normal `{{ }}` template syntax:

```
1 <div class="container">
2   <div ng-controller="MainCtrl">
3     {{ date.raw }}
4   </div>
5 </div>
```

When we go back to the browser and refresh, we'll see an unformatted Date object ticking in the view:



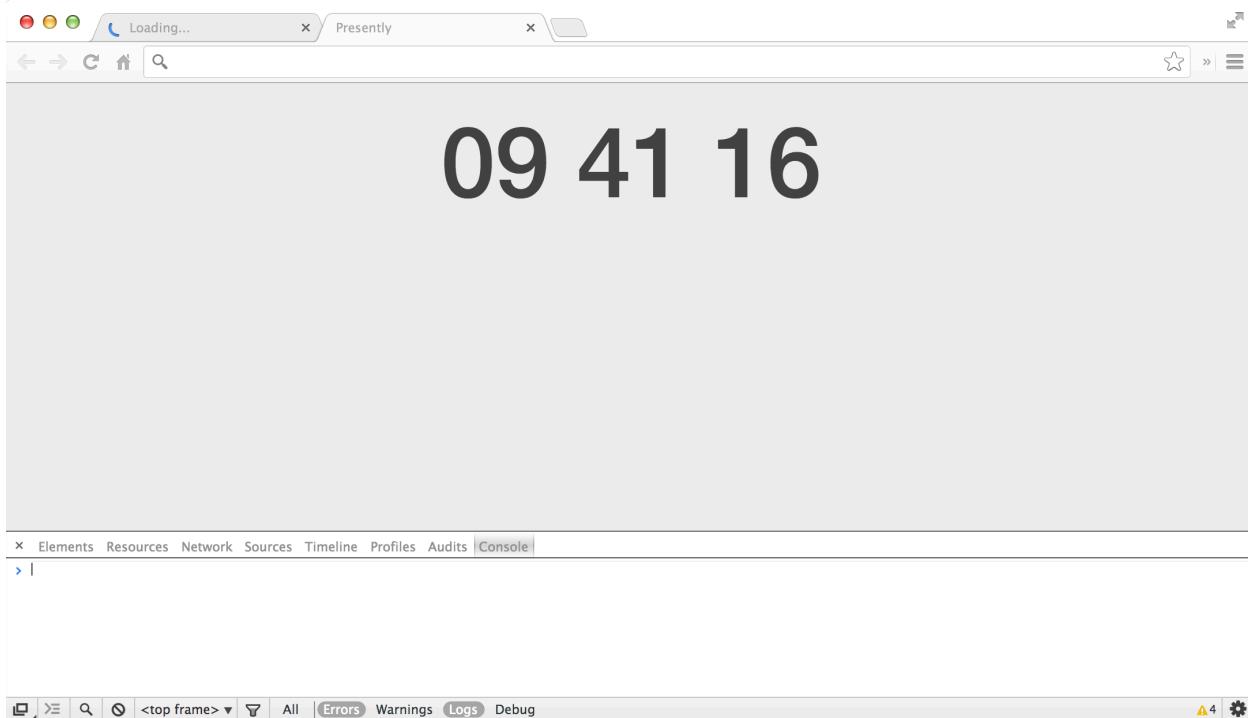
Unformatted date

The date is very ugly in the browser as it stands now. We can utilize Angular's built-in filters to format our date in a much more elegant manner.

Following along with how *Currently* formats the date in their homescreen, we'll format ours similarly. Updating the view, we will move our date into its own nested div and add formatting to display the date:

```
1 <div class="container">
2   <div ng-controller="MainCtrl">
3     <div id="datetime">
4       <h1>{{ date.raw | date:'hh mm ss' }}</h1>
5     </div>
6   </div>
7 </div>
```

With a little CSS and help from bootstrap, our dates will appear on the screen in a much more human-friendly format.



First screen

We're using the CSS rules to align the date and times to the center of the screen and increasing the font-size to be prominently displayed on-screen.

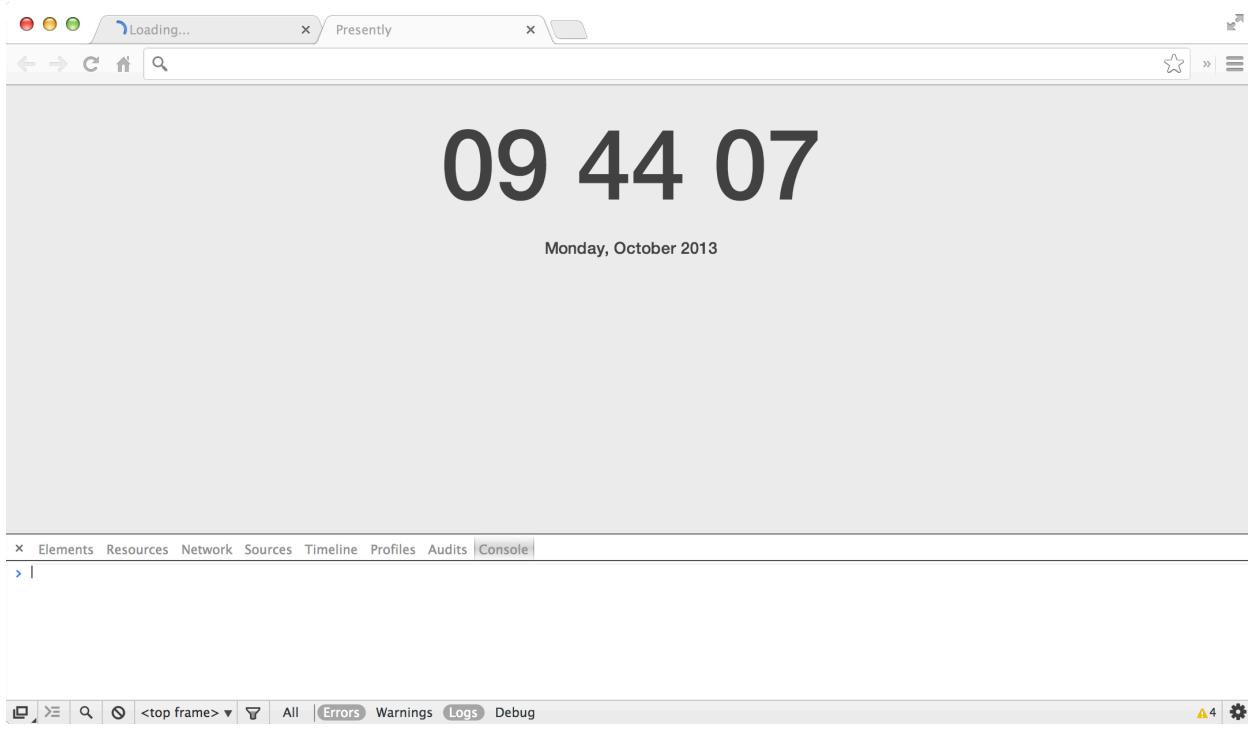
```
1 #datetime {  
2   text-align: center;  
3 }  
4 #datetime h1 {  
5   font-size: 6.1em;  
6 }
```

We can add a second date in our view that simply shows our date with a human-friendly display. This is simply a matter of adding a second formatted date:

```
1 <!-- ... -->  
2 <div id="datetime">  
3   <h1>{{ date.raw | date:'hh mm ss' }}</h1>  
4   <h2>{{ date.raw | date:'EEEE, MMMM yyyy' }}</h2>  
5 </div>  
6 <!-- ... -->
```

Our CSS for the `#datetime h2` tag simply increases the size of the `<h2>` tag:

```
1 #datetime h2 {  
2   font-size: 1.0em;  
3 }
```



Sign up for wunderground's weather API

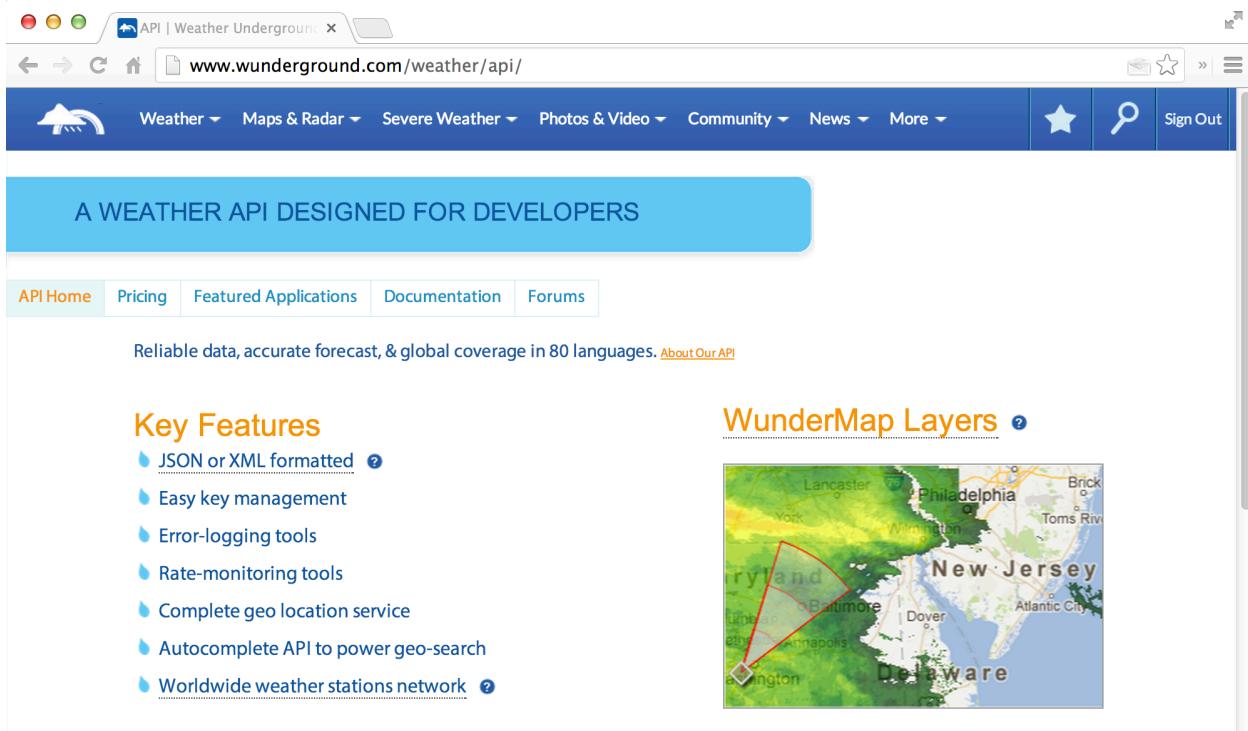
Our app will need to reach out to foreign sources to fetch the current weather for the location we're interested in. In this application, we're using the [wunderground¹⁴⁴](#) api.

In order to use the wunderground api, we'll need to get an access api key.

To get an access api key, we'll need to sign up first. Head to the weather api wunderground page at <http://www.wunderground.com/weather/api/>¹⁴⁵ and click "Sign Up for Free!".

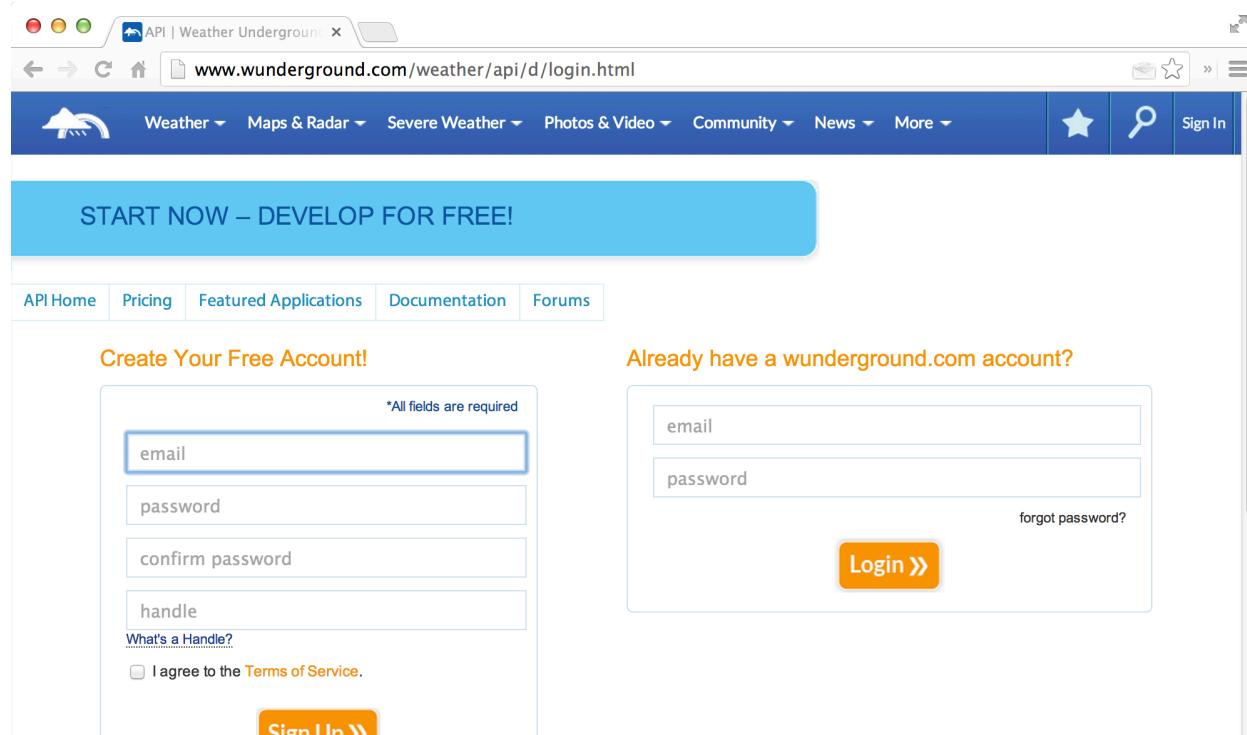
¹⁴⁴<http://www.wunderground.com/>

¹⁴⁵<http://www.wunderground.com/weather/api/>



The screenshot shows the Weather Underground API homepage. At the top, there's a navigation bar with links for Weather, Maps & Radar, Severe Weather, Photos & Video, Community, News, More, a sign-in button, and a search icon. Below the navigation is a blue header bar with the text "A WEATHER API DESIGNED FOR DEVELOPERS". Underneath this, there's a menu with "API Home" (which is highlighted in orange), Pricing, Featured Applications, Documentation, and Forums. A sub-menu below "API Home" lists "Reliable data, accurate forecast, & global coverage in 80 languages." followed by a link to "About Our API". To the left, under "Key Features", there's a list of seven items: JSON or XML formatted, Easy key management, Error-logging tools, Rate-monitoring tools, Complete geo location service, Autocomplete API to power geo-search, and Worldwide weather stations network. To the right, there's a section titled "WunderMap Layers" with a question mark icon, featuring a map of the Mid-Atlantic region (Maryland, Delaware, New Jersey, and parts of Pennsylvania) with a red polygon highlighting the Baltimore/Washington area. At the bottom center, there's a call-to-action button labeled "Sign up for wunderground".

Fill out the relevant details on the following page and we'll click through until we reach the detail page that shows our API key.



Fill out details

Once we're set, locate the wunderground api key and save it. We'll be using it shortly.

Building the angular service

We won't place our logic into the Controller to fetch the weather as it is both inefficient (as the controller will be blown away when we navigate to another page and we'll need to re-call the api every time the controller is loaded) and poor design to mix in business logic details with implementation details.

Instead, we'll use a [service](#). A service persists across controllers, for the duration of the application's lifetime and is the appropriate place for us to hide business logic away from the controller.

As we'll need to *configure* our app when it boots up, we'll use the `.provider()` method of creating a service. This is the **only** method for creating services that can be injected into `.config()` functions.

To build the service, we'll use the `.provider()` api method that takes both a name of the service as well as a function that defines the actual provider.

```
1 angular.module('myApp', [])
2   .provider('Weather', function() {
3     })
```

Inside here, we'll need to define a `$get()` function that returns the methods available to the service. To configure this service, we'll need to allow a method for the api key to be set on configuration. These methods will live outside of the scope of the `$get()` function.

```
1 .provider('Weather', function() {
2   var apiKey = "";
3
4   this.setApiKey = function(key) {
5     if (key) this.apiKey = key;
6   };
7
8   this.$get = function($http) {
9     return {
10       // Service object
11     }
12   }
13 })
```

With this minimal amount of code, we can now *inject* the Weather service into our `.config()` function and configure the service with our wunderground api key.

When angular encounters a provider created with the `.provider()` api method, it creates a `[Name]Provider` injectable object. This is what we'll inject into our config function:

```
1 // .provider('Weather', function() {
2 // ...
3 // })
4 .config(function(WeatherProvider) {
5   WeatherProvider.setApiKey('YOUR_API_KEY');
6 })
7 // .controller('MainCtrl', function($scope, $timeout) {
8 // ...
```

The wunderground API requires that we pass the API key with our request in the URL. In order to pass our api key in with every request, we'll create a function that will generate the url.

```
1 var apiKey = "";
2 // ...
3 this.getUrl = function(type, ext) {
4   return "http://api.wunderground.com/api/" +
5     this.apiKey + "/" + type + "/q/" +
6     ext + '.json';
7 };
```

Now, we can create our API call for the Weather service to get us the latest forecast data from the wunderground API.

We'll create our own promises that we can use to resolve in the view as we'll want to return only the relevant results from our API call:

```
1 this.$get = function($q, $http) {
2   var self = this;
3   return {
4     getWeatherForecast: function(city) {
5       var d = $q.defer();
6       $http({
7         method: 'GET',
8         url: self.getUrl("forecast", city),
9         cache: true
10       }).success(function(data) {
11         // The wunderground API returns the
12         // object that nests the forecasts inside
13         // the forecast.simpleforecast key
14         d.resolve(data.forecast.simpleforecast);
15       }).error(function(err) {
16         d.reject(err);
17       });
18       return d.promise;
19     }
20   }
21 }
```

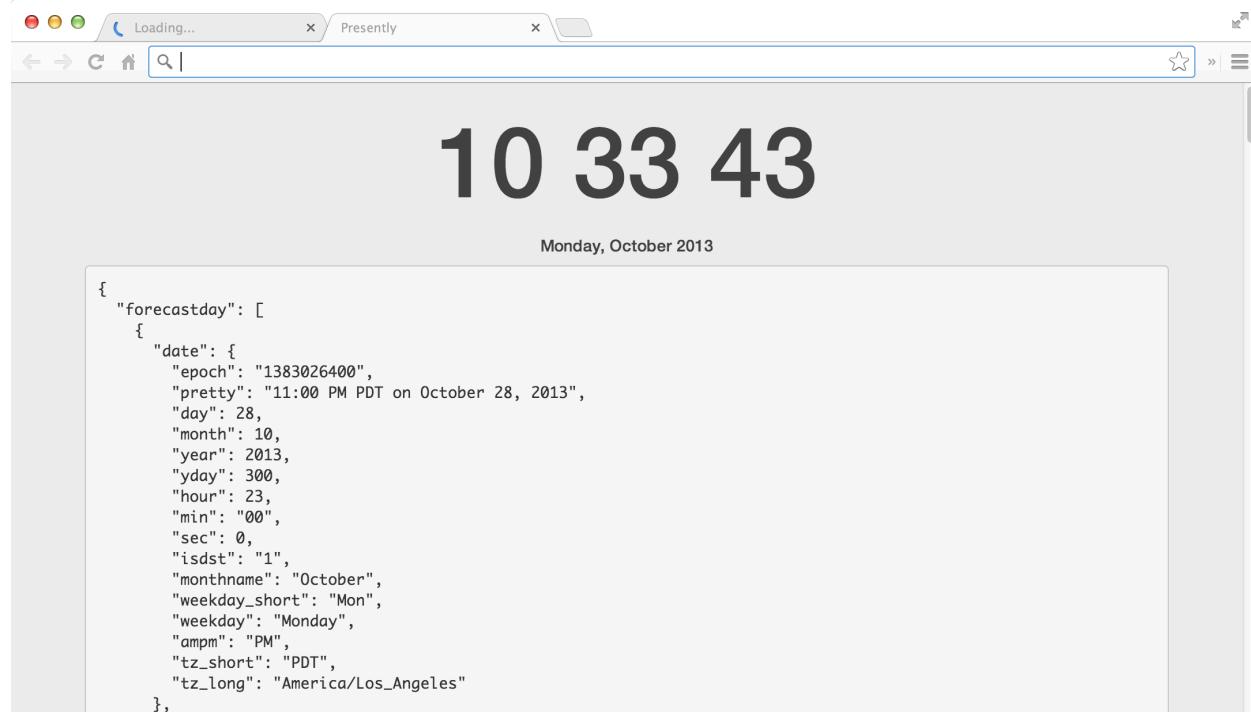
Now, we can inject the Weather service into our controller and simply call the method `getWeatherForecast()` and respond to the promise instead of dealing with the complexity of the API in our controller.

