AngularJS IN ACTION

Brian Ford
Lukas Ruebbelke

MEAP

MANNING

MEAP Edition
Manning Early Access Program
AngularJS in Action
Version 4

For more information on this and other Manning titles go to
www.manning.com

brief contents

1 Introduction - AngularJS

5 key points:

- AngularJS is a client-side JavaScript framework
- It's used to build web applications
- It is prescriptive
- Makes creating a user interface (UI) easier through data-binding
- It helps organize and architect an application

If you are reading this book, you probably know something about AngularJS. You might've heard that it has custom elements and attributes, or it uses dependency injection. At very least you're curious. In short, AngularJS is a framework that helps you build front-ends for web-based applications. In this chapter, we're first going to describe AngularJS by defining some key terms and explaining AngularJS's relation to them. Then we'll look at the benefits and dig into a hello world example. By the end of the chapter you should have a pretty good idea of what AngularJS does and what role it can play in your software projects. All of what you learn in chapter 1 will set the stage for when we introduce our running example in chapter 2.

1.1 What is AngularJS?

In a sentence, AngularJS is a prescriptive client-side JavaScript framework used to make single-page web apps. That's a mouthful! Let's talk break it down piece by piece. AngularJS is prescriptive in the sense that it has a recommended way of building web apps. It has its own spin on the ubiquitous Model View Controller (MVC) pattern that is especially well suited for JavaScript. In this way, Angular is prescribing a way for you to divide your application up into smaller parts. AngularJS is a framework that runs in the web browser. A framework (as opposed to a library) has some overarching idea of how an app should work, and typically handles things like starting the app up and navigating to different parts of the app. AngularJS is a JavaScript framework because (unremarkably) it is used to write apps in JavaScript. If you are comfortable writing JavaScript, you should be right at home using AngularJS. Beyond that, we said AngularJS is for single page apps, meaning the browser only loads the page once, but then makes asynchronous calls to the server to fetch new information. Here asynchronous means that the application can continue to function without having to stop and wait for the server to respond.

In this section we'll look at the features of AngularJS and talk about the benefits to using Angular over other approaches in building web apps. We're going to expand on these ideas in 1.1.1 as we talk about the core features of AngularJS and why they are important. We'll also walk through a super basic AngularJS app to get your feet wet.

1.1.1 Core Features

As we kick off the AngularJS tour, we are going to visit some of the most important features so that you will have a familiarity of all the pieces we are going to use to make the awesome sample application in the rest of the book.

Table 1.1 Core Features of AngularJS

Two Way Data-binding	Model View Whatever
HTML Templates	Deep Linking
Dependency Injection	Directives

TWO WAY DATA-BINDING

Let's say you have a form to store user contact info and you want to update the addresses input box when data is returned from the server. You could do the following:

Listing 1.1

```
<span id="yourName"></span>
```

Listing 1.2

```
document.getElementById('yourName')[0].text = 'bob';
```

You manually seek to the element you want to change, calling a long, awkward series of DOM methods. This is brittle and can require some heavy lifting depending on how complex your DOM is. You also have to make this call every time something causes the address to change. Rather than spend your time worrying about these details, you could instead use AngularJS's two-way data binding. If we take the same example above and use data-binding we accomplish the same thing but faster. Here's our example with data-binding:

Listing 1.3

```
<span>{{yourName}}</span>
```

Listing 1.4

```
var yourName = 'bob';
```

In this approach you avoid repeating yourself, or having to worry about when some state changes and the form needs to be updated. It all happens automatically, thanks to AngularJS. This means that there is only one place in your application where that piece of information is stored. When that data changes, everything that is displaying that information just needs to update to reflect the new value. This is where two way data-binding comes in! It makes it easy to change a value and effortlessly have it update the DOM.

MODEL VIEW WHATEVER

If you're an experienced web developer, you're likely familiar with the Model View Controller (MVC) design pattern. If you've never heard of MVC, don't worry about it, the idea is quite simple. MVC is a pattern for dividing an application into different parts (called Model, View, and Controller), each with distinct responsibilities. There are many different variants on MVC, depending on how responsibilities are divided between parts. Two examples of popular variants are Model View Presenter (MVP) and Model View ViewModel (MVVM).

Although AngularJS's architecture closely resembles MVC and its variants, the way in which AngularJS maps responsibilities to parts doesn't perfectly fit any of them. Rather than become embroiled in a debate about the philosophical nuances of design pattern architecture, the AngularJS team humorously refers to AngularJS as a "Model View Whatever" framework.

We have personally found that the Model View ViewModel (MVVM) pattern is a close enough approximation for a general illustration of how AngularJS works. Take a look at Figure 1.1 to get a better picture of how MVVM works. In the diagram you can see that app.html serves as the View and is bound to ViewModel which is app.js. If someone were to change the name property on `$scope.user` in app.js, it would automatically update the property in the name field in app.html. The converse is true in that if someone were to change the name field it in app.html, it would automatically update the `$scope.user` object. App.html can also issue commands to app.js such as calling `$scope.save` and letting app.js perform some unit of logic such as saving the `$scope.user` object to users.js so that it can be shared elsewhere in the application.