Back to our `MainCtrl`, we can inject the Weather service and set the result on our scope:

```
1 .controller('MainCtrl',
2   function($scope, $timeout, Weather) {
3     // ...
4     $scope.weather = {}
5     // Hardcode San_Francisco for now
6     Weather.getWeatherForecast("CA/San_Francisco")
7     .then(function(data) {
8       $scope.weather.forecast = data;
9     });
10    // ...
```

To view the result of the API call in our view, we'll need to update our `tab.html`. For debugging purposes, we like to use the `json` filter inside a `<pre>` tag:

```
1 <div id="forecast">
2   <pre>{{ weather.forecast | json }}</pre>
3 </div>
```

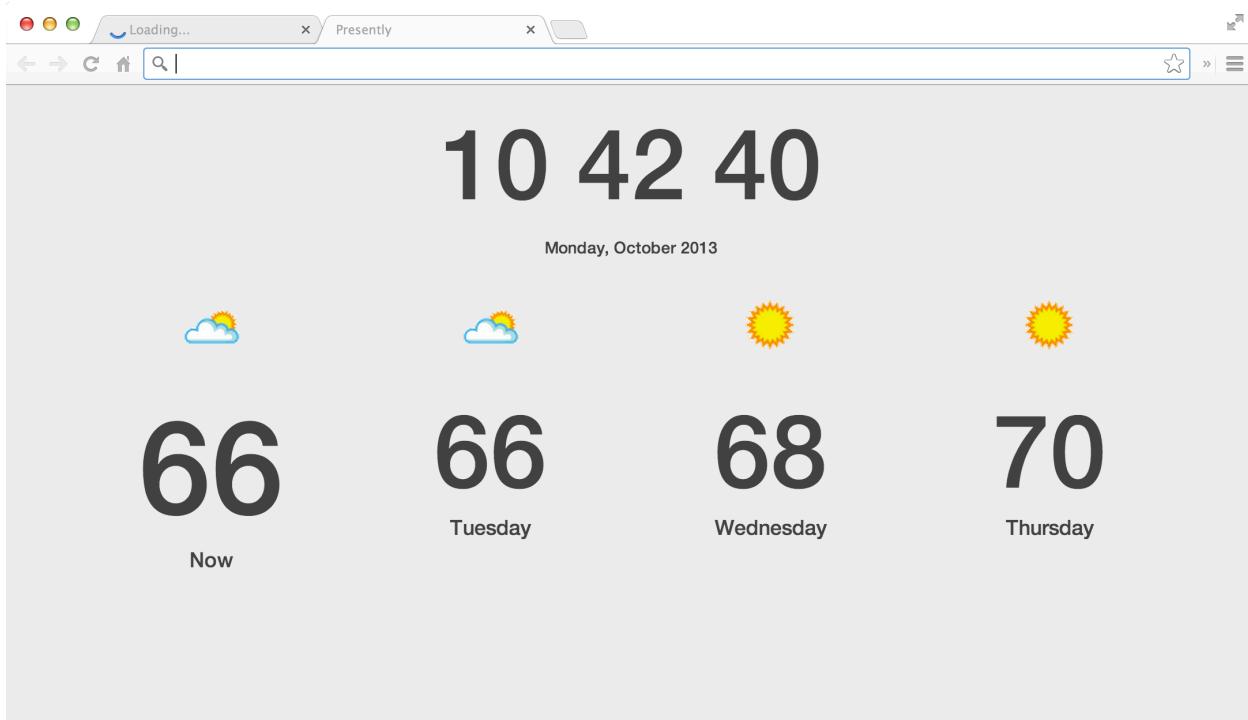


Weather API debugging call

We can see that the view is updated with the latest weather and now we're good to go to create a more polished view.

The view itself will iterate over the `forecast.forecastday` collection. For each element, we'll create a view that displays the weather icon given to us by the wunderground api as well as the human-readable date and high.

```
1 <div id="forecast">
2   <ul class="row list-unstyled">
3     <li ng-repeat="day in weather.forecast.forecastday" class="col-md-3">
4       <div ng-class="{today: $index == 0}">
5         
6         <h3>{{ day.high.fahrenheit }}</h3>
7         <h4 ng-if="$index == 0">Now</h4>
8         <h4 ng-if="$index != 0">{{ day.date.weekday }}</h4>
9       </div>
10      </li>
11    </ul>
12  </div>
```



Clean HTML weather view

The style we've set in the view is set as:

```
1 #forecast ul li {
2   font-size: 4.5em;
3   text-align: center;
4 }
5 #forecast ul li h3 {
6   font-size: 1.4em;
7 }
8 #forecast ul li .today h3 {
9   font-size: 1.8em;
10 }
```

A settings screen

Currently our app only has one view, with a hard-coded city that fetched for every browser. Although this works for all of us here in San Francisco, it does **not** work for anyone outside of it.

In order to allow our users the ability to customize their experience with *Presently*, we'll need to add a second screen: a setting screen.

To introduce a second screen (and multiple views), we'll need to add the `ngRoute` module as a dependency of our app module.

```
1 angular.module('myApp', ['ngRoute'])
```

Now we can define our separate views and routes as well as pull our home screen view out of the main `tab.html` view.

In defining our routes, note that we'll need two; one for each of the two different screens of our app.

```
1 // angular.module('myApp')
2 // ...
3 .config(function($routeProvider) {
4   $routeProvider
5     .when('/', {
6       templateUrl: 'templates/home.html',
7       controller: 'MainCtrl'
8     })
9     .when('/settings', {
10       templateUrl: 'templates/settings.html',
11       controller: 'SettingsCtrl'
12     })
13     .otherwise({redirectTo: '/'});
14 })
```

Now we can take our entire `tab.html` html between the `.container` div, move it into the file `templates/home.html`, and replace it with `<div ng-view></div>`.

```
1 <div class="container">
2   <div ng-view></div>
3 </div>
```

When we refresh the page, we'll see that nothing has appeared to have *changed*, but our html is no longer loaded inside the `tab.html`, but from the `templates/home.html` template.

Currently, we have no way of navigating between our two screens. We can add some footer-based navigation that we can allow our users to navigate between the pages. We'll simply add two links at the bottom of the page to navigate between pages, like so:

```
1 <div id="actionbar">
2   <ul class="list-inline">
3     <li><a class="glyphicon glyphicon-home" href="#"></a></li>
4     <li><a class="glyphicon glyphicon-cog" href="#/settings"></a></li>
5   </ul>
6 </div>
```

In order to add them to the the bottom right-hand corner of the screen, we'll apply a bit of CSS to absolutely position them:

```
1 #actionbar {
2   position: absolute;
3   bottom: 0.5em;
4   right: 1.0em;
5 }
6 #actionbar a {
7   font-size: 2.2rem;
8   color: #000;
9 }
```

Now, if we navigate to our settings page by clicking on the cog button, we'll see that nothing is rendered. We need to define our `SettingsCtrl` so we can start manipulating the view and working with our user.

```
1 // ...
2 .controller('SettingsCtrl',
3   function($scope) {
4     // Our controller will go here
5   })

```

The settings screen itself will feature a single form that will be responsible for allowing the user to change cities that they are interested in. The HTML itself will look similar to this (with a few features we have yet to implement):

```
1 <h2>Settings</h2>
2 <form ng-submit="save()">
3   <input type="text"
4     ng-model="user.location"
5     placeholder="Enter a location" />
6   <input class="btn btn-primary"
7     type="submit" value="Save" />
8 </form>
```

Implementing a User service

For the same reasons that we are hiding away the complexity of the wunderground API, we'll also hide away our User api. This will enable us to use localstorage as well as communicate across our controllers about the user settings at any part of the app.

The UserService itself is straightforward and does not need to be configured in our app. Without the use of localstorage, our UserService will simply be:

```
1 // ...
2 .factory('UserService', function() {
3   var defaults = {
4     location: 'autoip'
5   };
6   var service = {
7     user: defaults
8   };
9
10  return service;
11 })
```

This service will hold on to our user object for the lifetime of the application. That is to say, while the browser window is open, the settings of the application will remain constant to the user's settings. However, if our user opens a new tab in chrome, these settings disappear, which is not ideal.

We can persist our settings across our app by using Chrome's sessionStorage capabilities. Luckily, this api is straightforward and simple.

We'll add two functions to the UserService:

- save
- restore

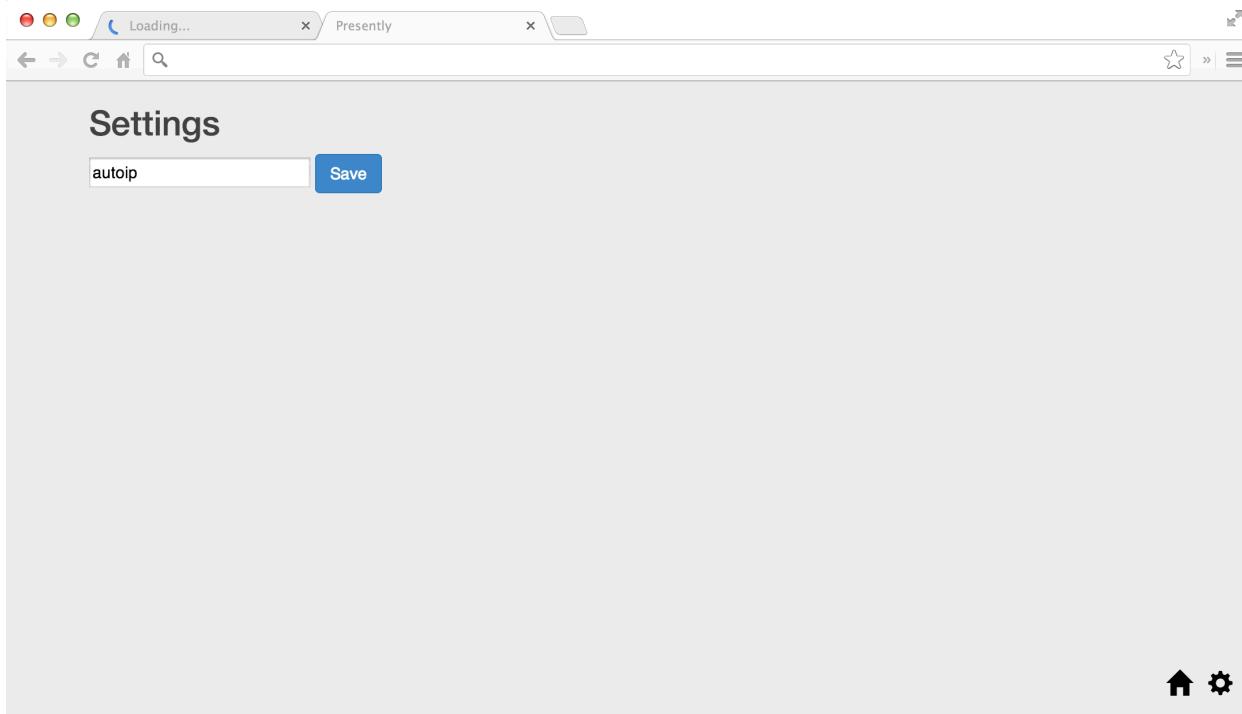
Even with these capabilities, the UserService has not grown

```
1 // ...
2 .factory('UserService', function() {
3   var defaults = {
4     location: 'autoip'
5   };
6
7   var service = {
8     user: {},
9     save: function() {
10       sessionStorage.presently =
11         angular.toJson(service.user);
12     },
13     restore: function() {
14       // Pull from sessionStorage
15       service.user =
16         angular.fromJson(sessionStorage.presently) || defaults
17
18       return service.user;
19     }
20   };
21   // Immediately call restore from the session storage
22   // so we have our user data available immediately
23   service.restore();
24   return service;
25 })
26 // ...
```

Now, we can inject this UserService across our Chrome app and have access to the same user data. Heading back to our SettingsCtrl, we can now set up a user object to define settings with the new service:

```
1 .controller('SettingsCtrl',  
2   function($scope, UserService) {  
3     $scope.user = UserService.user;  
4   });
```

If we refresh the browser, we'll now see that we have a default set for the user as 'autoip', which is the default we set up in the `UserService` definition.



Settings page

Now, we only need a way for our user to save their data into their session storage so we can use it across the app. In our `templates/settings.html`, we defined the form as having a `ng-submit="save()"` action, thus when our user submits the form, the `save()` function will be called.

Inside our `SettingsCtrl`, we'll implement the `save()` function that will call `save` on the `UserService` and persist the user's data into their `sessionStorage`.

```
1 .controller('SettingsCtrl',  
2   function($scope, UserService) {  
3     $scope.user = UserService.user;  
4  
5     $scope.save = function() {  
6       UserService.save();  
7     }  
8   });
```

Now, with the single input field bound to `user.location`, if we change the value and press save, our user's `sessionStorage` will be updated:

The screenshot shows the Google Chrome DevTools Elements tab. At the top, there is a settings page titled "Settings" with a text input field containing "CA/San_Francisco" and a blue "Save" button. Below the browser window, the DevTools interface is visible, showing the "Elements" tab selected. In the main pane, the "Session Storage" section is expanded, displaying a table with one item: a key "presently" with a value object {"location": "CA/San_Francisco"}. The status bar at the bottom right of the DevTools shows "4" notifications.

Key	Value
presently	{"location": "CA/San_Francisco"}

sessionStorage

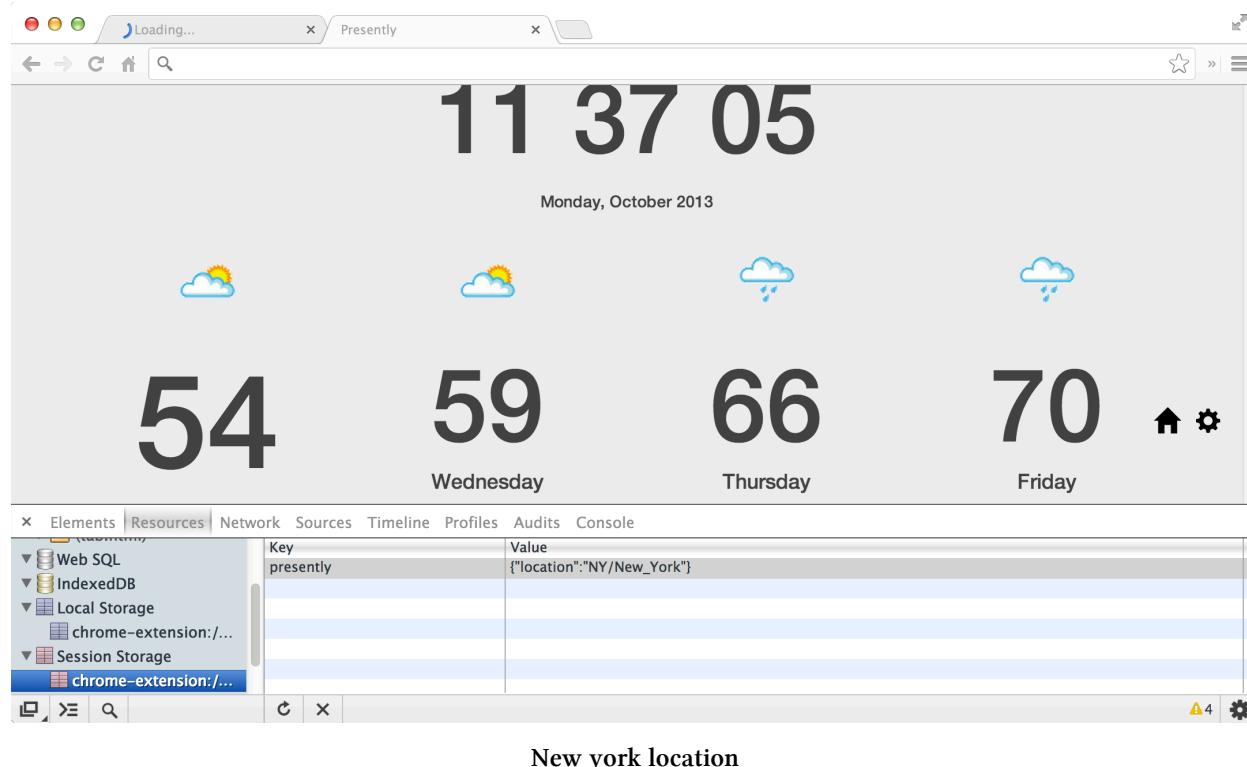
By using the `UserService` in our `HomeCtrl`, we can now remove the hardcoded value of “CA/San_Francisco” and replace it with our new `UserService` object’s location.

```

1 // ...
2 .controller('MainCtrl',
3   function($scope, $timeout, Weather, UserService) {
4     // ...
5     $scope.user = UserService.user;
6     Weather.getWeatherForecast($scope.user.location)
7     .then(function(data) {
8       $scope.weather.forecast = data;
9     });
10    // ...
11  })

```

As we can see, if we flip back and forth from the settings view and input “NY/New_York”, for instance we can see the weather changing based upon the location we place in the settings page.



City autofill/autocomplete

It's pretty inconvenient to need to type a city that conforms to the wunderground API formats (lat/long, city and state, country codes, etc). Luckily, the wunderground api also provides us an [autocomplete API¹⁴⁶](#).

¹⁴⁶<http://www.wunderground.com/weather/api/d/docs?d=autocomplete-api>

Instead of requiring our users to *know* the specific city format, we'll provide a list of them for our user's to select.



For simplicity and flexibility purposes, we're only going to create a raw javascript-based autocomplete, rather than use a plugin library, such as the typeahead.js or jQuery plugin libraries.

To do this, we'll create a directive that we'll place on the `<input>` element that will append a `` element with a list of suggested places.

```
1 .directive('autoFill', function($timeout) {
2   return {
3     restrict: 'EA',
4     scope: {
5       autoFill: '&',
6       ngModel: '='
7     },
8     compile: function(tEle, tAttrs) {
9       // Our compile function
10      return function(scope, ele, attrs, ctrl) {
11        // Our link function
12      }
13    }
14  }
15 })
```

As we will be creating a new element, we'll need to use the `compile` function, rather than just the link function and a template, since our `` element cannot be nested underneath an `<input>` element.

Without diving too deeply into how the `compile` function works, we're going to create a new element and we'll set up the bindings on our new element:

```
1 // ...
2 compile: function(tEle, tAttrs) {
3   var tplEl = angular.element('<div class="typeahead">' +
4     '<input type="text" autocomplete="off" />' +
5     '<ul id="autolist" ng-show="reslist">' +
6     '  <li ng-repeat="res in reslist" ' +
7     '    >{{res.name}}</li>' +
8     '</ul>' +
9     '</div>');
10  var input = tplEl.find('input');
```

```
11  input.attr('type', tAttrs.type);
12  input.attr('ng-model', tAttrs.ngModel);
13  tEle.replaceWith(tp1E1);
14
15  return function(scope, ele, attrs, ctrl) {
16    // ...
```

Inside of our link function, we'll bind on a keyup event and check that we have at least a minimum number of characters in our input field. Once there are a minimum number of characters, we'll run a function set by the use of the directive to fetch the auto-suggested values.

auto-complete api

Examining how we invoke this directive, we call it by passing a function to the `auto-fill` directive call as well as binding the location to the `user.location` value:

```
1 <input type="text"
2   ng-model="user.location"
3   auto-fill="fetchCities"
4   autocomplete="off"
5   placeholder="Location" />
```

In our Weather service, we'll create another function that specifically calls the `autocomplete` api and resolves a promise with a list of suggestions completions for a query term.

```
1  getWeatherForecast: function(city) {
2    // ...
3  },
4  getCityDetails: function(query) {
5    var d = $q.defer();
6    $http({
7      method: 'GET',
8      url: "http://autocomplete.wunderground.com/" +
9        'aq?query=' +
10       query
11    }).success(function(data) {
12      d.resolve(data.RESULTS);
13    }).error(function(err) {
14      d.reject(err);
15    });
16    return d.promise;
17 }
```

Back in our `SettingsCtrl`, we can simply reference this function as the function that retrieves the list of suggested values. Remember, we'll need to inject the `Weather` service in the controller to reference it.

```
1 .controller('SettingsCtrl',
2   function($scope, UserService, Weather) {
3     // ...
4     $scope.fetchCities = Weather.getCityDetails;
5   });

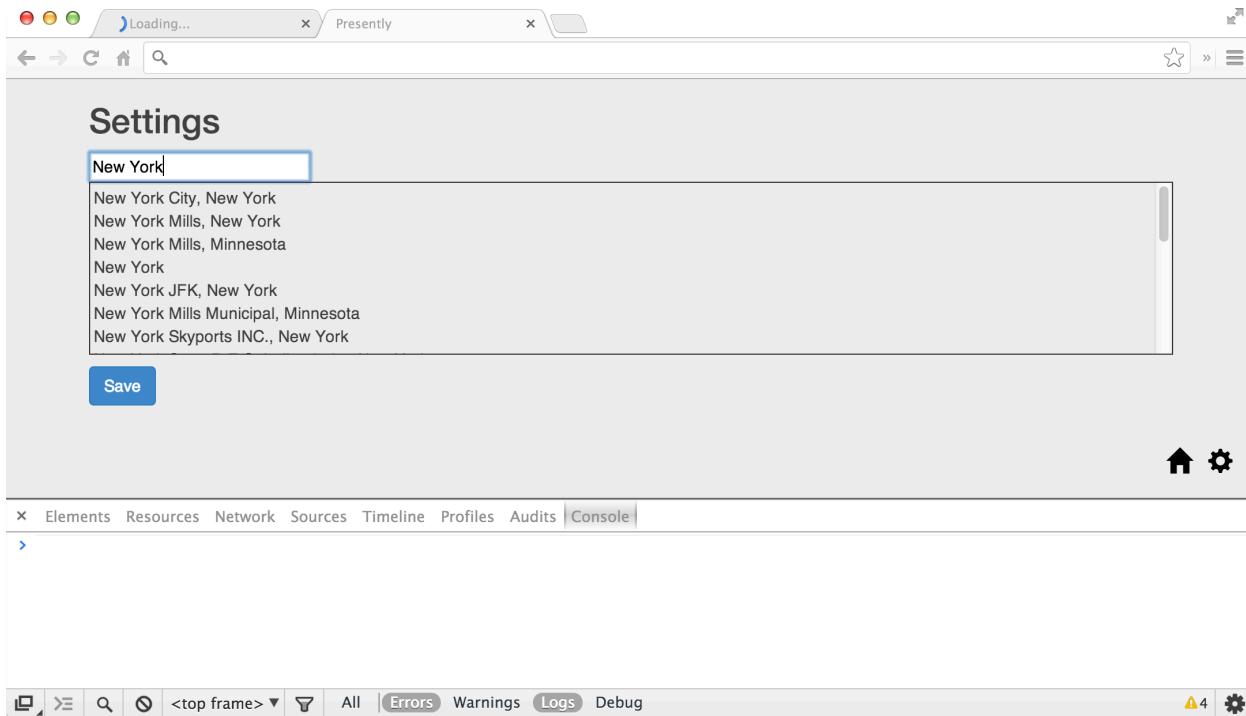
```

In the directive, we can now call this function that we'll reference when we create.

```
1 // ...
2 tEle.replaceWith(tp1El);
3 return function(scope, ele, attrs, ctrl) {
4   var minKeyCount = attrs.minKeyCount || 3,
5     timer,
6     input = ele.find('input');
7
8   input.bind('keyup', function(e) {
9     val = ele.val();
10    if (val.length < minKeyCount) {
11      if (timer) $timeout.cancel(timer);
12      scope.reslist = null;
13      return;
14    } else {
15      if (timer) $timeout.cancel(timer);
16      timer = $timeout(function() {
17        scope.autoFill()(val)
18        .then(function(data) {
19          if (data && data.length > 0) {
20            scope.reslist = data;
21            scope.ngModel = data[0].zmw;
22          }
23        });
24      }, 300);
25    }
26  });
27 // Hide the reslist on blur
28 input.bind('blur', function(e) {
29   scope.reslist = null;
30   scope.$digest();
```

```
31    });
32 }
```

We're using a *timeout* so that we only call the function once we are done typing. This is a simple way to prevent the function from being called repeatedly while we're really only interested in the first call to the suggestion API.



Sprinkling in timezone support

Finally, we also want our clock to update and reflect the new location that the user has set in their settings. Updating the clock to include timezone support is easy to add as we've implemented the most difficult part already through the autocomplete API.

First, we'll add one more attribute to our directive usage as `timezone`:

```
1 <input type="text"
2   ng-model="user.location"
3   timezone="user.timezone"
4   auto-fill="fetchCities"
5   autocomplete="off"
6   placeholder="Location" />
```

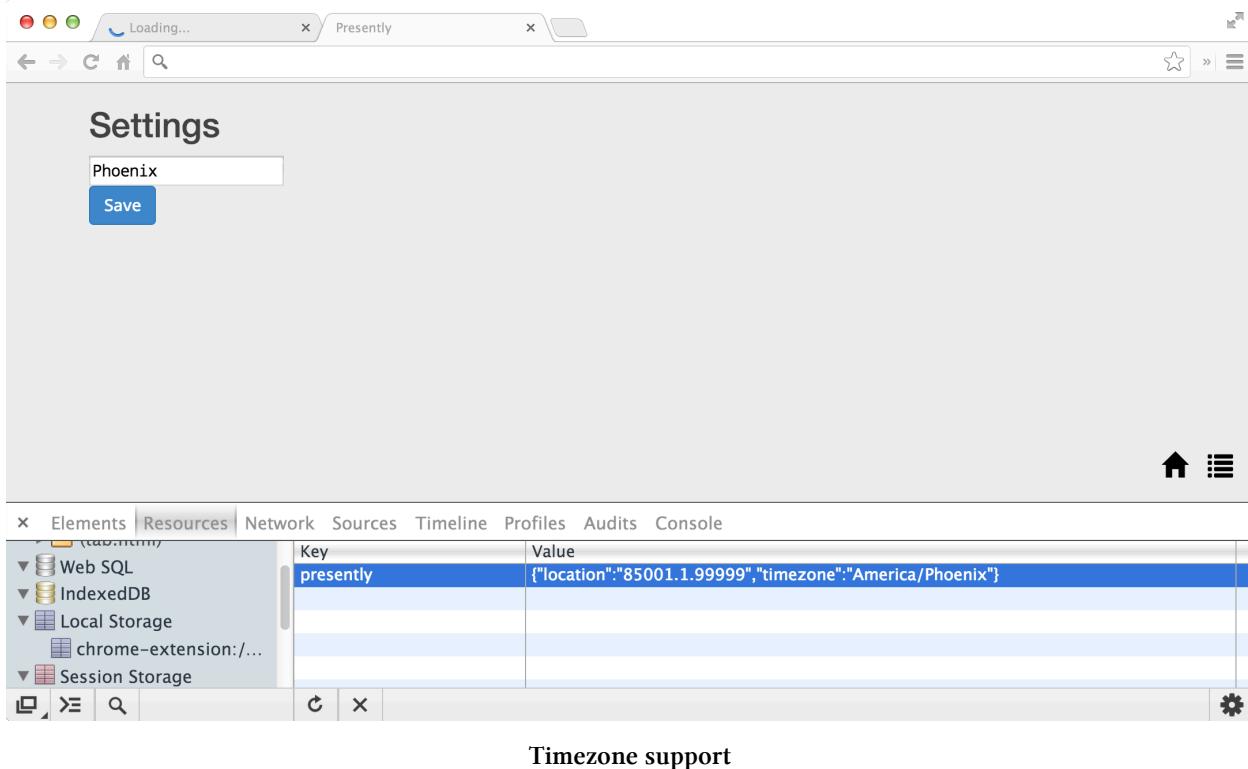
Next, we'll need to add the timezone attribute on to our generated `<input>` field in our directive's compile function:

```
1 // ...
2 input.attr('type', tAttrs.type);
3 input.attr('ng-model', tAttrs.ngModel);
4 input.attr('timezone', tAttrs.timezone);
5 tEle.replaceWith(tp1E1);
6 // ...
```

Last, but not least, we'll simply save the user's timezone when we save the topmost value for the user's location in the autocomplete link function:

```
1 // ...
2 scope.reslist = data;
3 scope.ngModel = data[0].zmw;
4 scope.timezone = data[0].tz;
5 // ...
```

Back in the browser, when we type our city we'll also save the timezone along with the new value of the city as well:



Timezone support

Finally, we'll need to update the time and date in our `MainCtrl` to take into account the new timezone.

Previously, matching timezone names to their GMT offsets was a difficult task. The Mozilla and Chrome teams have implemented the `toLocaleString` with the new `timeZone` argument that enables us to remap a date according to its timezone. Since we are writing a Chrome app, we can depend upon this function being available for us to use in our app.

Back in our `MainCtrl`, we'll create a new Date based off the saved timezone:

```

1 .controller('MainCtrl',
2   function($scope, $timeout, Weather, UserService) {
3     $scope.date = {};
4
5     var updateTime = function() {
6       $scope.date.tz = new Date(new Date().toLocaleString(
7         "en-US", {timeZone: $scope.user.timezone}
8       ));
9       $timeout(updateTime, 1000);
10    }
11  // ...

```

Now, instead of using the `$scope.date.raw` in our view, we'll switch over to using the `$scope.date.tz`. Now, the time will change along with the timezone.

The screenshot shows a weather application interface for Chicago. At the top, it displays the date and time: **03 06 31** and **Tuesday, October 2013**. Below this, there are four weather icons representing different conditions: sun, rain, lightning, and sun. Underneath each icon is a temperature reading: **59**, **63**, **64**, and **57**. The word "Chicago" is centered at the bottom of the main content area.

The bottom portion of the screenshot shows the **Resources** tab of the developer tools. It lists storage types: IndexedDB, Local Storage, and Session Storage. In the Local Storage section, there is one entry for "presently" with the value: {"location": "60290.1.99999", "timezone": "America/Chicago"}.

The screenshot shows a weather application interface for Hawaii. At the top, it displays the date and time: **10 06 45** and **Monday, October 2013**. Below this, there are four weather icons representing different conditions: rain. Underneath each icon is a temperature reading: **72**, **72**, **72**, and **72**. The word "Hawaii" is centered at the bottom of the main content area.

The bottom portion of the screenshot shows the **Resources** tab of the developer tools. It lists storage types: IndexedDB, Local Storage, and Session Storage. In the Local Storage section, there is one entry for "presently" with the value: {"location": "96718.1.99999", "timezone": "Pacific/Honolulu"}.

Debugging AngularJS

When we're building large angular apps, it's not uncommon that we'll run into head-scratching issues that are seemingly difficult to uncover.

Debugging from the DOM

Although not always necessary nor a first step, we can get access to the angular properties that are attached to any DOM element. We can use these properties to peek into how the data is flowing in our application.



We should never rely on fetching element properties from a DOM element during the life-cycle of an application. The techniques are presented as techniques for debugging purposes.

To fetch these properties from the DOM, we need to find the DOM element we're interested in. If we have the full jQuery library available, then we can use the jQuery selector syntax: `$("selector")`.

We don't need to rely on jQuery, however to target and fetch elements from the DOM. Instead, we can use the `document.querySelector()` method.



Note that the `document.querySelector()` is not available on all browsers and is *generally* good for non-complex element selections whereas [Sizzle¹⁴⁷](#) (the library jQuery uses) or [jQuery¹⁴⁸](#) support more complex selections.

We can retrieve the `$rootScope` from the DOM by selecting the element where the `ngApp` directive is placed and wrapping it in an angular element (using the `angular.element()` method).

With an angular element, we can call various methods to inspect our angular app from inside the DOM. To do this, we'll need to select the element from the DOM. Using purely JavaScript and angular, we can do this like so:

```
1 var rootEle = document.querySelector("html");
2 var ele = angular.element(rootEle);
```

With this element, we can fetch various parts of our application.

¹⁴⁷<http://sizzlejs.com/>

¹⁴⁸<http://jquery.com/>

scope()

We can fetch the \$scope from the element (or it's parent) from using the scope() method on the element:

```
1 var scope = ele.scope();
```

Using the scope, we can inspect any elements that are on it, such as custom variables that we set on the scope in our controllers as well as elements looking into it's \$id, it's \$parent object, the \$\$watchers that are set on it and even manually walk up the scope chain.

controller()

We can fetch the current element's controller (or it's parent) by using the controller() method:

```
1 var ctrl = ele.controller();
2 // or
3 var ctrl = ele.controller('ngModel');
```

injector()

We can fetch the current injector of the element (or the containing element) by using the injector() method:

```
1 var injector = ele.injector();
```

With this injector, we can then then instantiate any angular object inside our app, such as services, other controllers, or any other object inside of our angular app.

inheritedData()

We can fetch the data associated with an element's \$scope simply by using the inheritedData() method on the element:

```
1 ele.inheritedData();
```

This inheritedData() method is how angular finds data up the scope chain as it walks up the DOM until it's found a particular value or until the top-most parent has been reached.



If you're using Chrome, we can use a shortcut with the developer tools. Simply find the element you're interested in, right click on it in the browser, and select *inspect element*. The element itself is stored as the \$0 variable and we can fetch the angular-ized element by calling: angular.element(\$0).

debugger

Google's [Chrome¹⁴⁹](#) has it's own debugger tool to create a breakpoint in our code. The debugger statement will cause the browser to *freeze* during execution that allows us to examine the running code from inside the actual application and at the point of execution inside the browser.

To use the debugger, we can simply add the it inside the context of our application code:

```
1 angular.module('myApp')
2 .factory('SessionService', function($q, $http) {
3     var service = {
4         user_id: null,
5         getCurrentUser: function() {
6             debugger; // Set the debugger inside
7                 // this function
8             return service.user_id;
9         }
10    }
11
12    return service;
13 });


```

Inside this service, we'll call the `debugger;` method that will effectively *freeze* our application when it encounters this call.

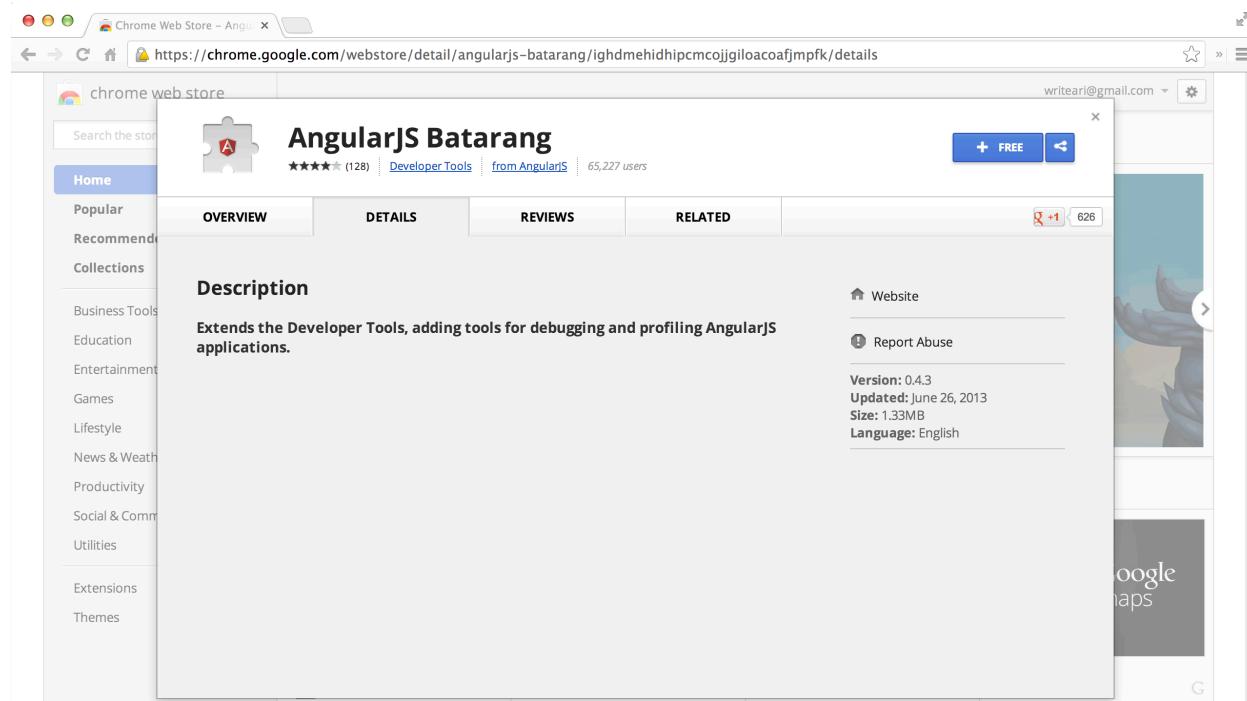
As long as the Chrome development tools are open in our browser, we can use `console.log()` and other javascript commands at the point where this application code executes.

When we're done debugging the application code, we'll need to make sure we **remove this line** as it will freeze the browser, even in production.

Angular Batarang

Angular Batarang is a Chrome extension developed by the Angular team at Google and integrates very nicely as a debugging tool for Angular apps.

¹⁴⁹<https://www.google.com/chrome>



Batarang chrome extension

Installing batarang

To install batarang, we'll simply need to download the application from the web store or from the github repo: <https://github.com/angular/angularjs-batarang>¹⁵⁰.

Once that's set, we can start it up by navigating to our developer tools and clicking *enable* to enable Batarang to start collecting debugging information about our page.

Batarang allows us to look at scopes, performance, dependencies and other key metrics in Angular apps.

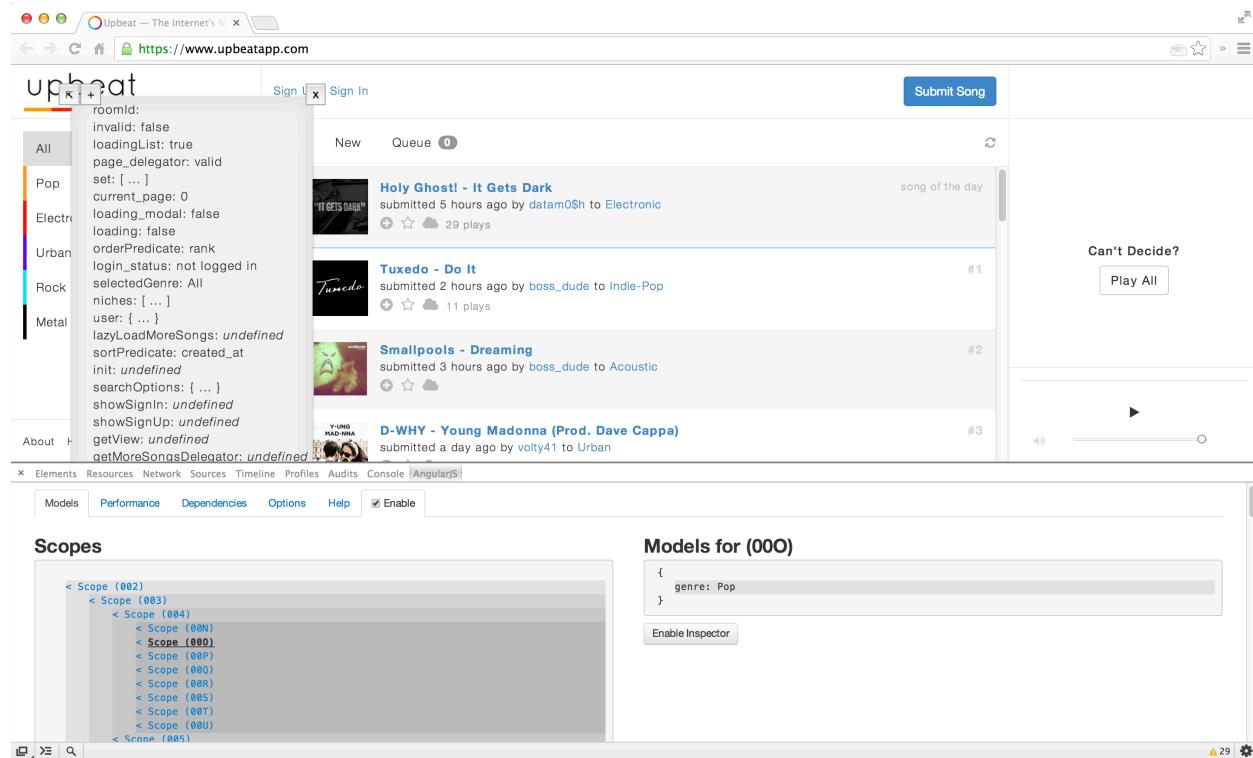
Inspecting the models

After we've started up Batarang, the page will reload and we'll notice that we have the panel that enables us to select different scopes in our page.

We can select a scope by clicking on the + button, hovering over and clicking on an element we're interested in.

Once we select a scope using the inspector, we can look at all the different properties on our scope element and their current values.

¹⁵⁰<https://github.com/angular/angularjs-batarang>

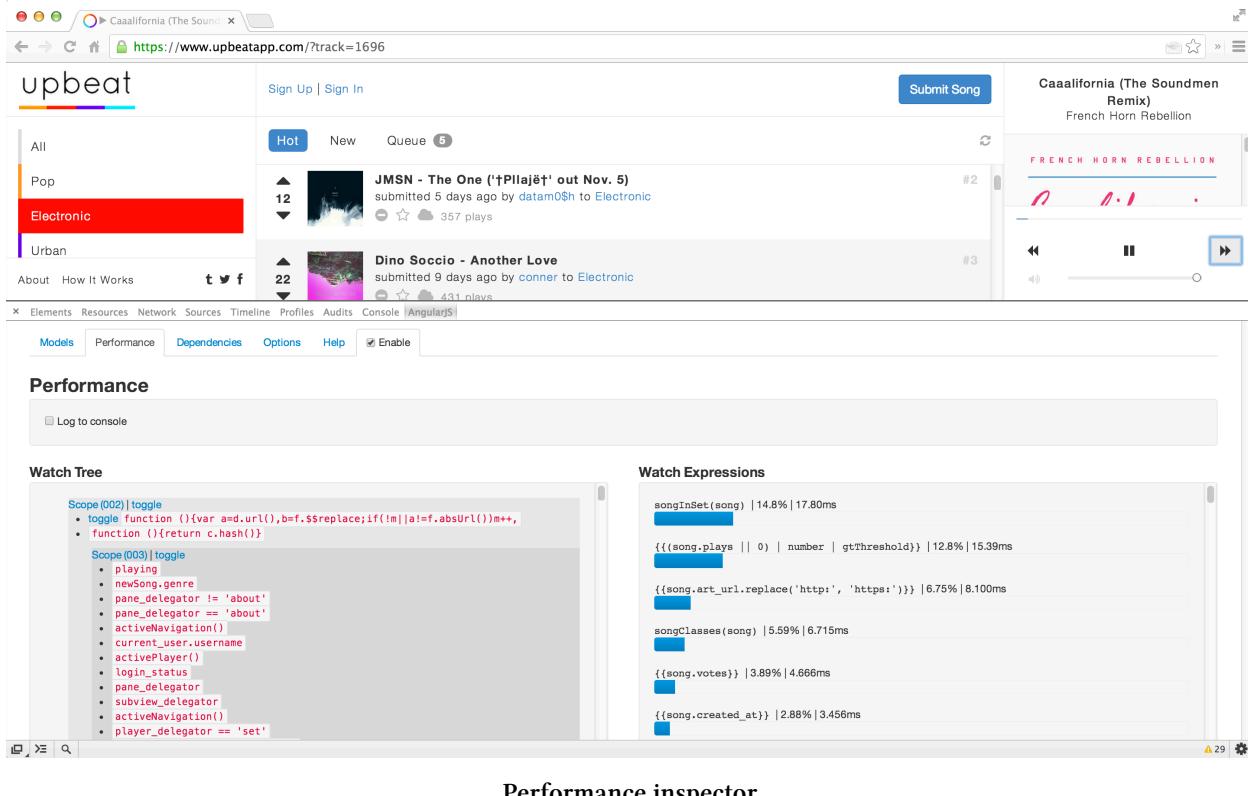


Model inspector

Inspecting performance

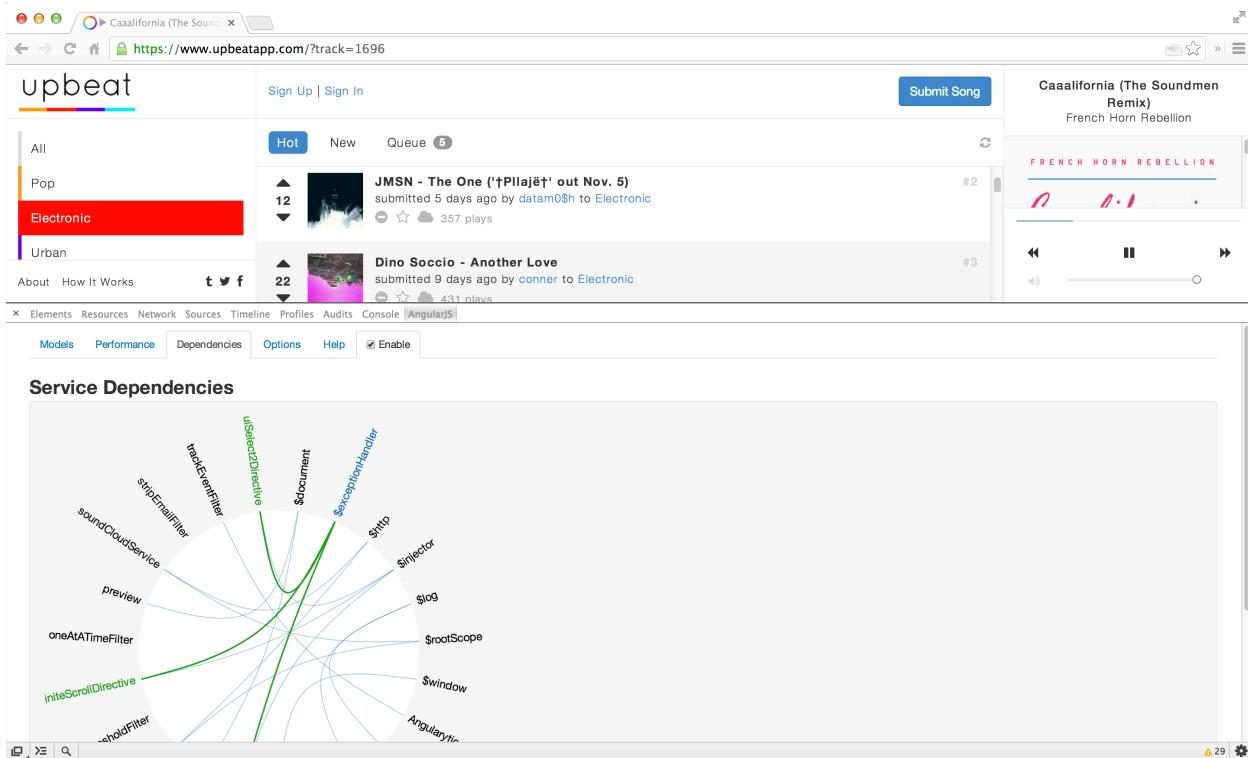
We can also peek into the performance of our application by using the performance section of Batarang.