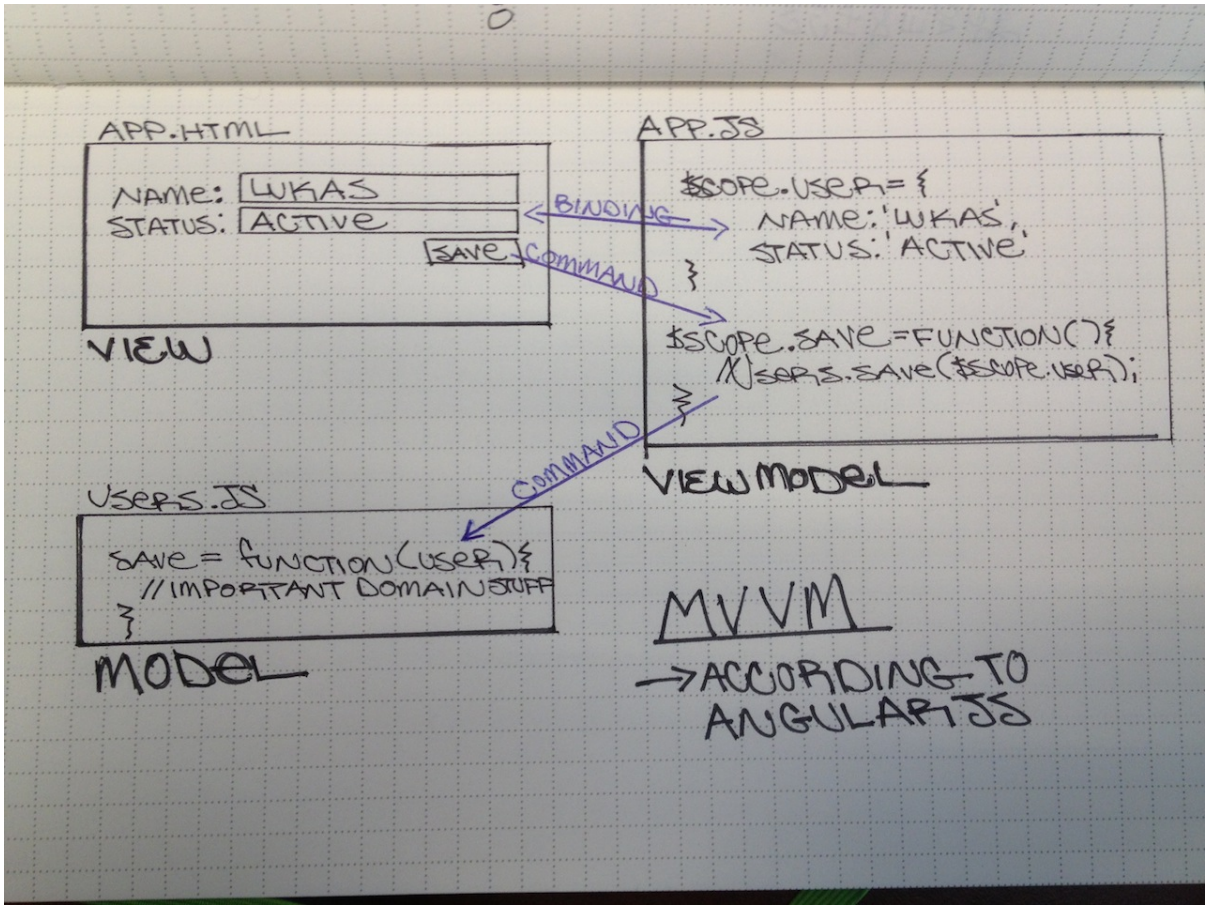

Figure 1.1 MVVM According to AngularJS

The MVVM pattern works really well for applications involving a rich user interfaces because the View is bound to the ViewModel and when the state of the ViewModel changes the View will update itself automatically. This is possible because we have two way databinding built into the framework.

In AngularJS, the View is simply your HTML with the AngularJS syntax compiled into it. Once the initial compilation cycle completes, the View can then bind to the `$scope` object which is essentially the ViewModel. We will get into this later but `$scope` is just a JavaScript object with some eventing attached to it to enable two way data-binding. You can also issue commands to the ViewModel to perform some operation as well.

We will elaborate more on each part later in the book but this should give you the general idea how AngularJS is constructed. This pattern creates a clean separation between the View and the logic driving the View. One of the most profound side-effects of the ViewModel pattern is that it makes your code very testable.

HTML TEMPLATES

Another core feature of AngularJS is that it uses HTML for templates. HTML templates are useful when you want to predefine a layout with dynamic sections to be filled in it is connected with an appropriate data structure. An example of this would be if you wanted to repeat the same DOM element over and over in a page such as a list or a table. You would define what you want a single row to look like and then attach a data structure, such as a JavaScript array, to it. The template would repeat each row for as many items in the array filling in each instance with the values of that current array item.

AngularJS has some really nice features that are going to make designer types feel right at home. It has a templating system that is not built on top of HTML but is HTML. It allows web designers to bring their current skill set to the table and start developing immediately. No new syntax. No pre-processing. Just HTML.

There are plenty of templating libraries out there but most of them require you learn a new syntax. This additional complexity can really slow down new developers. Furthermore, these other templates often need to be run through a pre-processor.

AngularJS templates are HTML and are validated by the browser just like the rest of the HTML in the page. There is some clever AngularJS functionality built in that allows you to control how your data structure is rendered within the HTML template. We will cover this in depth in a later chapter.

DEEP LINKING

AngularJS is a library for single-page apps, but there's a good chance your users won't even notice (besides being pleasantly surprised at the speed). While many modern web apps "break the back button," AngularJS makes it easy for you to update the URL of your application based on what the user is doing. Angular will use either the HTML5 history API, or fallback to hasbang URLs when the HTML5 history API is not available.