In this panel, we get a peek into the watch list of the application at the different scopes as well as the amount of time that each expression takes, both in absolute time as well as percentage of the overall application time.



Inspecting the dependency graph

One very nice feature of the Batarang tool is its ability to visualize the dependency graph in-line. We can look at the dependencies of our application and view the different states at what they depend upon as well as track elements that aren't dependencies of the application at all.



Dependency graph

Visualizing the app

Batarang allows us to look deep into the application on the page itself. Using the Options panel, we can look at:

Applications The different applications that are on a single page (the `ngApp` directive uses)

bindings The bindings that are set in the view, where we use either `ng-bind` or `{ } elements`.

scopes The scopes in the view that we can target and inspect deeper.

The options panel also allows us view the angular version of the app as well as the CDN usage or not of Angular in the app.

The screenshot shows a web browser window for the [upbeat app](https://www.upbeatapp.com/?track=1696). The left sidebar lists genres: All, Pop, Electronic, Urban, Rock, and Metal. The main content area shows a 'Hot' feed with four song entries:

- JMSN - The One ('Plajet' out Nov. 5) submitted 5 days ago by datam0sh to Electronic. 857 plays.
- Dino Soccio - Another Love submitted 9 days ago by conner to Electronic. 431 plays.
- Yeasayer - Henrietta submitted 9 days ago by rogov to Electronic. 405 plays.
- French Horn Rebellion - Caaalifornia (The Soundmen Remix) submitted 11 days ago by conner to Electronic. 467 plays.

A detailed view of the 'French Horn Rebellion' track is shown on the right, featuring its cover art, title, artist, and play count. Below the main content are social sharing icons (Twitter, Facebook) and a 'Buy' button. At the bottom, there's a media player interface with volume and playback controls. The bottom navigation bar includes links for About, How It Works, and social media (t, v, f). The bottom right corner shows the Batarang extension interface with tabs for Models, Performance, Dependencies, Options, Help, and Enable, along with an 'Info' section displaying Angular version and CDN status.

Options

All in all, the Batarang tools gives us a lot of power when diving into how our Angular apps work in real-time.

Next steps

Now that you know AngularJS, we'll discuss what it will take and what tools you can use to move into a production environment.

jqLite and jQuery

Although angular encourages breaking away from relying on the [jQuery¹⁵¹](#) library, we can use it if we need to within our app by ensuring that we load it *before* the DOMContentLoaded event has been fired or we manually bootstrap our app.

Angular itself includes a compatible library called jqLite.

The `angular.element()` method that we've been using throughout this book returns a jqLite object, which is a *subset of the jQuery library that allows Angular to manipulate the DOM in a cross-browser compatible way*.

The jqLite library does not attempt to cover the entire jQuery library methods as it is intended on being lite and cover only those methods that are needed by Angular.

The library itself covers the following jQuery methods:

addClass() Adds the specified class(es) to the element.

after() Insert content to the end of the element

append() Insert content to the end of the element.

attr() Get or set the value of the attribute for the element.

bind() / on() Attach an event handler function for one or more events of the selected element.

children() Get the children of the element.

clone() Create a deep copy of element.

¹⁵¹<http://jquery.com/>

contents() Get the children of each element in the set including text and comment nodes.

css() Get or set the value of a style property for the element.

data() Store or return the value of arbitrary data associated with the element.

eq() Get the element at the specified index.

find() Get the descendants of the element filtered by tagname only.

hasClass() Determine if the element itself is assigned a given class.

html() Get or set the HTML contents of the element.

next() Get the immediately following sibling of the element.

off() / unbind() Remove an event handler by name.

parent() Get the parent of the element.

prepend() Insert content to the beginning of the element.

prop() Get or set the value of a property for the element.

ready() Specify a function to execute when the DOM is fully loaded.

remove() Remove the element from the DOM.

removeAttr() Remove an attribute from the element.

removeClass() Remove a single, multiple, or all classes from the element.

removeData() Remove the previously stored data from the element.

replaceWith() Replace the element with the provided new content.

text() Get or set the combined text contents of the element.

toggleClass() Add or remove one or multiple classes from the element.

triggerHandler() Execute all handlers attached to an element for an event.

val() Get or set the current value of the element.

wrap() Wrap an HTML structure around the element.

Essential tools to know about

The AngularJS community is fantastic and there are some great tools that have been written to support AngularJS development. We'll range from discussing build tools and frameworks, to live interaction tools.

Grunt

Grunt¹⁵² is a pure javascript task runner. It will save you tons of time in developing javascript applications, both server-side and client-side. It will make repetitive tasks disappear and handle running them for you automatically.

The javascript community has jumped all over the Grunt tool and created hundreds of plugins. If a plugin you need or want has not been developed, the Grunt tool has made it very easy to create your own.

Installation

First, we'll have to make sure you have NodeJS¹⁵³ installed. NodeJS is a *platform built on Chrome's JavaScript runtime* and allows you to write JavaScript as a server-side language.

To install grunt, we'll use the built-in *npm* tool that comes with NodeJS:

```
1 $ npm install -g grunt-cli
```

¹⁵²<http://gruntjs.com/>

¹⁵³<http://nodejs.org/>



Passing the `-g` flag will make the `grunt` command available in any directory on your computer.

With grunt installed, you'll also need to have a *Gruntfile* alongside your app to configure how and what grunt runs. In order to do anything useful with Grunt, let's create a `Gruntfile.js` in our project.

First things first, we'll have to create a `package.json` file that will tell Node what to install as dependencies.



Just as AngularJS handles dependencies, NodeJS has a clever method of dependency management. The `package.json` file will be your friend as you write more NodeJS apps.

To make the default `package.json` file, you can either run a generator or copy and paste from the default `package.json`. Since the `npm init` command is built-in, let's use that:

```
1 $ npm init
```

This command will ask you a series of questions, like what is the *name* of your new app, what *version*, and a few more. You can use all of the defaults it will set for you, although it's probably a good idea to set the name of your app.

Once this command completes, it'll create a `package.json` file that looks something like:

```
1 {
2   "name": "myapp",
3   "version": "0.0.0",
4   "description": "Your myapp description",
5   "main": "index.js",
6   "scripts": {
7     "test": "echo \\\"Error: no test specified\\\" && exit 1"
8   },
9   "author": "Your name",
10  "license": "MIT"
11 }
```

Now, we can install the basic grunt command in our `package.json` file by using the `npm` command again:

```
1 $ npm install grunt --save-dev
```



The `--save-dev` flag will save the grunt as a dependency for development. If you want to save a dependency for runtime, you can use the `--save` flag.

A common use of grunt is to minify our javascript files so we send the smallest file to the browser. This is particularly useful so your app loads the quickest it can; especially on mobile devices.

We'll install the uglify plugin to handle this for us.

```
1 $ npm install grunt-contrib-uglify --save-dev
```

Great, now we can configure grunt using the Gruntfile. To configure Grunt, load your `Gruntfile.js` in your text editor and add the following:

```
1 module.exports = function(grunt) {
2   // Configuration
3   grunt.initConfig({
4     pkg: grunt.file.readJSON('package.json')
5   });
6   // Load plugins
7   // Default task(s).
8
9};
```

To setup a configuration, we'll need to tell grunt to load the Npm tasks of all the plugins we'll want to use. Since we're loading the `uglify` tasks, we'll tell grunt to load our `grunt-contrib-uglify` plugin tasks:

```
1 grunt.loadNpmTasks('grunt-contrib-uglify');
```

To configure uglify, we can place a `configure` block inside of the `initConfig` object with the key of `uglify` (this is dependent upon the plugin you'll use). In this case, we'll make a minimal update to the configuration where we'll only set the `src` and the `dest` locations.

```
1 grunt.initConfig({
2   pkg: grunt.file.readJSON('package.json'),
3   uglify: {
4     build: {
5       src: 'src/<%= pkg.name %>.js',
6       dest: 'build/<%= pkg.name %>.min.js'
7     }
8   }
9 });
```

All of the available options for block of configuration code for the `grunt-contrib-uglify` module can be found in the README for the project (available [here¹⁵⁴](#)).



Note, that when using grunt modules, most often the configuration documentation will be available in the project's README file or otherwise will point to the available configuration options.

With that set, grunt will look for the javascript file named the whatever our `package.json` name is set to in the `src/` directory. It will then run the `uglify` task on this file.

To actually tell Grunt to run the task, you can run the task `uglify`:

```
1 $ grunt uglify
```

You can also configure grunt to run multiple tasks in one go by declaring a task with multiple subtasks:

```
1 grunt.initConfig({
2   // config
3 });
4 grunt.registerTask('default', ['uglify']);
```

Now you can run `grunt default` and all of the tasks you've defined in the array will run. The `default` task has special meaning in Grunt. With the `default` task configured like so, you can just run the `grunt` command and all of those tasks will run.

You might be wondering why does this even matter. This example, we've only setup one task to run, but if you are using coffeescript, want to package all of your angular templates into a single file, package your less css files, etc. etc. grunt will handle that for you.

Finally, one of the most useful features of grunt is it's ability to *watch* the filesystem for file changes and execute commands on the file changes.

To setup watch, we'll do the same two steps as we did above. First, install the `grunt-contrib-watch` npm package:

¹⁵⁴<https://github.com/gruntjs/grunt-contrib-uglify>

```
1 $ npm install grunt-contrib-watch --save-dev
```

Next, setup a config block in the `initConfig` object:

```
1 grunt.initConfig({
2   pkg: grunt.file.readJSON('package.json'),
3   // 
4   watch: {
5     js: {
6       files: 'src/**/*.js',
7       tasks: ['uglify']
8     }
9   }
10});
```

Now we can run `grunt watch` and grunt will start watching all of the javascript files in your `src/` directory. When any of them change, it will run the `uglify` task.

grunt-angular-templates

By default, angular will fetch templates over XHR when it cannot find them locally in its `$templateCache`. When the XHR request is slow or our template is large, this can seriously impede the experience of our app for our users.

One way we can avoid this delay is by “faking” the `$templateCache` in thinking that it has already been filled up, so angular doesn’t have to load the templates from afar. We can do this manually in JavaScript like so:

```
1 angular.module('myApp', [])
2 .run(function($templateCache) {
3   $templateCache.put('home.html',
4     'This is the home template');
5 });
```

Now, when angular needs to fetch the template named ‘`home.html`’, it will find it in the `$templateCache` and not need to fetch it from the server.

This is quite cumbersome to do manually if we want to package our app for our servers. Luckily, the `grunt-angular-templates` grunt task will do this for us.

Installation

First, we need to install the grunt task. We’ll do this with `npm` as follows:

```
1 $ npm install --save-dev grunt-angular-templates
```



We'll use the `--save-dev` task to store the grunt task in our package.json file and is good practice to do so that the next person has all the dependencies that we're using. If we're not using a package.json file, then we can ignore this flag, but it won't do any harm to keep it in also. `npm` will simply output a message warning us that we're not using a package.json.

Next, we'll need to reference this new task in our `Gruntfile.js` file, like so:

```
1 grunt.loadTasks('grunt-angular-templates');
```

Now, we can safely use this task in our Grunt tasks.

Usage

The task itself will compile a JavaScript file that we will need to load inside of our `index.html`. For instance, if we load tell the task to generate the `templates.js` file, we'll need to load it inside of our `index.html`:

```
1 <script src="template.js"></script>
```

First off, like any other Grunt task we'll need to configure it. The configuration template key is: `ngtemplates`. Inside of this `ngtemplates` configuration block, we'll set a subtask that will become the name of the angular module we're loading.

For example:

```
1 ngtemplates: {
2   myApp: {}
3 }
```

This will generate the output of our `template.js` file as:

```
1 angular.module('myApp')
2   .run(['$templateCache', function($templateCache) {
3     $templateCache.put('home.html', ...);
4   }])
```

Notice that the name of the subtask `myApp` is the same as the angular module that the `$templateCache` is referenced.

Inside of this subtask is where we'll set our options.

Available options

bootstrap

By default, angular-grunt-templates wraps the `function($templateCache) {}` inside of the `angular.module('myApp').run(['$templateCache', ____]);` We can change this if we're using CommonJS or RequireJS with the bootstrap option:

```
1 // ...
2 bootstrap: function(module, script) {
3   return 'module.exports[module]= ' + script + ';';
4 }
```

concat

This is the name of the concat target to append our compiled template path.

htmlmin

Just as we can minimize our css and javascript files, we can also minimize our html using a tool called (unsurprisingly) `htmlmin`. `grunt-angular-templates` plays nicely with `htmlmin` and even allows us to minimize the html inside of our templates as well.

We can set options for `htmlmin` inside of our configuration, like so:

```
1 ngtemplates: {
2   myApp: {
3     options: {
4       htmlmin: {
5         collapseBooleanAttributes: true,
6         collapseWhitespace: true,
7         removeAttributeQuotes: true,
8         removeEmptyAttributes: true,
9         removeRedundantAttributes: true
10        removeScriptTypeAttributes: true,
11        removeStyleLinkTypeAttributes: true
12      }
13    }
14  }
15 }
```

module

The name of the `angular.module` that the template cache will be registered with will be the name of the subtask that we are working with the options *unless* we set it in the options as `module`.

```
1 ngtemplates: {
2   myApp: {
3     options: {
4       module: 'myBestApp'
5     }
6   }
7 }
```

This will result in the templates being set as:

```
1 angular.module('myBestApp')
2   .run(['$templateCache',
3   function($templateCache) {
4     // ...
5   }
6 }
```

prefix

We can set a prefix for all of our template URLs, so for instance if we want to use absolute URLs where our templates are accessed from an absolute location in a different directory, we'd set our prefix to look like:

```
1 ngtemplates: {
2   app: {
3     options: {
4       prefix: '/public'
5     }
6   }
7 }
```

source

The source option can be set to a function that gets called before the rest of the templates are compiled, after the source has been minified so that we can customize the template source output.

The function gets called with the options:

- source - the minified template source
- path - the path to the template file
- options - the task options object

standalone

This boolean options flag tells the grunt task if the templates are a part of an existing module, such as `myApp` or if they are standalone. Mostly, this option should be set to false (as it is by default).

url

Setting the `url` option will override the template's `$templateCache` URL. Mostly, this option is here for special circumstances as we'll set the `cwd` and the `src`, which will make the templates available both through XHR and through the `$templateCache`.

Usage

The authors of `grunt-angular-templates` have given us a lot of options for how we can use this task.

concat

The *easiest* way to use the task is inside of the `concat` task. This will place the responsibility of the location of the task into the minified `concat` task.

```
1 concat: {
2   app: {
3     src: ['*.js', '<%= ngtemplates.app.dest %>'],
4     dest: [ 'app.js' ]
5   }
6 }
```

Now our templates will be attached at the end of our `app.js` file.

usemin

When using `grunt-usemin`, a task that minifies and combines javascript inline request files. For instance:

```
1 <!-- build:js module.js -->
2 <script src="scripts/app.js"></script>
3 <script src="scripts/controllers.js"></script>
4 <!-- endbuild -->
```

The name of the file that will be minified is `module.js`. We can use this as a target to attach our templates.

```
1 ngtemplates: {
2   app: {
3     src: 'templates/*.html',
4     dest: 'template.js',
5     options: {
6       concat: 'module.js'
7     }
8   }
9 }
```

dest

Lastly, we can normally generate the `templates.js` just by specifying a destination, rather than appending it to another file by simply assigning a `dest:` key.

```
1 ngtemplates: {
2   app: {
3     src: 'templates/*.html',
4     dest: 'template.js'
5   }
6 }
```

Lineman

The Lineman tool is a build-tool that allows you to focus on building fat-client (or client-side) web apps. It mixes an incredible amount of functionality to make client-side webapp development fun and easy.

Lineman is built and maintained by the community to keep the front-end webapp development *productive* in a maintainable, manageable manner.

We've been developing our client-side application in a single `index.html` file that we've been loading through our browser for most of this book. Lineman takes a different approach and serves the application through a local server.

By serving files through a local server, Lineman can offer unique functionality we can't get with static files. It will:

- Compile and serve [Coffeescript¹⁵⁵](#) files as javascript upon saving a file
- Run [Less¹⁵⁶](#) and [Sass¹⁵⁷](#) preprocessors and serve the generated CSS

¹⁵⁵<http://coffeescript.org/>

¹⁵⁶<http://lesscss.org/>

¹⁵⁷<http://sass-lang.com/>

- Provides backend stubbing tools so you can develop *with or without* a backend server
- Pre-compile your javascript templates
- Proxy XHR requests to your backend server
- Make testing incredibly easy and fun

Lineman explicitly does not deal with any backend webserver (although it does provide a way for us to stub backend calls as we'll see). Its focus is on building AngularJS apps that can be compiled, minified, and deployed as a static webapp.

To use lineman, we'll need to make sure we have [nodejs¹⁵⁸](#) installed which comes prepackaged with the npm tool. To install *lineman* itself, we'll install it with npm and we'll install it globally.

```
1 $ npm install -g lineman
```

Although we'll be using lineman itself to run our project, we won't use the packaged generator. Instead we'll use the AngularJS template created by David Mosher.

```
1 $ git clone https://github.com/davemo/lineman-angular-template my-app
```

Once this has been cloned (using git), we'll use npm again to install the dependencies that lineman needs to operate:

```
1 $ cd my-app && npm install -d
```

After the dependencies have been installed, we can start working on our app. We'll work with the workflow of editing our app while running both the tests as well as the server.

To run the app, start the lineman tool in the `my-app` directory.

```
1 $ lineman run
```

Now, we'll have our app running in the browser at `http://localhost:8000`.

¹⁵⁸<http://nodejs.org>



Running lineman

As you can see, the angular template has a few templates with it. You'll also notice the directory structure has a bunch of directories:

- **app** - contains the app files
 - **css** - our css files (less or css files)
 - **img** - our img files
 - **js** - our angular app
 - **pages** - the html templates to be compiled
 - **templates** - the angular templates
- **config** - the lineman-specific configuration files
- **doc** - a directory for documentation for the app
- **dist** - a generated directory where our production app is built
- **generated** - a generated directory for the `lineman run` app
- **spec** - all specs that are not end-to-end specs
- **spec-e2e** - our *protractor* specs live here
- **tasks** - any custom lineman tasks should go here
- **vendor** - contains any vendor css, javascript, and image files
- **Gruntfile.js** - the gruntfile that powers lineman
- **package.json** - the app customization, defines dependencies and other metadata

Lineman provides an efficient structure for editing webapps quickly and confidently.

Bower

Bower is a package manager for front-end files on the web. Similar to how `npm` is a package manager for node modules, which allows developers to write *shareable* modules for the server bower offers similar functionality for web components.