What this means for users is that they can bookmark and share application state. While this seems like a small convenience, it's actually immensely powerful in the age of social media. It also makes changing application state easy for you, the developer -- you can use good 'ol hyperlinks to navigate the user around your application.

DEPENDENCY INJECTION

Dependency injection (DI) is a really fancy phrase that describes a technique you have been using from when you first started programming. If you have ever defined a function that accepts a parameter you have leveraged dependency injection. You are injecting something that your function depends on to complete its work. It is that simple.

DI encourages you to write code that clearly describes its dependencies. This is handy because it makes your code to test and debug. It's easier to test because you can pass in a replacement for any dependency, as long as it provides the same interface. This could be a real object that you have constructed or a mock object that simulates the object. It is easier to debug because it encourages you to write small units of code which is easier to step through because of its granular nature.

In AngularJS, in most cases, gaining access to a particular component in the application is simply a matter of declaring it is a variable of the method you are using to define the element that needs that component.

Let's look at a simple case where this behavior can help us reduce ordering constraints, or the requirement that we explicitly define when parts of the application get loaded when. One simple case is that we conditionally want to swap out one function's implementation for an implementation in a different file. Consider this code:

Listing 1.5 Defining functions without DI

```
function a () {
    return 5;
}
function b () {
     return a() + 1;
}
console.log(b());
```

Is there some way to load a file either before or after we load the above to swap out the implementation of function a? Maybe, but it takes some tricks. You've also introduced another level of coupling: ordering constraints. Functions cannot easily be moved around. This will make it harder and harder to add new parts to an app as you write it. Now consider this code using DI:

Listing 1.6 Defining functions with DI

```
service('a', function () {
    return 5;
});
service('b', function (a) {
    return a() + 5;
});
service('main', function (b) {
    console.log(b());
});
```

There are several practical benefits to this change. The first is that functions can be moved around in any order. This may seem small, but for larger applications it can save valuable time. Next, you can easily override any of the functions, like this:

Listing 1.7 Overriding with DI

```
// swap out 'b' easily by redefining it in another file:
service('b', function (a) {
    return 1; // this is a test value
});
```

This is important in unit testing, or any instance where you want to have a drop-in replacement for some part of your code.

DIRECTIVES

Directives are our favorite part of AngularJS because it allows you to extend HTML to do some really powerful things. By "some really powerful things" we mean "pretty much anything you can imagine."

You can create custom DOM elements, attributes, or classes that attach functionality that you define in JavaScript. If you've worked with Flex before, you might recognize that this feature is similar to how you would define your behavior in an ActionScript class, then instantiate the class in MXML. HTML is excellent for declaring layout but other than that it is inherently dumb. Directives provide the perfect way to merge the declarative nature of HTML and the functional nature of JavaScript into one place.

Let's say you're writing an application that has a dropdown. Normally, you have some weird, difficult to remember class hierarchy like this:

Listing 1.8

```
<div class="container">
  <div class="inner">
    <ul>
      <li>Item
        <div class="subsection">item 2</div>
      </li>
    </ul>
  </div>
</div>
```

You can use a directive to make a shortcut so you only have to type:

Listing 1.9

```
<dropdown>
  <item>Item 1
    <subitem>Item 2</subitem>
  </item>
</dropdown>
```

But more than that, you can attach animations, behaviors, and more to this dropdown, and have all of these features included in your app just by writing `<dropdown></dropdown>`.

With our core intact, we want to point out what the benefits are of AngularJS!

1.1.2 Benefits

With our core in tact, we want to point out what the benefits are of AngularJS. We couldn't introduce you to Angular without showing you how it can help you in your day-to-day work. By looking at how Angular can help you, you'll be able to see where could you start using it, and how to leverage these strengths in your projects.

TESTABLE

How do you REALLY know your app works? The answer is "by writing tests," but it's often more complicated than that.

Remember when we said that AngularJS has distinct parts? One of the main driving forces between dividing the responsibilities between parts like this is so

that they would be easy to test in isolation. Being able to test parts of your app in isolation means that if something does break, you can quickly pinpoint exactly where.

Tests mean you can refactor, add new features, or make optimizations with confidence that everything still works. Web standards and browsers are moving fast these days. Features pop in and out, specs and APIs change. Tests mean you can ensure that your app works with the latest versions of your user's web browsers, even if you don't religiously follow every change to every browser.

If testing is easy, you're more likely to do it. In turn, this makes your application easier to maintain. AngularJS makes it easy to write unit tests. Mocking out parts of your application, you can test each feature in isolation. Angular also makes it easy to write integration tests that ensure your whole stack is functioning as expected.

As we walk through example applications, we'll show you how to write tests for each part.

EMBEDDABLE

AngularJS is not an "all or nothing" proposition. Imagine you have an existing website that is built entirely on jQuery and you want to use AngularJS on just single feature in the page, this is not a problem at all. It is easy to embed AngularJS into an existing website with as much surface area as you wish.

You can simply attach the AngularJS application a specific DOM element and not have to worry about nasty side effects outside of that element and into the rest of the page. This makes it very easy to get started with AngularJS and sets up a nice path to refactor legacy applications into something more manageable.