It offers a solution to the dependency problem through a generic, un-opinionated, and easy-to-use interface. It runs over git and is package agnostic. It also supports any type of transport, like requireJS, AMD, and others.

Installation

Installation is simple: we'll use the `npm` package manager to install bower:

```
1 $ npm install -g bower
```

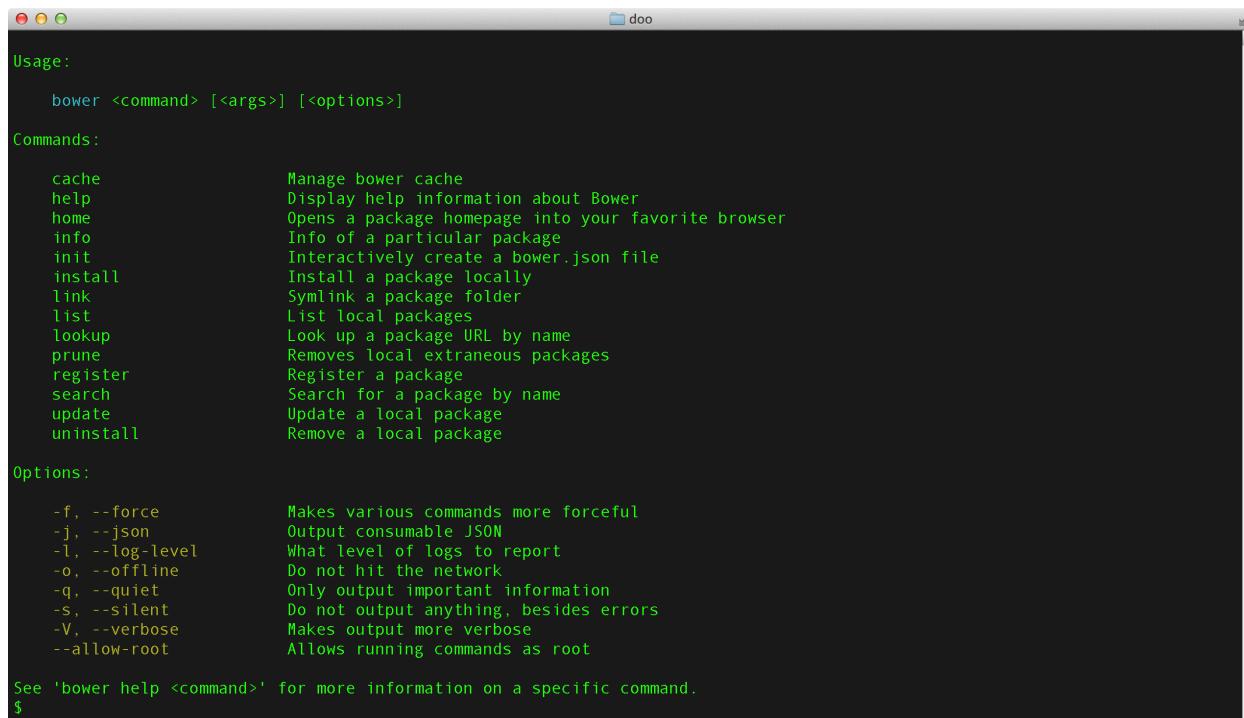


bower depends upon git, node, and npm.

From here, we can verify that it's working by typing the `help` command:

```
1 $ bower help
```

If output is displayed to the screen, then we are good to go.



The screenshot shows a terminal window titled 'doo' with the Bower help screen displayed. The output includes sections for Usage, Commands, and Options, along with detailed descriptions for each command and option. At the bottom, it says 'See 'bower help <command>' for more information on a specific command.'

```
Usage:
  bower <command> [<args>] [<options>]

Commands:
  cache          Manage bower cache
  help           Display help information about Bower
  home           Opens a package homepage into your favorite browser
  info            Info of a particular package
  init            Interactively create a bower.json file
  install         Install a package locally
  link            Symlink a package folder
  list             List local packages
  lookup          Look up a package URL by name
  prune           Removes local extraneous packages
  register        Register a package
  search          Search for a package by name
  update          Update a local package
  uninstall       Remove a local package

Options:
  -f, --force      Makes various commands more forceful
  -j, --json       Output consumable JSON
  -l, --log-level  What level of logs to report
  -o, --offline    Do not hit the network
  -q, --quiet      Only output important information
  -s, --silent     Do not output anything, besides errors
  -V, --verbose    Makes output more verbose
  --allow-root     Allows running commands as root

See 'bower help <command>' for more information on a specific command.
$
```

Bower help screen

Bower overview



Although we'll cover a brief overview, we encourage more exploration at the bower homepage: [bower.io¹⁵⁹](http://bower.io).

With our webapp, we're likely going to want to share the source with other developers or deploy to other development machines. Similar to the `package.json` file for npm, we can use a `bower.json` file to store our front-end dependencies.

To get started with a `bower.json`, we can use the `init` command provided by bower. We should execute this in the root of our project directory:

```
1 $ bower init
```

This command will launch a setup wizard that will ask us a few questions about our new package. When it's done, it will generate a new `bower.json` file in our current directory.

¹⁵⁹<http://bower.io/>

Configuring bower

Bower comes with sane defaults, but it is highly configuration. We can configure what directory packages are installed, which registry to use to install components.



Bower has great documentation on configuration available [here¹⁶⁰](#). We recommend you check them out for more detailed configuration.

Although in-depth bower configuration is outside of the scope of this chapter, we'll look at the two most commonly modified configuration items (based upon our own experience).

To configure bower, we can edit the `.bowerrc` file, pass config arguments, or set environment variables. The `.bowerrc` file can be located in several places:

- the current working directory of the project
- in any subfolder in the directory tree
- in the current user's home folder
- in the global bower folder

The `.bowerrc` file contains a JSON object for configuration. For example, to change the color configuration, the `.bowerrc` file would contain:

```
1 {
2   "color": false
3 }
```

For the purposes of simplicity, we like to keep the `.bowerrc` file in the root of the project directory. If it doesn't already exist, create it in the root of the project directory:

```
1 $ echo "{}" > .bowerrc
```

cwd The `cwd` configuration variable is the directory from which bower should run. All other paths should relate directly to this directory.

```
1 {
2   "cwd": "app"
3 }
```

directory The `directory` configuration variable is the path in which installed components should be saved. This defaults to `bower_components`. Depending on how we are creating an app, we'll change this to suit a different directory structure:

¹⁶⁰<https://docs.google.com/document/d/1APq7oA9tNao1UYWyOm8dKqlRP2blVkROYLZ2fLJjtWc/edit#heading=h.4pzytc1f9j8k>

```
1 {
2   "directory": "app/components"
3 }
```

Searching for packages

To find a package for installation, bower includes a search command to search through the index of it's registry:

```
1 ## Searching for bootstrap-sass
2 $ bower search bootstrap-sass
```

Installing packages

Installing packages is equally as easy. If we have an existing bower.json file, we can simply run the install command. This will pull-down and install the front-end dependencies in the bower directory.

```
1 $ bower install
```

We can install a package locally by explicitly calling install on the file. It's possible to install a specific version of the package and even set an *alias* for the package install.

```
1 # Install a local or
2 # default remote version of a package
3 $ bower install <package>
4 # Install a specific version of a package
5 $ bower install <package>#<version>
6 # Alias install a package
7 $ bower install name=<package>#<version>
8 # For instance
9 $ bower install bootstrap=bootstrap-sass
```

The bower.json file stores several types of dependencies, either dependencies needed by the runtime (such as angular or jquery) or dependencies needed in the development process (like karma or bootstrap-sass).

```
1 # Install a runtime dependency
2 $ bower install angular-route --save
3 # Install a dev dependency
4 $ bower install bootstrap-sass --save-dev
```

If we dump out the contents of our `bower.json`, we'll see that it is updated with our new dependencies:

```
1 $ cat bower.json
2 {
3   "name": "myApp",
4   "version": "0.0.1",
5   "authors": [
6     "Ari Lerner <ari@fullstack.io>"
7   ],
8   "license": "MIT",
9   "dependencies": {
10     "angular-route": "~1.2.0-rc.2"
11   },
12   "devDependencies": {
13     "bootstrap-sass": "~3.0.0"
14   }
15 }
```

Using packages

Now that our packages are installed, we can include the packages just like any other script in our local directory: using a `script` tag in the HTML source.

```
1 <script
2   src="/bower_components/angular/angular.js">
3 </script>
```

Removing packages

It's also possible to remove packages through bower. We can either manually delete the files in our bower directory, or we can run the `uninstall` command.

The `uninstall` command allows us to use the `--save` and `--save-dev` flags to reflect the changes in the `bower.json` file.

```
1 # Remove a dependency
2 $ bower uninstall --save-dev angular-route
3 # Remove a devDependency
4 $ bower uninstall --save-dev bootstrap-sass
```

Yeoman

Yeoman¹⁶¹ is a collection of the previous tools we've been discussing in this chapter:

- Yeoman
- Grunt
- Bower

Yeoman itself is a scaffolding tool and helps us build new applications by setting up our grunt configuration, building our application workspace, and managing our workflow, regardless of the type of application we are building.

There are just under 300 generators written by the community available as of the time of this writing to set up many projects of all sorts of different types from [Angular¹⁶²](#) sites, to [Backbone.js¹⁶³](#) and even [Python flask¹⁶⁴](#) projects.

Grunt is set up as the build tool and bower is set up for handling dependency management.

Installation

Installation for yeoman is simple. First, we'll need to make sure we have [node.js¹⁶⁵](#) and [git¹⁶⁶](#) installed. Some generators require [ruby¹⁶⁷](#) and [compass¹⁶⁸](#) to be installed.

Once we have the dependencies, we can install yeoman itself using npm:

```
1 $ npm install -g yo
```

Installing yeoman will install *Grunt* and *Bower* automatically.

Next, in order to use it, we'll need to install a generator as yeoman is useless without a generator.

We'll install the angular generator:

¹⁶¹<http://yeoman.io/>

¹⁶²<https://github.com/yeoman/generator-angular>

¹⁶³<https://github.com/yeoman/generator-backbone>

¹⁶⁴<https://github.com/romainberger/yeoman-flask>

¹⁶⁵<http://nodejs.org>

¹⁶⁶<http://git-scm.com/>

¹⁶⁷<https://www.ruby-lang.org/>

¹⁶⁸<http://compass-style.org/>

```
1 $ npm install -g generator-angular
```



To search all the available community generators, we can check out the web interface at <http://yeoman.io/community-generators.html>¹⁶⁹.

Usage

Using the yeoman workflow is easy as well. First and foremost, we'll want to create a directory to work in. Yeoman does not create a working directory for us, rather it assumes the directory we're working in is the directory that houses our app.

```
1 $ mkdir myapp && cd $_
```

Inside of this directory, we'll run the yeoman generator which will scaffold our project. In this example, we're using the angular generator `generator-angular`:

```
1 $ yo angular
```

A screenshot of a terminal window titled "myapp — node — Solarized Dark xterm-256color — 80x24". The window shows the command line history:

```
$ mkdir myapp && cd $_  
$ yo angular  
[?] Would you like to include Twitter Bootstrap? (Y/n) [
```

The terminal has a dark background with light-colored text and uses color-coded syntax highlighting for commands and file paths.

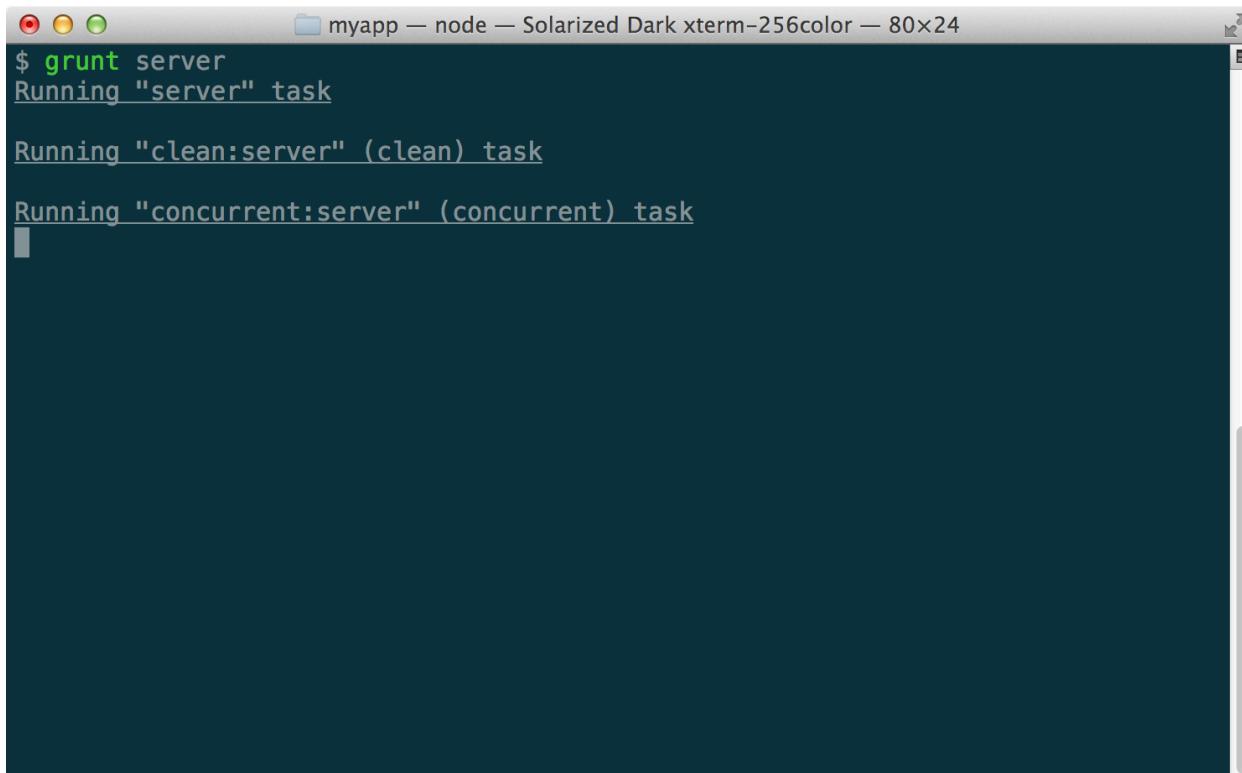
Yeoman install

¹⁶⁹<http://yeoman.io/community-generators.html>

We'll be asked a few questions and it will then create our application. In these steps, it will call `npm install` as well as `bower install` for us to ensure we have all the dependencies we expect so we can get developing immediately.

We'll use the `grunt` command to kick off our development process.

```
1 $ grunt server
```

A screenshot of a terminal window titled "myapp — node — Solarized Dark xterm-256color — 80x24". The window shows the command "\$ grunt server" being run, followed by the output: "Running \"server\" task", "Running \"clean:server\" (clean) task", and "Running \"concurrent:server\" (concurrent) task".

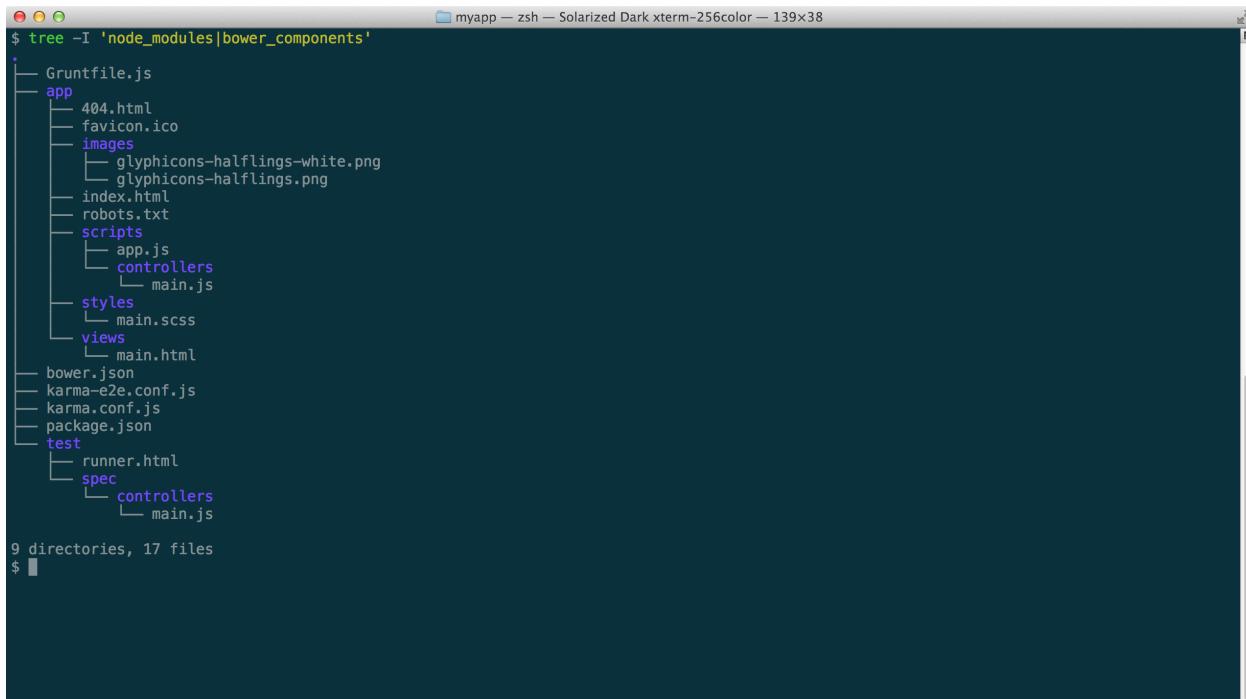
```
$ grunt server
Running "server" task
Running "clean:server" (clean) task
Running "concurrent:server" (concurrent) task
```

Running yeoman

The `grunt server` command will start a local server that serves our app locally. It uses [livereload](#)¹⁷⁰ to automatically reload the browser when we save a file in our workspace.

The directory that is built for us has an opinionated structure that enforces an easily extensible design for our angular apps.

¹⁷⁰<http://livereload.com/>



```
$ tree -I 'node_modules|bower_components'  
.  
+-- Gruntfile.js  
+-- app  
|   +-- 404.html  
|   +-- favicon.ico  
|   +-- images  
|   |   +-- glyphicons-halflings-white.png  
|   |   +-- glyphicons-halflings.png  
|   +-- index.html  
|   +-- robots.txt  
|   +-- scripts  
|   |   +-- app.js  
|   |   +-- controllers  
|   |   |   +-- main.js  
|   +-- styles  
|   |   +-- main.scss  
|   +-- views  
|   |   +-- main.html  
+-- bower.json  
+-- karma-e2e.conf.js  
+-- karma.conf.js  
+-- package.json  
+-- test  
|   +-- runner.html  
|   +-- spec  
|   |   +-- controllers  
|   |   |   +-- main.js  
9 directories, 17 files
```

Yeoman generated structure

Yeoman creates a directory structure that builds both an `app/` and a `test/` directory. Inside the `app`, we'll build our angular app as well as house our views, our styles and other various parts of our application.

When we want to create a controller, we'll need to add a file to the `controllers` directory with a descriptive name. Then we'll need to ensure that we include it in our `index.html` as a file to load.

For instance, adding a dashboard controller, we'll `app/scripts/controllers/dashboard.js` and create our `DashboardCtrl` definition:

```
1 'use strict';  
2  
3 // in app/scripts/controllers/dashboard.js  
4 angular.module('myappApp')  
5 .controller('DashboardCtrl', function($scope) {  
6 });
```

Now, to include this controller we'll tell our application to load this file in our `index.html`. We'll want to make sure we include it inside the build comments in our `app` so that our `htmlmin` task includes it when it's minifying our HTML.

```
1 <!-- build:js({.tmp,app}) scripts/scripts.js -->
2 <script src="scripts/app.js"></script>
3 <script src="scripts/controllers/main.js">
4 </script>
5 <script src="scripts/controllers/dashboard.js">
6 </script>
7 <!-- endbuild -->
```

Now we can use the controller in our app. This same process will work for any type of angular component that we'll use in our app, services, filters, directives, etc.

If we break our app into multiple components (highly recommended) as dependencies of our app, we'll need to ensure we include those before our `app.js` file above. For instance, if we follow the multiple-module pattern where we generate a new module for each component:

```
1 // in app/scripts/services/api.js
2 angular.module('myApp.services', [])
3 .factory(' ApiService', function() {
4   return {};
5 });
```

And set these modules as dependencies for our app:

```
1 // in app.js
2 angular.module('myApp', ['myApp.services']);
```

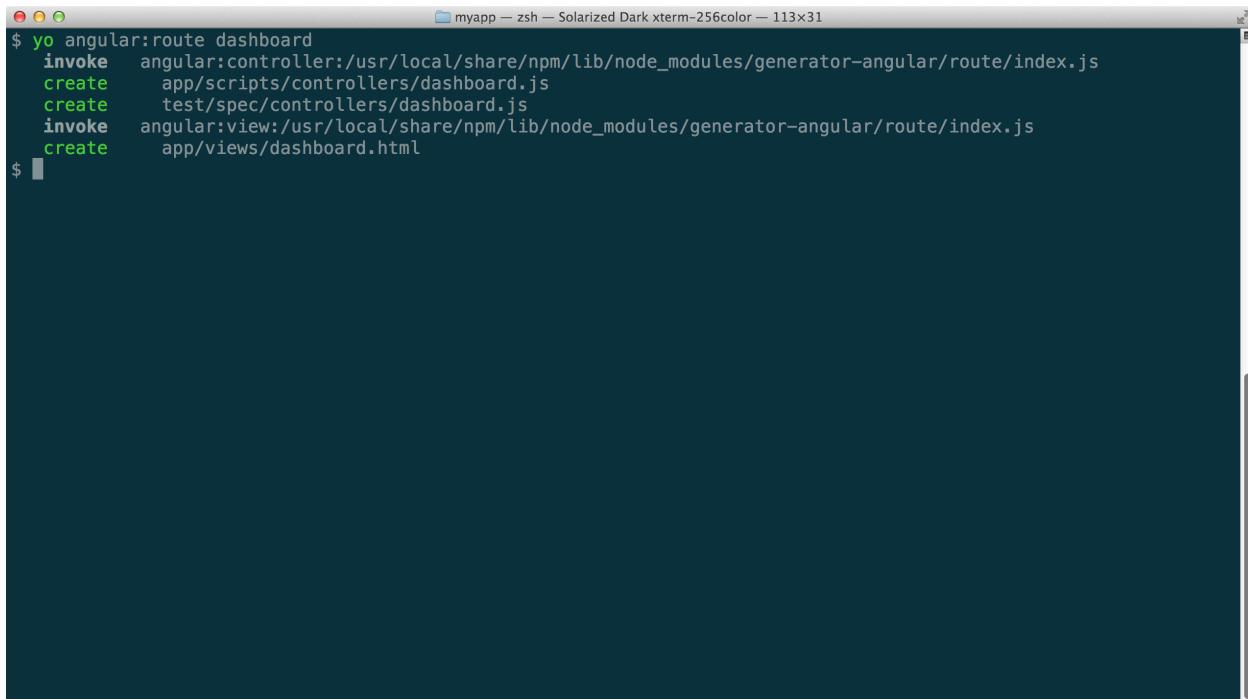
We'll need to include these services *before* we include `app.js` in our HTML.