JUST JAVASCRIPT

AngularJS embraces JavaScript. While this may seem like an obvious thing for a JavaScript framework to do, other frameworks sometimes try to abstract the JavaScript away. In practice, this means you don't have to memorize pages and pages of APs or extend some deep class hierarchy to reap the benefits of AngularJS. Just plain old JavaScript here. This makes your code easy to test, maintain, reuse, and again free from boilerplate.

Because AngularJS is just JavaScript, you can easily use it with any language that compiles to JavaScript if you prefer. This includes CoffeeScript, TypeScript, ClojureScript, and any of the other compile-to-JavaScript languages yet to come.

A quick recap: AnguarJS has many benefits and features. If you are were to

pick three things to remember about it, they are data-binding, testability, and "just JavaScript." Or consider the old way of writing web-apps with event handlers and DOM mutations, and think of how much easier it would make your development process to use Angular instead.

1.2 Hello Angular

The traditional "Hello World" program is a bit too simple to show off AngularJS features. Instead, we're going to make a "Hello ___" program, where we have an input box that fills the blank in with whatever we type into the box.

This is what our "hello world" will look like:

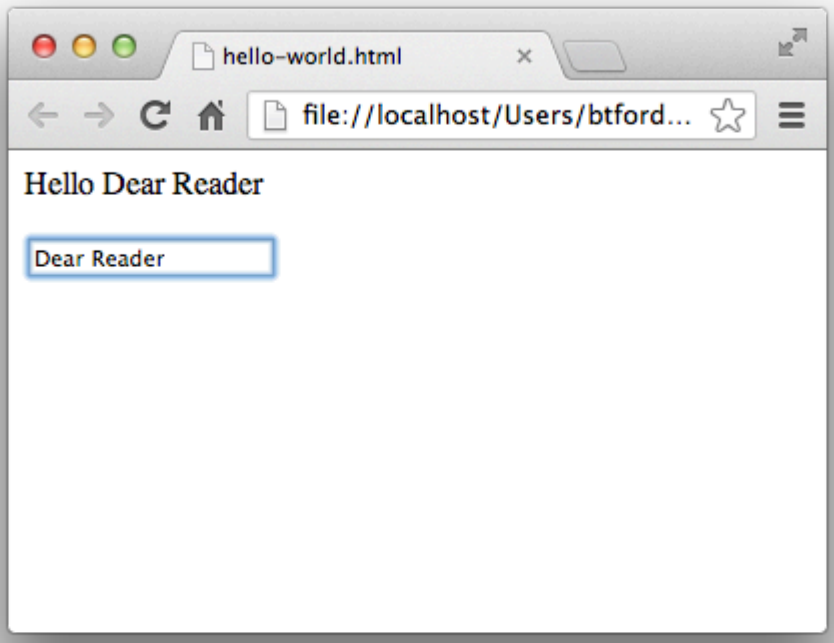

Figure 1.2 A screenshot of "Hello ___"

As mentioned before, as you type into the input box, The line "Hello ___" gets filled in with the contents of the input box. One thing that might surprise you coming from jQuery or another framework is that it changes letter-by-letter. This behavior is a direct result of AngularJS's data-binding.

1.10 shows what the code looks like for this "hello world."

Listing 1.10

```
<body ng-app="helloApp">
  <div ng-controller="HelloCtrl">
    <p>{{greeting}} {{person}}</p>
    <input ng-model="person">
  </div>
  <script>
    angular.module('helloApp')
      .controller('HelloCtrl', function ($scope) {
        $scope.greeting = 'Hello';
        $scope.person = 'World';
      });
  </script>
</body>
```

(1)(1) These are organizational parts of Angular
(2) ng-model sets up data-binding to {{greeting}} and {{person}}

Note the special attributes in the HTML: "ng-app," "ng-controller," and "ng-model." These are all things that AngularJS uses to do its "magical" data-binding. The first line of JavaScript and the first two lines of HTML are organizational parts of Angular. We'll talk about how what controllers and modules do in the next chapter, but the short version is that they help organize and namespace your code.

The main thing we want to draw your attention to is `ng-model` and the "double curly brace" variables, like `{{greeting}}` and `{{person}}`. As you probably noticed, there's a correspondence between the second two lines of JavaScript, and the text between double curly braces. That is, the value of $scope.greeting gets put into `{{greeting}}`. In the case of `{{person}}`, the value is initially set to "world," but then constantly updated to reflect the value of the input box. This is thanks to the `ng-model` attribute of the input box.

Try running this example yourself. What happens when you create other `ng-model` input boxes?

1.3 Summary

Let's quickly recap. AngularJS is a framework used to build dynamic, single page web applications. The biggest feature AngularJS brings to the table is data-binding, which makes synchronizing application data and UI state fast and easy. The most important general benefits of using AngularJS are that it improves the organization and robustness of your app.

Now that we have covered the major features and benefits of AngularJS, we are going to spend the next chapter going over a very simple app. We are also going to

cover a few high level best practice consideration and take our first pass at the code.

See you in 3 - 2 - 1.

2 Hello Angular

Objectives:

1. Build something simple with AngularJS
2. Build a subset of the main project that we can reuse later
3. Provide a gentle introduction to getting up and running

2.1 Getting to Know AngularJS

Our goal is to build a substantial application by the end of the book but getting into that now would be like asking you to marry AngularJS on the first date. We are going to shoot for puppy love in this chapter by showing you how easy it is to get up and running with AngularJS. Once we rock your socks off with about 50 lines of JavaScript, we can talk about a long term relationship in Chapter 3.

2.1.1 Introducing Angello Lite

There is a popular project management web application called Trello that is the stuff legends are made of. We are going to be building a homage to Trello in AngularJS called Angello. See what we did there?

Angello is an agile software project board that follows the principles of agile with user stories, acceptance criteria, swim lanes, etc. We are going to focus solely on the user stories aspect of the application and so that we can get to market sooner we are going to call it Angello Lite.

We are going to be covering enough of AngularJS for identifiable patterns to emerge so that you can use them in other projects. We are also going to be able to pick up right where we left off in the next chapter and continue to add features to

this all the way to the end of the book. So not an ounce of wasted effort here!

Below you can get a quick glimpse of what Angello Lite will look like in Figure 2.1.

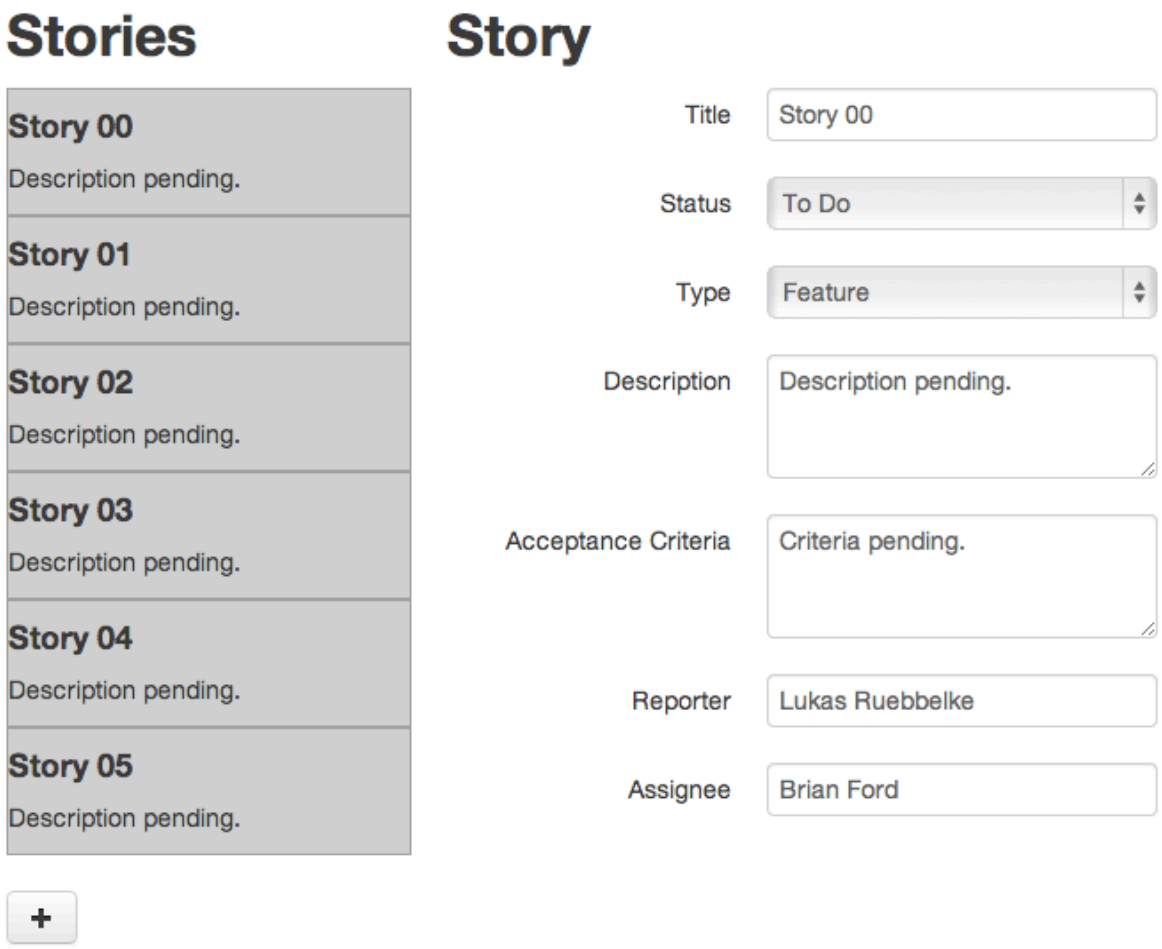

Figure 2.1 Angello Lite

2.2 Round One: Welcome to the Party

Now that we have you in a state of frenzy with anticipation it is time to hold up our end of the bargain.

First, create a directory you want to put your files in. And from there create three files called index.html, app.js and app.css. We are going to presume that you have a fundamental knowledge of HTML and so we are not going to get into basic HTML document structure and how to include JavaScript and CSS files into your document.

We are also going to try to structure our instructions in a way that one step is not dependent on the next step if possible. This reduces the need for you to have to keep more than one step in your short term memory stack but if we need to we will

call it out.

2.2.1 An AngularJS App That Does Nothing In Two Easy Steps

So the very first thing we need to do to get an AngularJS app going is to define the module that you want your application to live in. A module is simply a place where you can define and collect the different pieces of code you need in your app. So below, we are defining a module called Angello.

Listing 2.1 STEP 1 app.js

```
var myModule = angular.module('Angello', []);
```

Then we need to initialize the AngularJS application and the easiest way to do that is by bootstrapping the application with the ng-app. You will notice that we are passing in an attribute Angello to indicate that we want to bootstrap with that module.