Alternatively, the Angular generator itself comes with a bunch of helpful generators that make the process of building an angular app a cinch.

Create a route

To create a route, which includes a controller and the corresponding test for the controller, includes the `<script>` tag in the HTML, and creates a view for the route, run the following in our terminal:

```
1 $ yo angular:route home
```

A screenshot of a terminal window titled "myapp — zsh — Solarized Dark xterm-256color — 113x31". The window shows the command \$ yo angular:route dashboard being run. The output indicates that the generator is invoking angular:controller, creating app/scripts/controllers/dashboard.js, creating test/spec/controllers/dashboard.js, invoking angular:view, and creating app/views/dashboard.html.

```
$ yo angular:route dashboard
  invoke  angular:controller:/usr/local/share/npm/lib/node_modules/generator-angular/route/index.js
  create    app/scripts/controllers/dashboard.js
  create    test/spec/controllers/dashboard.js
  invoke  angular:view:/usr/local/share/npm/lib/node_modules/generator-angular/route/index.js
  create    app/views/dashboard.html
```

Create a new route

Create a controller

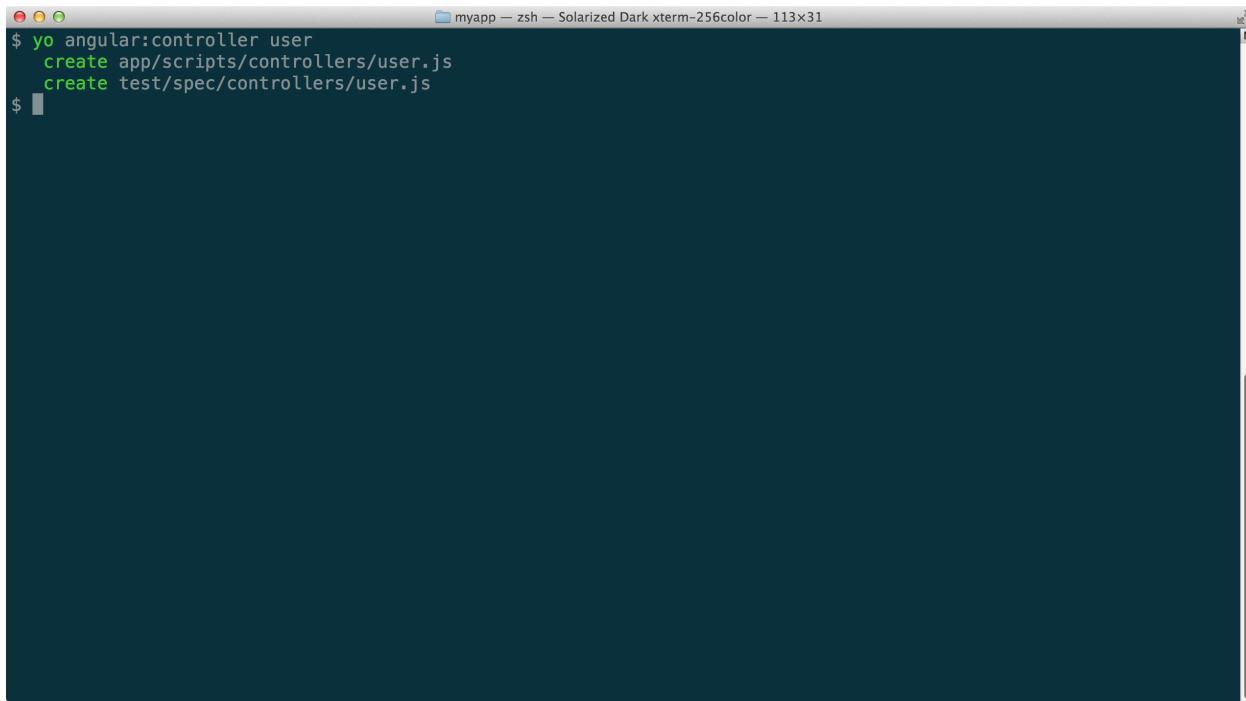
To create a simple controller as well as the corresponding test, we can use the generator in the terminal:

```
1 $ yo angular:controller user
```

Create a custom directive

To create a directive and the corresponding test, we can create a directive using the following command:

```
1 $ yo angular:directive tabPanel
```

A screenshot of a terminal window titled "myapp — zsh — Solarized Dark xterm-256color — 113x31". The window shows the command \$ yo angular:controller user and its output: create app/scripts/controllers/user.js and create test/spec/controllers/user.js.

```
$ yo angular:controller user
  create app/scripts/controllers/user.js
  create test/spec/controllers/user.js
$
```

Create a directive

Create a custom filter

We can also create a custom filter in our app and the corresponding test, we can use the following generator:

```
1 $ yo angular:filter capitalize
```

Create a view

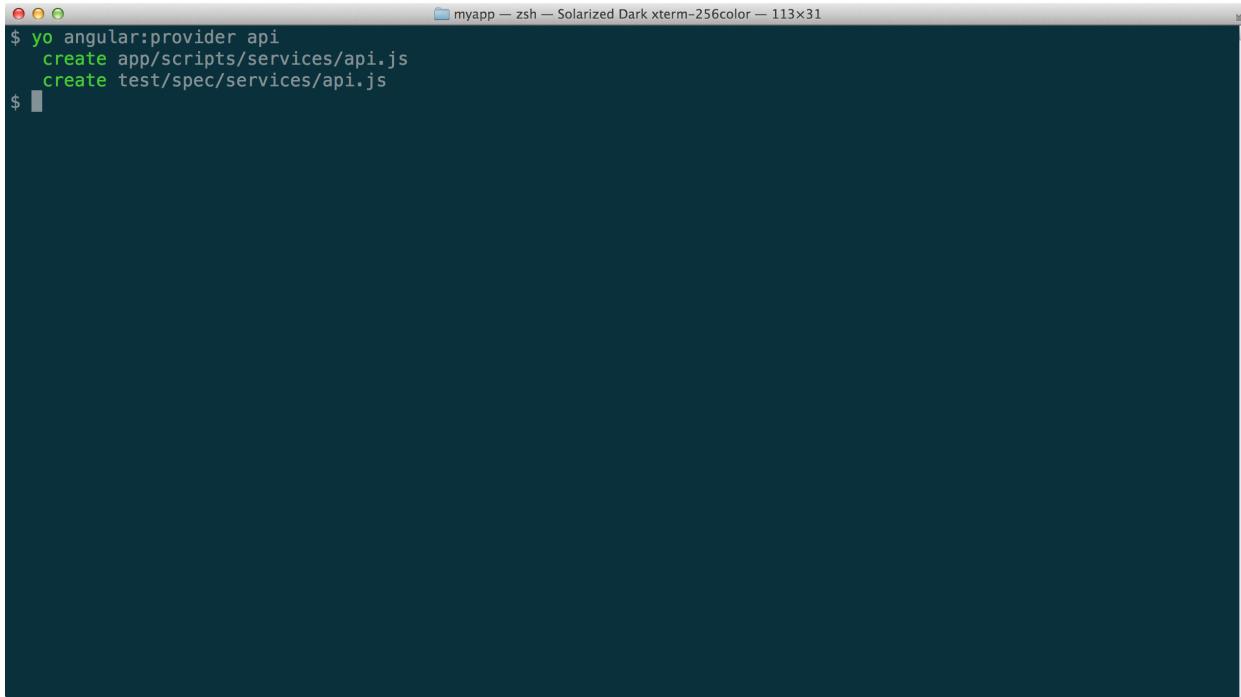
To generate a simple view, we can use the angular generator command:

```
1 $ yo angular:view dashboard
```

Create a service

It's possible to use the generator to create a service as well. The following will create a service as well as the corresponding tests in the different formats that we are able to create a service.

```
1 $ yo angular:service api  
2 $ yo angular:factory api  
3 $ yo angular:provider api  
4 $ yo angular:value api  
5 $ yo angular:constant api
```

A screenshot of a terminal window titled "myapp — zsh — Solarized Dark xterm-256color — 113x31". The window shows the command "\$ yo angular:provider api" followed by two "create" messages: "create app/scripts/services/api.js" and "create test/spec/services/api.js". The terminal has a dark background with light-colored text and a light gray cursor.

Create a provider

Create a decorator

The angular generator also comes with the ability to create a decorator over other services simply. Run this in our terminal:

```
1 $ yo angular:decorator api
```

Configuring the angular generator

With any of the previous generators (including the main generator), we can pass options to configure our scripts in a custom manner.

Coffeescript

If we want to generate coffeescript files instead of javascript files, we can easily do this by passing the --coffee option:

```
1 $ yo angular:controller user --coffee
```

Minification safe

Although not strictly necessary as our yeoman generator includes `ngMin`, we can tell the generator to add dependency injection annotations in our generated file using the `--minsafe` flag:

```
1 $ yo angular:controller user --minsafe
```

Skipping index

By default, all of the previous generators add the appropriate files to be loaded in the `index.html`. We can tell the generator **not** to include the scripts in the `index.html`.



We may want to skip adding a file to the index page as we may be building a 3rd party plugin.

```
1 $ yo angular:factory session --skip-add
```

Testing our app

One of the nicest features of the yeoman angular generator is it's ability to allow us to seamlessly test our application as we are developing it.

The generator comes packaged with a test command that will run whenever we save a file in our app. This makes the process of testing easily translatable into our workflow.

To run our tests without watching our files, in otherwords to run it once, we can use the command:

```
1 $ grunt test
```

This will run once and quit. We recommend making two changes to our workflow to introduce automated testing in our app.

First, we'll open the `Gruntfile.js` at the root of the application and we'll find the karma task. Change the option from `singleRun: true` to `false`:

```
1      // ...
2  ]
3 },
4 karma: {
5   unit: {
6     configFile: 'karma.conf.js',
7     singleRun: false // CHANGE THIS TO FALSE
8   }
9 },
10 cdnify: {
11 // ...
```

Second, open the `karma.conf.js` file and change the option of `autoWatch` from `false` to `true`.

Now, when we run `grunt test`, rather than running once and quitting, the task will stay open and *watch* our files. Once we change a file and save it, then the tests will re-run.

Packaging our app

After we are done building our application, we'll want to set up a distribution for our app. Creating a distribution of our app includes minifying all of the javascripts, our HTML, packaging the views, preprocessing the CSS, etc.

To run the build task, we can simply run the `grunt build` command:

```
1 $ grunt build
```

This will take a few minutes to run the entire generator. When it has completed, we'll end up with a `dist/` folder in the root of our app. This will contain all of the files suitable for production deployment.

We can upload this to our server as-is or include it in deployment with our server and let our application run to the masses.

Packaging our templates

One method that we can use to make our apps appear incredibly fast and not depend upon servers to deliver our templates is to turn our template files into a “javascript” file.

Using the `angular templateCache`, we'll include our templates in a javascript file. For instance, instead of telling angular to fetch the HTML using XHR as HTML:

```
1 <div class="hero-unit">
2   <h1>'Allo, 'Allo!</h1>
3   <p>You now have</p>
4   <ul>
5     <li ng-repeat="thing in awesomeThings">{{thing}}</li>
6   </ul>
7   <p>installed.</p>
8   <h3>Enjoy coding! - Yeoman</h3>
9 </div>
```

We can package them up into a javascript file and let the javascript stand alone like so:

```
1 angular.module('myApp')
2 .run(['$templateCache', function($templateCache) {
3   $templateCache.put('views/main.html',
4     "<div class=\"hero-unit\">\n" +
5     "  <h1>'Allo, 'Allo!</h1>\n" +
6     "  <p>You now have</p>\n" +
7     "  <ul>\n" +
8    "    <li ng-repeat=\"thing in awesomeThings\">{{thing}}</li>\n" +
9    "  </ul>\n" +
10   "  <p>installed.</p>\n" +
11   "  <h3>Enjoy coding! - Yeoman</h3>\n" +
12   "</div>\n"
13 );
14 }]);
```

To set this up, we'll modify our `Gruntfile.js` to include a new task using the `grunt-angular-templates` npm package.

First, install the package:

```
1 $ npm install --save-dev grunt-angular-templates
```

Next, we'll modify our `Gruntfile.js` to include the `ngtemplates` task.

```
1 // ...
2 }
3 },
4 ngtemplates: {
5   myappApp: {
6     cwd: '<%= yeoman.app %>' ,
7     src: 'views/**/*.html' ,
8     dest: '<%= yeoman.app %>/scripts/templates.js'
9   }
10 },
11 // Put files not handled in other tasks here
12 copy: {
13   // ...
```

This will simply create a new file in our app directory that will contain the template files loaded as javascript.

We'll need to make sure that this task is run in the build process. Luckily, it's easy to add a task to the build process task. Find the line: `grunt.registerTask('build', [` and make sure we've added `ngTemplates` into the array of tasks **after** the concat task:

```
1 grunt.registerTask('build', [
2   // ...
3   'concat',
4   // ...
5   'cssmin',
6   'ngtemplates',
7   'uglify',
8   'rev',
9   'usemin'
10 ]);
```

Lastly, we'll need to ensure that we include the `templates.js` file in our `app/index.html` file **after** we include the `scripts/app.js` file:

```
1 <!-- build:js({.tmp,app}) scripts/scripts.js -->
2 <script src="scripts/app.js"></script>
3 <script src="scripts/controllers/main.js"></script>
4 <script src="scripts/templates.js"></script>
5 <!-- endbuild -->
```

Now, when we build our app, our templates will come packaged along with the rest of the application.

Note that when we're developing our application, if the template isn't found in the cache, it will load from the server automatically, so we can safely delete the `app/scripts/template.js` file if we need to throughout our development process.

If this file exists, then the views that it caches won't be reloaded as it will *think* it has the template available.

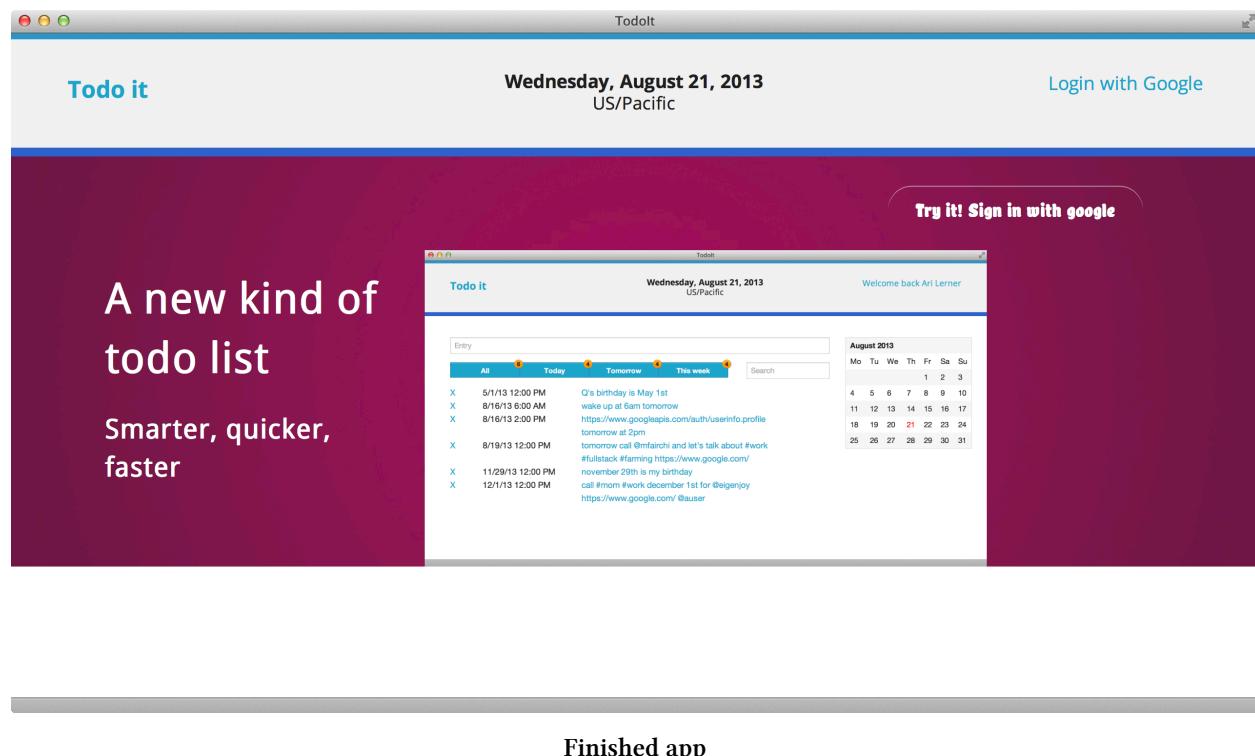
Our app

App development environment

In this book, we'll walk through the process of building a dynamic web app together and explain AngularJS concepts along the way.

We'll be building a complex TODO-style app. The final product will allow a user to log in and authorize the client-side app with Google. It will communicate with Google's calendar API to interact with the Google calendar.

Users will input (in English) the text of their schedule. The app will enable tagging, live search, grouping, and filtering, and it will include a custom calendar element.



The screenshot shows the TodoIt application interface. At the top, it displays "TodoIt", the date "Wednesday, August 21, 2013", and the time zone "US/Pacific". A message "Welcome back Ari Lerner" is also visible. Below the header, there's a search bar containing the text "Call @mom tomorrow at 5pm to wish her happy #birthday #family #todo". Underneath the search bar, there are five tabs: "All" (2 items), "Today" (1 item), "Tomorrow" (1 item), and "This week" (1 item). A "Search" input field is also present. The main area lists two tasks: "X 5/1/13 12:00 PM Q's birthday is May 1st" and "X 11/29/13 12:00 PM november 29th is my birthday". To the right, there's a calendar for August 2013, showing the days of the week and the dates from 1 to 31. Below the calendar, sections for "Users" (@mom) and "Tags" (#birthday, #family, #todo) are displayed.

Finished app



For the purposes of keeping this book focused on learning AngularJS, we will walk through how to build a raw AngularJS app without using any tools beyond AngularJS. We'll discuss production tools at length in our *production tools* chapter.

To build this app, we're going to start with a super simple server. In fact, we won't even need to write code for our server: We'll use the Python SimpleHTTPServer for the time being.



Python's SimpleHTTPServer is a simple server written in Python that serves all the files in a single directory. It enables us to get started working quickly, rather than needing to write our own.

To start the app using the SimpleHTTPServer, we'll head to our terminal and change into the directory where we'll start building the app and type:

```
1 $ python -m SimpleHTTPServer 9000
```

Let's navigate to a directory where we'll keep the development of our code. We recommend creating a directory in a location that you navigate often. For instance, we recommend creating a directory in your Home folder (OSX/*nix) or creating a Development directory in My Documents/ (Windows). Ultimately, the choice is up to you.

For the duration of this book, we'll refer to this top level directory as your app's root directory.

When we start building our app, we'll place an `index.html` in this directory and head to our browser at `http://localhost:9000`. In the browser, our `index.html` will be served through our local server.

To build the template for our app, you can do one of two things: Use git to clone the example repository that we've made available with the book or manually build the template.

Cloning the demo app repository

We have made the example code available on github. If you prefer to start with the example code that we have provided, simply clone the repository from github.

You can either head to <https://github.com/fullstackio/ng-book>¹⁷¹ and download the latest zip.

It's also an option to clone the repository on your local machine using the following git command:

```
1 $ git clone git@github.com:fullstackio/ng-book.git
```

If you want to follow along with building the sample app, the code is included with the book. For the majority of this book, we'll be working with the custom-built `sample_app`.

This code can be found in the `bookdirectory/sample_app/todoit-custom`.