Listing 2.2 STEP 2 app.html

```
<html ng-app="Angello">
```

NOTE: An AngularJS application exists at the DOM element that you add the ng-app directive. I generally put it in the html tag but you could easily put it in a div tag somewhere in your page if you wanted to isolate it. If your application is just not working make sure you haven't put ng-app in the head tag as it will bite you like it has bitten me!

Technically, you now have a working AngularJS application. Life is good!

2.2.2 Wiring Up the Controller and View

Lets actually puts some wheels on this thing! The very next thing that needs to happen after an AngularJS application has been bootstrapped is a controller needs to be defined. The controller will be responsible for hooking our models and business logic into our views.

We are going to define a MainController for our application like so.

Listing 2.3 STEP 3 app.js

```
myModule.controller('MainCtrl', function($scope) { });
```

NOTE: You will notice that the second parameter of this function call is an anonymous function with a $scope parameter. $scope is the lifeblood of AngularJS. We will get into the molecular structure of how $scope works later but for now all you need to know is that $scope is where you put your methods and properties that you want to to bind to in your view.

And now we are going to assign the MainCtrl to a our page via ng-controller.

Listing 2.4 STEP 4 app.html

```
<body>
  <div ng-controller="MainCtrl"></div>
</body>
```

We now have a controller in place to start adding properties and behavior, such as the models representing user stories, to our view. Now that we have a skeleton and the nervous system in place, it is time to start putting some meat on them bones!

2.2.3 Displaying the Stories

We want to display our user stories in the view. I have taken the liberty of mocking up some user stories so that we have something to work with. I am creating a stories property on $scope and initializing it with an array of objects that have a title and description property.

Listing 2.5 STEP 5 app.js

```
myModule.controller('MainCtrl', function($scope) {
  $scope.stories = [
            {title:'Story 00', description:'Description pending.'},
            {title:'Story 01', description:'Description pending.'},
            {title:'Story 02', description:'Description pending.'},
            {title:'Story 03', description:'Description pending.'},
            {title:'Story 04', description:'Description pending.'},
            {title:'Story 05', description:'Description pending.'}
        ];
});
```

The amusing thing about this is that our dummy data is more lines of code than the code to actually get it on the page. Let me prove my point.

We are going to use an AngularJS directive called ng-repeat. What ng-repeat essentially does is repeat the DOM element that you put it on as many times as

there are items in the collection you send it. Within that DOM element you are allowed to create dynamic elements using double curly brace markup. As we saw a bit earlier, these curly braces tell AngularJS to data bind models to the view.

In this specific case we are using ng-repeat to iterate over the stories array on $scope. We want to dynamically display the title and description of each story and thus the {{story.title}} and {{story.description}} in the h4 and p tags respectively.

Listing 2.6 STEP 6 app.html

```
<body>
  <div ng-controller="MainCtrl">
    <div class="span4 sidebar-content">
      <h2>Stories</h2>
      <div class="story" ng-repeat="story in stories”>
        <h4>{{story.title}}</h4>
        <p>{{story.description}}</p>
      </div>
    </div>
  </div>
</body>
```

And what have we built? If you have been following along you should have something that looks similar to Figure 2.2.

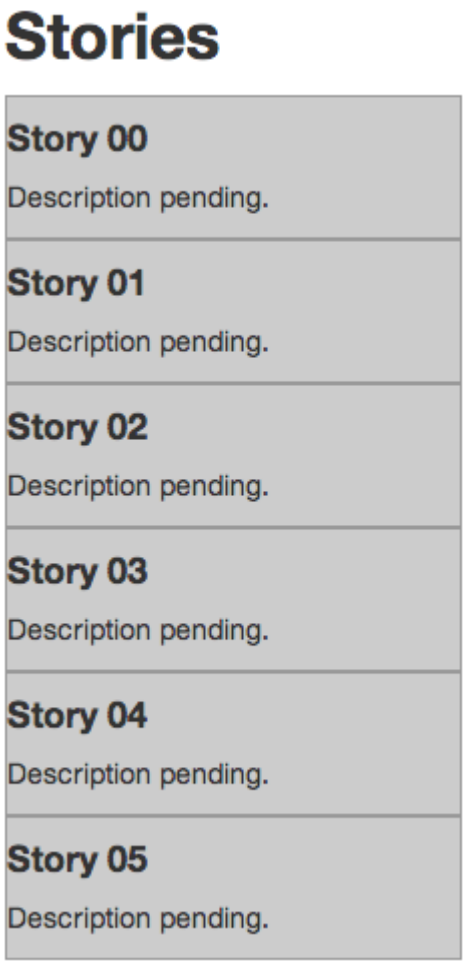

Figure 2.2 The stories

2.2.4 Displaying a Story's Details

So this is pretty awesome! We are displaying all of our stories with just a few lines of code but lets take this a step further and display an individual story's details so that we can view and edit them.

We are going to create another property on $scope called currentStory and that is going to be responsible for keeping track of our current story. We are also going to create a method on $scope call setCurrentStory which is going to accept a story parameter and assign it to the currentStory property.

Listing 2.7 STEP 6 app.js

```
myModule.controller('MainCtrl', function($scope) {
    $scope.currentStory;

    $scope.setCurrentStory = function(story) {
        $scope.currentStory = story;
    };

    // CODE OMITTED
});
```

NOTE: It almost feels silly typing out our code and explaining what it does because we strive to write code that is "self documenting". It is pretty obvious what setCurrentStory does because it has a name that clearly indicates its responsibility. This is old news to seasoned programmers but if you are new to the game it is important to get in the habit of doing this.

As soon as a method is defined on $scope it is ready to be called from the view and so let's wire that up now. We are going to add the ng-click directive to our HTML and called the setCurrentStory method with the current story that is in the ng-repeat loop.

At first glance the code below is deceptive in its simplicity but the interesting thing is that AngularJS is keeping track of which story object that is being passed in when ng-click fires. Therefore, we can reuse the same methods exposed on our scope, but with different models depending on the context. This is one of those things in AngularJS that "just works" and we are so glad it does!

Listing 2.8 STEP 7 index.html

```
<div class="story"
  ng-repeat="story in stories"
  ng-click="setCurrentStory(story)">
    <!-- CODE OMITTED -->
</div>
```

And now that we are setting the current story this is where two-way databinding is going to shine! We are going to add another div to our markup and add the following code to it. This is a pretty standard form for setting the title and

description of a story. With one small and very important exception. We have included an ng-model directive and set it to currentStory.title and currentStory.description for their respective inputs.

Listing 2.9 STEP 8 index.html

```
<div class="span6 body-content">
  <h2>Story</h2>
  <form class="form-horizontal">
    <div class="control-group">
      <label class="control-label" for="inputTitle">Title</label>
      <div class="controls">
        <input type="text"
          placeholder="Title"
          ng-model="currentStory.title">
      </div>
    </div>
    <div class="control-group">
      <label class="control-label"
        for="inputDescription">Description</label>
      <div class="controls">
        <textarea id="inputDescription"
          placeholder="Description"
          rows="3"
          ng-model="currentStory.description">
        </textarea>
      </div>
    </div>
  </form>
</div>
```

So what happens when you click on a story in the list and that story is set to the $scope.currentStory? You guessed it! The inputs on the form is going to automatically update to reflect the values of the story you just selected.

Wait! What?! Where are all the complicated event listeners and DOM manipulation that we had to write to make this work? Gone! That's where! In most web applications up to this point, the majority of code written centered around trying to read and update the DOM based on user input. It is okay to weep from joy from not having to do that anymore. We did as well.

So what happens if we want to get a value back out of the form? For instance, what if you want to send values in the form back to the server? That is a good question. The story object on $scope automatically updates to the new values. Don't believe me? Try it! Select a story and change the title and description to something of your choosing. Did you see it automatically updated in the list? Pretty cool huh!?

2.2.5 Creating a New Story

It is pop quiz time! We have added a few properties and methods to our application so based on that information, how would you add a method to create a new story? Before you jump ahead and check the answer, think about how you would call that method in the view.

The solution is to add a createStory method to $scope and then call it via ng-click in the view. For now, we are going to push a new story object into stories array and that is all there is to it. You will see a new story automatically show up in the stories list in the view.

Add the createStory method? Check!

Listing 2.10 STEP 9 app.js

```
myModule.controller('MainCtrl', function($scope) {
  $scope.createStory = function() {
    $scope.stories.push({
      title:'New Story',
      description:'Description pending.'
    });
  };

  // CODE OMITTED
});
```

Call the createStory method? Check!

Listing 2.11 STEP 10 index.html

```
<div class="story"
  ng-repeat="story in stories"
  ng-click="setCurrentStory(story)">
<!-- CODE OMITTED -->
</div>
<br>
<a class="btn" ng-click="createStory()">
  <i class="icon-plus"></i>
</a>
```

Did we just add a new feature to our application in like two minutes? Yep! Keep this up and you are on your way to becoming a certified web application developer!

2.3 Round Two: Let's Kick Up The Jams!

So technically we have a fully functional and useful application. I would imagine by now you are looking around thinking this is a trap because it was just too easy. Let's add a few more features since we got done way ahead of schedule and we have some time to kill.

2.3.1 Spice Up the Story

We are going to expand our user story object to include a few more properties to make it a bit more robust. The two main properties we are going to focus on for the rest of the chapter are the status and type property. The reason we are going to focus on these is because they are going to involve introducing two new arrays and what it takes to work with multiple collections.

The big question that we are going to cover in this portion is how to take a property from an object in one array and correlate that to another object in a different array. So let's make another lap around the track!

Listing 2.12 STEP 11 app.js

```
{
  title:'Story 00',
  description:'Description pending.',
  criteria:'Criteria pending.',
  status:'To Do',
  type:'Feature',
  reporter:'Lukas Ruebbelke',
  assignee:'Brian Ford'},
```

The criteria, reporter and assignee are just text fields and so we can add those to the form and be done with those quite quickly. Notice that we are adding the HTML and then just wiring the input fields with ng-model.