Manually building your app

If instead, you'd prefer to build the app from scratch, the directory structure we will start off with will look like this:

```
1 /
2   app/
3     js/
4     css/
5     lib/
6     img/
7     views/
8     index.html
```

Create these directories and files on your local machine. Open your terminal and type the following to create the initial structure:

¹⁷¹<https://github.com/fullstackio/ng-book>

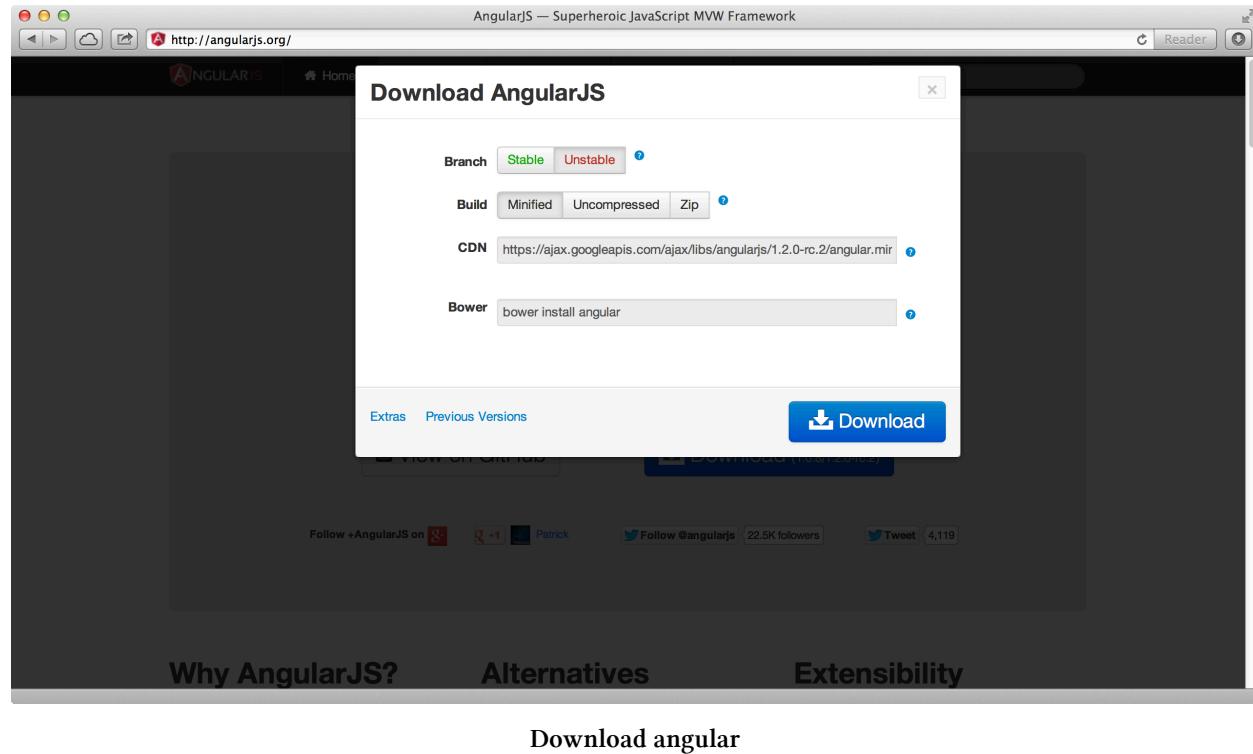
```
1 $ mkdir -p app/{js,css,lib,img,views}  
2 $ touch app/index.html
```

In building our app, we'll use the following directories to separate our different functions:

- **index.html**: our main application template
- **app/**: Our main app directory. This is where we'll serve our app from
- **app/js/**: where we'll store our application-specific JavaScript files that we write
- **app/css/**: where we'll store our CSS styles
- **app/lib/**: where we'll store any JavaScript library files that we download, like angular.js
- **app/views/**: where we'll store our AngularJS templates

Downloading angular

Once our directories are set, let's download [angular.js¹⁷²](#) and store the file in our `app/lib/` folder. Additionally, make sure you grab the [angular-route.js¹⁷³](#) library and store that in the `app/lib/` folder. The `angular-route.js` file is available at [angularjs.org¹⁷⁴](#) if you click on `extras` after clicking the download button.



¹⁷²<https://ajax.googleapis.com/ajax/libs/angularjs/1.2.0-rc.2/angular.min.js>

¹⁷³<http://code.angularjs.org/1.2.0-rc.2/angular-route.js>

¹⁷⁴<http://angularjs.org/>

Make sure you grab both angular.js and angular-route.js and store them in the app/lib folder.

In the demo application that comes with this book, we're using the [foundation.zurb.com¹⁷⁵](http://foundation.zurb.com) CSS framework. If you'd prefer to use another CSS framework, feel free to grab it and any JavaScript files that are associated with it and store them in the appropriate directories, but we recommend sticking with the book for the best experience.

Finally, when we're ready to go, let's fill out our index.html file like so:

```
1 <!doctype html>
2 <html lang="en" ng-app="myApp">
3 <head>
4   <meta charset="utf-8">
5   <title>TodoIt</title>
6   <link rel="stylesheet" href="css/normalize.css"/>
7   <link rel="stylesheet" href="css/app.css"/>
8   <meta name="viewport" content="width=device-width">
9 </head>
10 <body>
11   <!-- Library files -->
12   <script src="lib/angular.js"></script>
13   <script src="lib/angular-route.js"></script>
14   <!-- App files -->
15   <script src="js/filters.js"></script>
16   <script src="js/controllers.js"></script>
17   <script src="js/directives.js"></script>
18   <script src="js/services.js"></script>
19   <script src="js/app.js"></script>
20 </body>
21 </html>
```

Now that we have our index.html page filled, we need to create a few files that will contain our app.

Create these five JavaScript files in your app/js/ folder:

- app/js/filters.js
- app/js/controllers.js
- app/js/directives.js
- app/js/services.js
- app/js/app.js

We can do so with another terminal command:

¹⁷⁵<http://foundation.zurb.com>

```
1 $ touch app/js/{filters.js,controllers.js,directives.js}  
2 $ touch app/js/{services.js,app.js}
```

Each of these files will contain the different parts of our application that their titles suggest. We'll discuss what should go in each of these files as we continue through the book.

Booting it up

Finally, to see this app immediately in action, change into the `app/` folder in your terminal and start up the Python `SimpleHTTPServer`:

```
1 $ cd app/  
2 $ python -m SimpleHTTPServer 9000
```

We'll head to the browser and navigate to `http://localhost:9000`. We should see an empty, but running app.

Modules

As you may recall [modules](#) are how we define and organize our angularjs app. Let's create 5 different modules.

- A service module
- A directive module
- A filter modules
- A controller module
- A main app module

Our service module, `js/services.js` we'll make a services module. For the time being, we'll leave it relatively empty. We'll only declare an app version in the module.

js/services.js

```
1 angular.module('myApp.services', [])
2   .value('version', '0.0.1');
```

We'll contain our custom directives in the `js/directives.js` file. We'll cover directives in-depth in the [directives](#) chapter.

js/directives.js

```
1 angular.module('myApp.directives', []);
```

As we would expect, we'll contain our filters in the `js/filters.js` file. We'll cover filters in-depth in the [filters](#) chapter.

js/filters.js

```
1 angular.module('myApp.filters', []);
```

All of our controllers we will write will be placed in the controllers module in `js/controllers.js`. We will cover controllers in-depth in the [controllers](#) chapter.

js/controllers.js

```
1 angular.module('myApp.controllers', []);
```

Finally, we'll write our main app module that will depend upon all of the modules we just declared. When specifying a module the array [] is a list of all of the dependencies the app needs to be bootstrapped. Angular will not bootstrap the module until these dependencies have been found and met.

Our app depends on all of our modules to run, so we'll need to be sure to include them in the second parameter. We'll cover this *dependency injection* in the [dependency injection](#) chapter.

js/app.js

```
1 var app = angular.module('myApp', [
2   'myApp.services',
3   'myApp.directives',
4   'myApp.filters',
5   'myApp.controllers'
6 ]);
```

If we navigate to our app, we'll notice that we haven't extended the functionality yet, but we're ready to start developing a robust app.

Starting our app

Now that we have a pretty good understanding of data-binding, \$scopes, and controllers, let's start to add functionality to our application.

Provided we followed the steps in the introduction, we should have our development environment ready to go. If not, feel free to review the *introduction* chapter for a step-by-step of setting up our development environment.

In our app, we have a header that will show the current date. To accomplish this, we'll create a variable on our FrameController called today. This today variable will hold the value of our date.

If our development environment is freshly created, we should have a FrameController controller in our js/controller.js file that looks like:

```
1 angular.module('myApp.controllers', [])
2 .controller('FrameController', ['$scope', function($scope) {
3 }]);
```

We'll use this FrameController as the parent controller to all of our other controllers. Let's add the variable called today on it and set it equal to today's date:

```
1 angular.module('myApp.controllers', [])
2 .controller('FrameController', ['$scope', function($scope) {
3   $scope.today = new Date(); // Today's date
4 }]);
```

Now, in our view we can simply reference today and that value will be replaced with the value of new Date(). In our index.html, let's add a reference to today's date to see that change reflected in the view.

```
1 <!doctype html>
2 <html lang="en" ng-app="myApp">
3 <head>
4   <meta charset="utf-8">
5   <title>TodoIt</title>
6   <link rel="stylesheet" href="css/vendor/normalize.css"/>
7   <link rel="stylesheet" href="css/app.css"/>
8   <meta name="viewport" content="width=device-width">
9 </head>
10 <body ng-controller="FrameController">
11   {{ today }}
12   <!-- Library files -->
13   <script src="lib/angular.js"></script>
14   <script src="lib/angular-route.js"></script>
15   <!-- App files -->
16   <script src="js/filters.js"></script>
17   <script src="js/controllers.js"></script>
18   <script src="js/directives.js"></script>
19   <script src="js/services.js"></script>
20   <script src="js/app.js"></script>
21 </body>
22 </html>
```



First databinding

Now, when we can see we have a pretty ugly date in our view. We'll clean this up using filters later in this book.

Next, we'll eventually allow a user to log in to our app using [Google APIS¹⁷⁶](#), but for the time being we'll set our current user's name to a static name.

In our `FrameController` again, add the property `name` to the scope:

```
1 angular.module('myApp.controllers', [])
2 .controller('FrameController', ['$scope', function($scope) {
3   $scope.today = new Date(); // Today's date
4   $scope.name = "Ari Lerner"; // Our logged-in user's name
5 }]);
```

And in our view, we can simply reference our user's name, like like before.

¹⁷⁶<https://developers.google.com/>

```
1 <!-- head and include files above -->
2 <body ng-controller="FrameController">
3   <h1>{{ today }}</h1>
4   <h2>Welcome back {{ name }}</h2>
5 <!-- include files -->
```

Now that our FrameController is holding on to some variables we'll use in multiple templates, we can create a child controller of the FrameController called the DashboardController.

First, let's edit our js/controllers.js file and add a new controller:

```
1 angular.module('myApp.controllers', [])
2   .controller('FrameController',
3     ['$scope', function($scope) {
4       $scope.today = new Date();
5       $scope.name = "Ari Lerner"; // Our logged-in user's name
6     }])
7   .controller('DashboardController',
8     ['$scope', function($scope) {
9   }]);
```



With two variables bound

Almost all of our application code up until this point will be nested in the DashboardController. First, we'll need to nest this controller in our view.

```

1 <body ng-controller="FrameController">
2   <div ng-controller="DashboardController">
3     <h1>{{ today }}</h1>
4     <h2>Welcome back {{ name }}</h2>
5   </div>

```

As we saw above, we can still reference the variables defined on our FrameController inside the DOM element of our DashboardController.

Now, let's create an input field where our users will type their todo items. To do this, we'll create a entryInput attribute on the \$scope of the DashboardController.

This entryInput won't need to be initialized at the outset as it will be bound to the input field. If we were to initialize it, then the input field would be filled with the value of our entryInput.

First, in our `index.html` file, let's create an input field that we'll bind the entryInput variable:

```

1 <div ng-controller="DashboardController">
2   <h1>{{ today }}</h1>
3   <h2>Welcome back {{ name }}</h2>
4   <input ng-model="entryInput" type="text" placeholder="New Entry" />
5 </div>

```

Anytime that the input box value is changed, the `$scope.entryInput` will change thanks to the bi-directional data-binding. Our updated controller looks like:

```

1 angular.module('myApp.controllers', [])
2   .controller('FrameController',
3     ['$scope', function($scope) {
4       $scope.today = new Date();
5       $scope.name = "Ari Lerner"; // Our logged-in user's name
6     }])
7   .controller('DashboardController',
8     ['$scope', function($scope) {
9       $scope.entryInput = undefined;
10    }]);

```



Because we want the input field to start out as undefined, we do **not** need to reference it in the DashboardController, as we did above for the `name` and `today` attributes. It is good practice, however to create variables inside the controller to be explicit.

In the following snippets, we'll be setting up the functionality of our app the will allow users to input their friends into the todo input box.

We'll *watch* the `entryInput` field and parse the the value entered for dates, #hashtags, links, and @users.



Note: we'll move this functionality into a `$filter` to keep our controllers clean.

Looking at our current view, we haven't changed the html

```
1 <div ng-controller="DashboardController">
2   <h1>{{ today }}</h1>
3   <h2>Welcome back {{ name }}</h2>
4   <input ng-model="entryInput" type="text" placeholder="New Entry" />
5 </div>
```

Also suppose that we've got a list of friends that are associated with the current user on the controller's scope:

```
1 $scope.users = {
2   "ari": {
3     "twitter": "@auser"
4   },
5   "nate": {
6     "twitter": "@eigenjoy"
7   }
8 };
```

We'll set up a `$watch` on the `entryInput` value that will track any changes that happen on it. When the value changes, we'll want to try to match any user that is set in the input box with tokens that start with a @ symbol, such as: `@ari`.

As we previously set up, we'll only need to make sure we *inject* the `$parse` service into our controller:

```
1 .controller('DashboardController',
2 ['$scope', '$parse', function($scope, $parse) {
3   // Setting up a watch expression to watch the entryInput
4   $scope.$watch('entryInput', function(newVal, oldVal, scope) {
5     if (newVal !== oldVal) {
6       // newVal now has the latest updated version
7     }
8   });
9 }]);
```

On change, let's look through the input text box and find any tokens that start with the @ symbol using the built in regex feature of javascript called `match()`:

```
1 $scope.$watch('entryInput', function(newVal, oldVal, scope) {
2   if (newVal !== oldVal) {
3     // Look for any part of the string that starts with @
4     var strUsers = newVal.match(/[@]+[A-Za-z0-9_]+/g),
5       i;
6
7     // If any part of the string starts with @, then we
8     // will have a list of those tokens in strUsers
9     if (strUsers) {
10       // We'll loop through our users and parse the $scope
11       // looking for user
12       for (i = 0; i < strUsers.length; i++) {
13         var user = strUsers[i]; // Found user in the form @[user]
14       }
15     }
16   }
17 });

});
```

Now that we are looping through any user that we may find and storing that user into a local variable of user. Let's modify the loop to use the \$parse service to look through the current scope for the user:

```
1 for (i = 0; i < strUsers.length; i++) {
2   var user = strUsers[i], // Found user in the form @[user]
3     cleanUser = user.slice(1), // Remove the @ symbol
4     // In here, we'll look up the cleanUser in the local
5     // scope users object. For instance, if the user
6     // inputted "@ari" in the input, this would try to find
7     // users.ari in the local scope.
8   parsedUser = $parse("users." + cleanUser)(scope);
9
10  if (parsedUser) {
11    // A user was found on our scope in the "users" object
12  } else {
13    // No user was found on our scope in the "users" object
14  }
15}
```

From here, we can assume we have the user in the parsedUser object if it's found on the local "users" object. If the parsedUser is undefined, it means it could not find the user on the scope. If it is found, then parsedUser will contain the object, if not, then it will be undefined.



Note that this example is simply set up to demonstrate how to use the `$parse` service. A cleaner, more performant method of doing this would be to create a method on the scope that looks up the user, rather than appending a string as we've done above.

From here, we can choose to do what we want with the `parsedUser` object. In our case, we'll store it when we build the `parse $filter`.

In our app, we'll create several filters that will enable us to control the expected behavior of the app.

- `blankIfNegative` - a filter that ensures only the dates in the calendar's month appears
- `excludeByDate` - filters out any dates that do not belong in a date keep filter (such as today, tomorrow, next week, etc.)
- `parseEntry` - parses the `entryInput` field and pulls out any interesting tags, users, urls, and dates that the user writes into the entry field
- `searchFilter` - a search filter that enables us to *live search* the current list of events

In our app, we'll nest all of our filters in the `myApp.filters` module, as is best practice:

```
1 angular.module('myApp.filters', []);
```

We'll need to ensure that this module is set as a dependency for our app, we'll add this `myApp.filters` as a runtime dependency in our `myApp` module definition (in our `js/app.js` file):

```
1 angular.module('myApp', [
2   'myApp.filters'
3 ]);
```

blankIfNegative filter

We'll use the `blankIfNegative` filter in the calendar HTML that will prevent any negative values from showing up in the calendar table.

As we have not yet created our calendar's directive, this `filter` will be used in our HTML to show a space if the value passed in is non-positive. For instance:

```
1 <h2>{{ 15 | blankIfNegative }}</h2> <!-- 15 -->
2 <h2>{{ -15 | blankIfNegative }}</h2> <!-- -->
```

To create this filter, we'll use the `filter()` API:

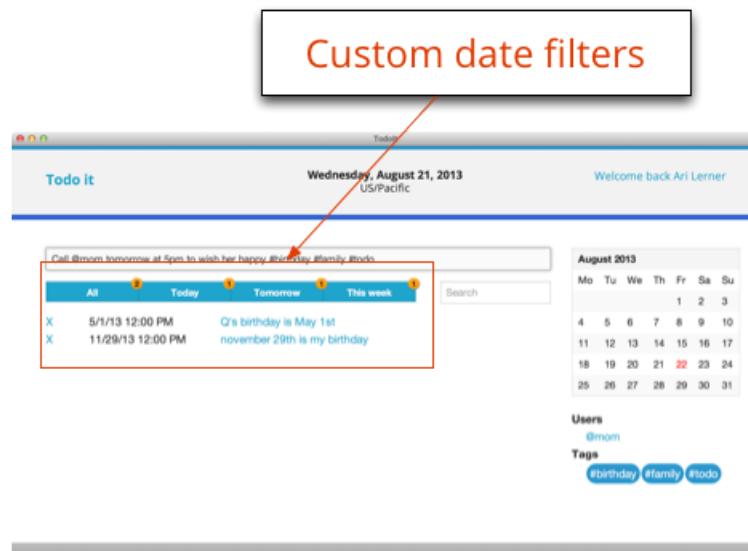
```

1 angular.module('myApp.filters', [])
2   .filter('blankIfNegative', function() {
3     return function(input) {
4       if (input <= 0) return '';
5       else return input;
6     }
7   });

```

excludeByDate filter

In the main dashboard view of our app, we're repeating over the list of our events in our Google calendar. Since we have a few date filters as objects in our scope, we'll want to filter by our date.



Custom date filters

In our HTML, we have buttons that are set to change a specific filter of a before-by date. Clicking on one of the filters will change the `keepDate` property on our `$scope` object.

Let's look at this process a bit deeper. The HTML we'll use looks similar to the following:

```

1 <ul>
2   <li><a ng-click='keepDate=dateFilters["all"]'>
3     All
4   </a></li>
5   <li><a ng-click='keepDate=dateFilters["today"]'>
6     Today
7   </a></li>
8   <li><a ng-click='keepDate=dateFilters["tomorrow"]'>
9     Tomorrow
10  </a></li>
11 </ul>

```

Clicking on these links will update the `keepDate` property on our `DashboardController`. Obviously we need to set up our `dateFilters` property on our scope. Since these dates won't change often, we can set them as a single property value on our scope.



To really be efficient, we can move the `dateFilters` into a service.

```

1 .controller('DashboardController',
2   ['$scope',
3   function($scope) {
4     var date = new Date();
5     $scope.dateFilters = {
6       'all': 'all',
7       'next week': new Date(date.setDate(date.getDate() + 7)),
8       'tomorrow': chrono.parseDate('tomorrow at 11:59pm'),
9       'today': chrono.parseDate('today at 11:59pm')
10    }
11 }]);

```



Note that for this function to work properly, we'll need to ensure we include the `chrono` library at the head of our HTML document.

With the backend set, we can implement this filter in the HTML by setting it to filter against the list of events in the standard lists filter manner:

With this set, we'll be using the `excludeByDate` filter in our view. The `excludeByDate` filter will receive the entire array of events as a parameter as well as a second parameter of the filter date.

The function itself will:

- check if the date passed in is ‘all’. If so, it will return the entire set
- Return only the events who’s start.dateTime is before the filterBy date

```

1 .filter('excludeByDate', function() {
2   return function(arr, date) {
3     if (date === 'all') return arr;
4     var newArr = [];
5     angular.forEach(arr, function(evt) {
6       var evtDate = new Date(evt.start.dateTime);
7       if (evtDate <= date)
8         newArr.push(evt);
9     });
10    return newArr;
11  }
12 });

```

One thing to note is that filters run a lot on almost all of the \$digest loops. We can achieve this same effect with slightly better performance and more control if we move this filter function to the controller. Luckily, this is not difficult to do.

Instead of calling this custom filter in the view, we’ll use the standard filter function in the view.

Inside the DashboardController we can change the function as it will be called once for each element and return true if we want to keep the date in the list or false if we do not:

```

1 $scope.excludeByDate = function(input) {
2   if ($scope.keepDate === 'all') {
3     return true;
4   } else {
5     return new Date(input.start.dateTime).getTime() <
6                   $scope.keepDate.getTime();
7   }
8 }

```

These two implementations of this filter process are effectively equivalent.

parseEntry filter

The last filter we’ll need in our app is the parseEntry filter. This filter, unlike our other two will be called directly from our controller. Instead of returning a true or false condition for display purposes, we’ll use this filter to parse our entryInput text.

The filter itself will take two parameters, a single value of the text inputed into the newEntry text and a javascript object of known users.

```

1 .filter('parseEntry', ['$parse', function($parse) {
2   return function(val, users) {
3     var i = 0;
4     var data = { raw: val };
5     if (val) {
6       // Our parsing functions will go here
7     }
8   }]);

```

Now inside our `parseEntry` filter, if we are given a value that makes sense, we can dissect the text. As the majority of this code is simple javascript, we won't cover it in detail, but the source is attached for completeness. Note that we are returning an object that potentially has the keys:

- raw - the raw entry
- tags - any text prepended with #
- users - any text prepended with @
- date - the chrono parsed date

```

1 .filter('parseEntry', ['$parse', function($parse) {
2   return function(val, users) {
3     var i = 0;
4     var data = {
5       raw: val
6     };
7
8     if (val) {
9       // Find urls
10    var strUrls = val.match(/[A-Za-z]+:\//[A-Za-z0-9-_]+\.[A-Za-z0-9-_:%&~\?\/\.
11 .=]+/g),
12    urls = [];
13
14    if (strUrls) {
15      for (i = 0; i < strUrls.length; i++) {
16        urls.push(strUrls[i]);
17      }
18      val = val.replace(/[A-Za-z]+:\//[A-Za-z0-9-_]+\.[A-Za-z0-9-_:%&~\?\/.=]+\
19 /g, '');
20      data['urls'] = urls;
21    }
22
23    // Find tags

```

```
24     var strTags = val.match(/[#+][A-Za-z0-9_]+/g),
25         tags = [];
26
27     if (strTags) {
28         for (i = 0; i < strTags.length; i++) {
29             tags.push(strTags[i]);
30         }
31         val = val.replace(/[#+][A-Za-z0-9_]+/g, '');
32         data['tags'] = tags;
33     }
34
35 // Find users
36 var strUsers = val.match(/[@][A-Za-z0-9_]+/g),
37     users = [];
38
39 if (strUsers) {
40     for (i = 0; i < strUsers.length; i++) {
41         var user = strUsers[i];
42
43         if (users) {
44             var parseVal = $parse("users." + user.slice(1))(users);
45             if (typeof(parseVal) === 'undefined') parseVal = user;
46         }
47         else {
48             var parseVal = user;
49         }
50
51         users.push(parseVal);
52     }
53     val = val.replace(/[@][A-Za-z0-9_]+/g, '');
54     data['users'] = users;
55 }
56
57 // Finally, parse the date
58 var date = chrono.parseDate(val);
59 if (date) {
60     data['date'] = date;
61 }
62 }
63
64 return data;
65 }
```

66 {])

Creating our-calendar directive

```
1 app.directive('ourCalendar', function() {
2   return {
3     restrict: 'A',
4     require: '^ngModel',
5     scope: {
6       ngModel: '='
7     },
8     template: '<table class="calendar"></table>'
9   }
10 });
});
```

Going back to our calendar directive, we'll need to pass in a date with our directive from which to display. To do that, we could put in a date via text, but that will be pretty ugly. Instead, we'll *bind* it to the surrounding controller of 'date'.

```
1 <div our-calendar ng-model="date"></div>
```

Our directive now looks like this:

```
1 app.directive('ourCalendar', function() {
2   return {
3     restrict: 'EA',
4     require: '^ngModel',
5     scope: {
6       ngModel: '='
7     },
8     template: '<table class="calendar"></table>'
9   }
10 });
});
```

Drawing the calendar

To draw the calendar, we'll need to fill out our template. Luckily, the template is not that complex:



We're using the \ at the end of each line to allow for us to be able to break up the template into multiple lines.

```

1 app.directive('ourCalendar', function() {
2   return {
3     restrict: 'EA',
4     require: '^ngModel',
5     scope: {
6       ngModel: '='
7     },
8     template: '<table class="calendarTable"> \
9       <thead> \
10      <tr> \
11        <td class="monthHeader" colspan="7">{{ monthName }} {{ year }}</td> \
12      </tr> \
13    </thead> \
14    <tbody> \
15      <tr> \
16        <td ng-repeat="d in weekDays">{{ d }}</td> \
17      </tr> \
18      <tr ng-repeat="arr in month"> \
19        <td ng-repeat="d in arr track by $index" ng-class="{currentDay: d == da \
20 y}"> \
21          {{ d }} \
22        </td> \
23      </tr> \
24    </tbody> \
25  </table> '
26 }
27 });

```

Now, in order to actually render our template with the date, we'll have to set up the binding with the `link` function. We'll set up a `watch` on the `ngModel` that will watch for changes on the date. That way, anytime the date changes in the containing controller, the calendar can reflect that change.

Controller option

In a directive, when we set the controller option, we are creating a controller for the directive. This controller is instantiated before the pre-linking phase.

The pre-linking and post-linking phases are executed by the compiler. The pre-link function is executed before the child elements are linked, while the post-link function is executed after. It is only safe to do DOM transformations after the post-link function.

We are defining a controller function in our directive, so we don't need to define either of these functions, but it is important to note that we cannot do DOM manipulations in our controller

function because the controller is instantiated before the pre-link phase, thus we have no access to the DOM yet.

What does a controller function look like? Just like any other controller. Let's see what it looks like when we inject the `$http` service in our controller (using the bracket injection notation):

```
1 app.directive('ourCalendar', function() {
2   return {
3     restrict: 'EA',
4     require: '^ngModel',
5     scope: {
6       ngModel: '='
7     },
8     template: '<table class="calendarTable"> \
9       <thead> \
10         <tr> \
11           <td class="monthHeader" colspan="7">{{ monthName }} {{ year }}</td> \
12         </tr> \
13       </thead> \
14       <tbody> \
15         <tr> \
16           <td ng-repeat="d in weekDays">{{ d }}</td> \
17         </tr> \
18         <tr ng-repeat="arr in month"> \
19           <td ng-repeat="d in arr track by $index" ng-class="{currentDay: d == da \
20 y}"> \
21             {{ d }}< \
22             </td> \
23           </tr> \
24         </tbody> \
25       </table>',
26     controller: ['$scope', '$http', function($scope, $http) {
27       $scope.getHolidays = function() {}
28     }]
29   }
30 });
```

Note that in our link function, we have access to all of the attributes that were declared in the DOM element. This will become important in a minute when we go to customize the directive.

We're not making any ajax or api calls in this chapter, we won't actually create our `getHolidays` function, but it's good to note that this is how we can make `http` requests inside our directive.

```
1 <div our-calendar ng-model="date"></div>
```

Let's start building our link method that will actually build and draw the calendar on screen.

The \$watch function will register a callback to be executed whenever the result of the expression changes. Inside the \$digest loop, every time AngularJS detects a change, it will call this function. This has the side effect that we cannot depend on state inside this function. To counter this, we'll check for the value before we depend on it being in place.

Here is our new \$watch function:

```
1 scope.$watch(attrs.ngModel, function(date) {  
2   if (date) {  
3  
4   }  
5 });
```

Every time the *date* property on our scope changes, our function declared in the watchExpression will fire.

Here's what we have so far:

```
1 app.directive('ourCalendar', function() {  
2   return {  
3     restrict: 'EA',  
4     require: '^ngModel',  
5     scope: {  
6       ngModel: '='  
7     },  
8     template: '<table class="calendarTable">\n      <thead>\n        <tr>\n          <td class="monthHeader" colspan="7">{{ monthName }} {{ year }}</td>\n        </tr>\n      </thead>\n      <tbody>\n        <tr>\n          <td ng-repeat="d in weekDays">{{ d }}</td>\n        </tr>\n        <tr ng-repeat="arr in month"> \n          <td ng-repeat="d in arr track by $index" ng-class="{currentDay: d == da\\  
20 y}">\n            {{ d }}\n          </td>\n        </tr>\n      </tbody>\n    </table>'  
21   }  
});
```

```

22         </td> \
23     </tr> \
24   </tbody> \
25 </table>',
26 controller: ['$scope', '$http', function($scope, $http) {
27   $scope.getHolidays = function() {}
28 },
29 link: function(scope, ele, attrs, ctrl) {
30   scope.$watch(attrs.ngModel, function(newDate, oldDate) {
31     if (!newDate)
32       newDate = new Date(); // If we don't specify
33                           // a date, make the
34                           // default date today
35   });
36 }
37 }
38 });

```

Adding in our drawing code isn't particularly interesting or angular specific, so we'll include it here:

```

1 app.directive('ourCalendar', function() {
2   // Array to store month names
3   var months = new Array('January', 'February', 'March', 'April', 'May', 'June', \
4 'July', 'August', 'September', 'October', 'November', 'December');
5   // Array to store month days
6   var monthDays = new Array(31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31);
7   // Array to store week names
8   var weekDay = new Array('Mo', 'Tu', 'We', 'Th', 'Fr', 'Sa', 'Su');
9   return {
10     restrict: 'EA',
11     require: '^ngModel',
12     scope: {
13       ngModel: '='
14     },
15     template: '<table class="calendarTable"> \
16       <thead> \
17         <tr> \
18           <td class="monthHeader" colspan="7">{{ monthName }} {{ year }}</td> \
19         </tr> \
20       </thead> \
21       <tbody> \
22         <tr> \

```

```
23      <td ng-repeat="d in weekDays">{{ d }}</td> \
24    </tr> \
25    <tr ng-repeat="arr in month"> \
26      <td ng-repeat="d in arr track by $index" ng-class="{currentDay: d == da\
27 y}"> \
28        {{ d }} \
29      </td> \
30    </tr> \
31  </tbody> \
32 </table>',
33 controller: ['$scope', '$http', function($scope, $http) {
34   $scope.getHolidays = function() {}
35 },
36 link: function(scope, ele, attrs, c) {
37   scope.$watch(attrs.ngModel, function(date) {
38     if (!date) date = new Date();
39     var week_day, day_counter, i, curr_week;
40
41     var day = date.getDate(),
42       month = date.getMonth(),
43       year = date.getFullYear();
44     scope.days_in_this_month = monthDays[month];
45     scope.monthName = months[month];
46
47     scope.currentWeek = 0;
48
49     scope.month = {};
50
51     var thisMonth = new Date(year, month, 1),
52       firstDay = new Date(thisMonth.setDate(1)).getDay(),
53       weeksOfMonth = Math.ceil((firstDay + scope.days_in_this_month)/ 7) + \
54 1;
55
56     scope.weekDays = weekDay;
57     // First week
58     curr_week = 0;
59
60     scope.month[curr_week] = [];
61     for (week_day = 0; week_day < thisMonth.getDay(); week_day++) {
62       scope.month[curr_week][week_day] = week_day * -1;
63     }
64     week_day = thisMonth.getDay();
```

```

65         for (day_counter = 1; day_counter <= scope.days_in_this_month; day_counter\
66 r++) {
67     week_day %= 7;
68
69     if (week_day == 0) {
70         curr_week += 1;
71         scope.month[curr_week] = [];
72     }
73
74     scope.month[curr_week].push(day_counter);
75
76     week_day += 1;
77 }
78
79 while(scope.month[curr_week].length < 7) {
80     scope.month[curr_week].push(day_counter * -1);
81 }
82
83     scope.day = day;
84     scope.year = year;
85 });
86 }
87 }
88 });

```

Now that we have our calendar being draw, let's give our directive some customizable features.

...

For instance, our calendar right now features the full name of the calendar in the header. However, if we are using this calendar in a place that does not have a lot of room on the page, we might want to hide the full length and only show a shortened version of it instead.

We have access to the attrs (the normalized list of the attributes that we define in our HTML), we can simply look there first; otherwise we can set a default to see what how we want to show the header.

Let's change the setting of the \$scope.monthname to:

```
1 if (attrs.showshortmonth) {  
2   scope.monthName = months[month].slice(0, 3);  
3 } else {  
4   scope.monthName = months[month];  
5 }
```

Now, we'll only show the first three characters of the name of the month if we include the showshortmonth as an attribute. When we invoke the directive, we'll use the attribute:

```
1 <div our-calendar showshortmonth='true'></div>
```

Now our calendar's header will change to show the shortened 3-character calendar header.

Transclude option

Although the name sounds complex, transclusion just refers to compiling the content of the element and making the source available to the directive. The transcluded function is pre-bound to the calling scope, so it has access to the current calling scope.

Looking at an example, let's change the invocation to:

```
1 <div our-calendar ng-model="date">  
2   <h3>On this day...</h3>  
3 </div>
```

To get this to show up in our template, we'll need to use a special directive called `ngTransclude`. This is how the template knows where to place the custom HTML. Let's modify the template in our directive to include the custom content:

```
1 template: '<table class="calendarTable">\n  <thead>\n    <tr><td class="monthHeader" colspan="7">{{ monthName }} {{ year }}</td>\n  </tr></thead>\n  <tbody>\n    <tr><td ng-repeat="d in weekDays">{{ d }}</td></tr>\n    <tr ng-repeat="arr in month"> \n      <td ng-repeat="d in arr track by $index" ng-class="{currentDay: d == day}">\n        {{ d }}\n      </td>\n    </tr>\n    <tfoot><tr>\n      <td colspan="7" ng-transclude></td>\n    </tfoot></tr></table> ,
```

Additionally, we'll have to tell AngularJS that our directive will be using transclusion. To do that, we'll have to set the transclude option either to:

- true, which will take the content of the directive and place it in the template
- 'element', which will take the entire defined DOM element including the lower priority directives (see notes on directive priority order below)

Set the transclude option to true, as we only want the content of our directive to show up in our template:

```
1 transclude: true
```

Next steps

We've covered how to build a directive from a very basic level to a higher level of complexity. Hopefully you've gained some confidence and knowledge about how to move forward in the future and provide and build your own custom directives.

The entire source of our calendar directive is here:

```
1 angular.module('myApp.directives', [])
2   .directive('ourCalendar', [function() {
3     // Array to store month names
4     var months = new Array('January', 'February', 'March', 'April', 'May', 'June', \
5     'July', 'August', 'September', 'October', 'November', 'December');
6
7     // Array to store month days
8     var monthDays = new Array(31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31);
9
10    // Array to store week names
11    var weekDay = new Array('Mo', 'Tu', 'We', 'Th', 'Fr', 'Sa', 'Su');
12    return {
13      require: '^ngModel',
14      restrict: 'EA',
15      transclude: true,
16      scope: {
17        ngModel: '='
18      },
19      template: '<table class="calendarTable"> \
20        <thead> \
21          <tr><td class="monthHeader" colspan="7">{{ monthName }} {{ year }}</td> \
22        </tr></thead> \
23        <tbody> \
24          <tr><td class="monthHeader" colspan="7">{{ monthName }} {{ year }}</td> \
25            <td>{{ day }}</td> \
26            <td>{{ day }}</td> \
27            <td>{{ day }}</td> \
28            <td>{{ day }}</td> \
29            <td>{{ day }}</td> \
30            <td>{{ day }}</td> \
31          </tr> \
32          <tr><td>{{ day }}</td> \
33            <td>{{ day }}</td> \
34            <td>{{ day }}</td> \
35            <td>{{ day }}</td> \
36            <td>{{ day }}</td> \
37            <td>{{ day }}</td> \
38            <td>{{ day }}</td> \
39          </tr> \
40          <tr><td>{{ day }}</td> \
41            <td>{{ day }}</td> \
42            <td>{{ day }}</td> \
43            <td>{{ day }}</td> \
44            <td>{{ day }}</td> \
45            <td>{{ day }}</td> \
46            <td>{{ day }}</td> \
47          </tr> \
48          <tr><td>{{ day }}</td> \
49            <td>{{ day }}</td> \
50            <td>{{ day }}</td> \
51            <td>{{ day }}</td> \
52            <td>{{ day }}</td> \
53            <td>{{ day }}</td> \
54            <td>{{ day }}</td> \
55          </tr> \
56        </tbody> \
57      </table> \
58    
```

```
23   <tbody>\n24     <tr><td ng-repeat="d in weekDays">{{ d }}</td></tr>\n25     <tr ng-repeat="arr in month"> \n26       <td ng-repeat="d in arr track by $index" ng-class="{currentDay: d == da\\n\ny}">\n27         {{ d | blankIfNegative }}\n28       </td>\n29     </tr>\n30   <tfoot><tr>\n31     <td colspan="7" ng-transclude></td>\n32   </tr></table>,\n33\n34   link: function(scope, ele, attrs, c) {\n35     scope.$watch(attrs.ngModel, function(date) {\n36       if (!date) date = new Date();\n37       var week_day, day_counter, i, curr_week;\n38\n39       var day = date.getDate(),\n40           month = date.getMonth(),\n41           year = date.getFullYear();\n42       scope.days_in_this_month = monthDays[month];\n43\n44       if (attrs.showshortmonth) {\n45         scope.monthName = months[month].slice(0, 3);\n46       } else {\n47         scope.monthName = months[month];\n48       }\n49\n50       scope.currentWeek = 0;\n51\n52       scope.month = {};\n53\n54       var thisMonth = new Date(year, month, 1),\n55           firstDay = new Date(thisMonth.setDate(1)).getDay(),\n56           weeksOfMonth = Math.ceil((firstDay + scope.days_in_this_month)/ 7) \\n\n+ 1;\n57\n58       scope.weekDays = weekDay;\n59       // First week\n60       curr_week = 0;\n61\n62       scope.month[curr_week] = [];\n63       for (week_day = 0; week_day < thisMonth.getDay(); week_day++) {
```

```

65         scope.month[curr_week][week_day] = week_day * -1;
66     }
67     week_day = thisMonth.getDay();
68     for (day_counter = 1; day_counter <= scope.days_in_this_month; day_coun\
69 ter++) {
70         week_day %= 7;
71         // We are at the start of a new week
72         if (week_day == 0) {
73             curr_week += 1;
74             scope.month[curr_week] = [];
75         }
76
77         scope.month[curr_week].push(day_counter);
78         week_day += 1;
79     }
80
81     while(scope.month[curr_week].length < 7) {
82         scope.month[curr_week].push(day_counter * -1);
83     }
84
85     scope.day = day;
86     scope.year = year;
87 });
88 }
89 }
```

Now to use this directive in-practice, we'll only have to tell the view where we want it placed and what to *bind* our ngModel to in the backend. To wrap all this back in the view, we'll set it as an attribute in the DOM element where we want it placed.

```

1 <div our-calendar showShortMonth="true" ng-model='newEntry.date'>
2   <h3>Today</h3>
3 </div>
```



The newEntry.date in the above example is bound to the main input field where our users will enter their todo entry.

Now that we've got the handle of building a directive under our belt, we're going to use another directive to setup a custom form validation.

Integrating form validation in our app

In our app, we have a single input field. This input field that requires that a date is filled out. Without a date, we won't be able to place an event on the calendar. In order to add a custom validation, we'll need to build a directive just as we did above.

```
1 angular.module('myApp.directives')
2   .directive('ensureHasDate', [function() {
3     // Our form validation for ensuring we have a date
4     // will go in here
5   }]);

```

To parse dates from our input field, we'll use the fantastic chrono library. To include the chrono library into our app, we'll need to download it and store it in our lib/ directory.

```
1 cd public/lib
2 curl -O https://raw.github.com/berryboy/chrono/master/chrono.min.js
```

Then we'll need to make sure we include this as a library in our index.html page. As we did previously with all of our library scripts, we'll just need to make sure that this is included at the bottom of our page:

```
1 <script src="lib/chrono.min.js"></script>
```

Now we can start to write our validation directive.

In our validation directive, we're going to use the chrono library to parse the text and find a date inside of it. If it finds a date, then we'll consider the form valid and set it as valid. If it doesn't not find a date, then we'll consider the input text to be invalid.

The first step in our directive is to *require* that the form has access to the ngModel and to create an *isolate* scope on that form validation with a binding to the parent's ngModel. In code, this looks like:

```
1 angular.module('myApp.directives')
2   .directive('ensureHasDate', [function() {
3     return {
4       require: '^ngModel',
5       scope: {
6         'ngModel': '='
7       }
8     }
9   }]);

```

Once our form gets loaded into the view, we'll want to attach a \$watch to watch for the ngModel to be populated. Remember, this is because the directive will get loaded at compile-time and ngModel won't be available until the run-time.

```

1 angular.module('myApp.directives')
2 .directive('ensureHasDate', [function() {
3   return {
4     require: '^ngModel',
5     scope: {
6       'ngModel': '='
7     },
8     link: function(scope, ele, attrs, ctrl) {
9       scope.$watch(attrs.ngModel, function(newVal) {
10         if (newVal) {
11           // We are in the runtime and ngModel has been populated
12         } else {
13           // At initialization stage
14         }
15       });
16     }
17   }
18 }]);
```

With this \$watch in place, we can now create the validation that checks if there is a date or not in the string. The chrono library will return a null object if no date is found and a Date object if one is found. Using this fact, we can check if the date that's returned is a null object.

To set the validity on the form element, we'll use the \$setValidity method on the directive's controller. This \$setValidity method will set not only the validity of the input field, but it will set it on the form in general.

```

1 angular.module('myApp.directives')
2 .directive('ensureHasDate', [function() {
3   return {
4     require: '^ngModel',
5     scope: {
6       'ngModel': '='
7     },
8     link: function(scope, ele, attrs, ctrl) {
9       scope.$watch(attrs.ngModel, function(newVal) {
10         if (newVal) {
11           // We are in the runtime and ngModel has been populated
12           var date = chrono.parseDate(newVal);
13           ctrl.$setValidity('hasDate', date !== null);
14         } else {
15           // Without a date, the form is always invalid
16           ctrl.$setValidity('hasDate', false);
17         }
18       });
19     }
20   }
21 }]);
```

```
17      }
18    });
19  }
20 }
21 })];
```

The `$setValidity` method will take care of telling AngularJS that the form is invalid. It will also let AngularJS set the proper classes for both the invalid form as well as the form itself.