Listing 2.13 STEP 12

```
<div class="span6 body-content">
  <h2>Story</h2>
  <form class="form-horizontal">
    <div class="control-group">
      <label class="control-label" for="inputTitle">Title</label>
      <div class="controls">
        <input type="text"
          id="inputTitle"
          placeholder="Title"
          ng-model="currentStory.title">
      </div>
    </div>
    <div class="control-group">
      <label class="control-label"
        for="inputDescription">Description</label>
      <div class="controls">
        <textarea id="inputDescription"
          placeholder="Description"
          rows="3"
          ng-model="currentStory.description"></textarea>
      </div>
    </div>
    <div class="control-group">
      <label class="control-label"
        for="inputAcceptance">Acceptance Criteria</label>
      <div class="controls">
        <textarea id="inputAcceptance"
          placeholder="Acceptance Criteria"
          rows="3"
          ng-model="currentStory.criteria"></textarea>
      </div>
    </div>
    <div class="control-group">
      <label class="control-label" for="inputReporter">Reporter</label>
      <div class="controls">
        <input type="text"
          id="inputReporter"
          placeholder="Reporter"
          ng-model="currentStory.reporter">
      </div>
    </div>
    <div class="control-group">
      <label class="control-label" for="inputAssignee">Assignee</label>
      <div class="controls">
        <input type="text"
          id="inputAssignee"
          placeholder="Assignee"
          ng-model="currentStory.assignee">
      </div>
    </div>
  </form>
</div>
```

2.3.2 Add in Types and Statuses

We are going to introduce two new collections into our application called types and statuses. We are going to populate these arrays with simple objects that we will use in just a moment.

Listing 2.14 STEP 13 app.js

```
myModule.controller('MainCtrl', function($scope) {
    $scope.statuses = [
            {name:'Back Log'},
            {name:'To Do'},
            {name:'In Progress'},
            {name:'Code Review'},
            {name:'QA Review'},
            {name:'Verified'},
            {name:'Done'}
        ];
    $scope.types = [
            {name:'Feature'},
            {name:'Enhancement'},
            {name:'Bug'},
            {name:'Spike'}
        ];

// CODE OMITTED
});
```

Once again, our dummy data takes up more lines of code than our actual code. It is awesome being awesome.

So now we are going to introduce a directive similar to ng-repeat that is designed specifically for a select control. The directive in question is ng-options and it works almost identical to ng-repeat except you are going to be a little more precise in its declaration. There are a few ways to use ng-option tags, but the most common is the "label for value in array" format. So for instance, we are saying “use the name property on the status object in the statuses array”.

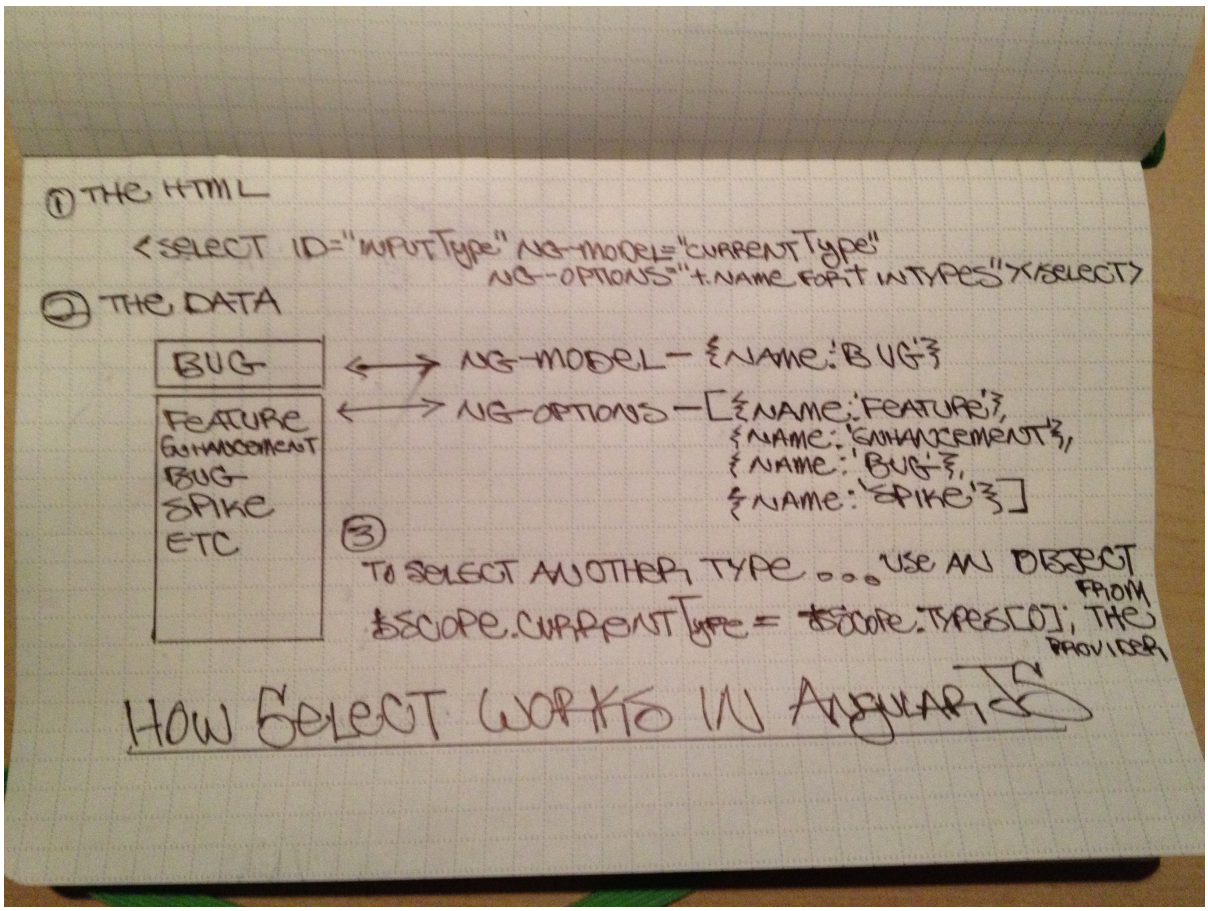

Figure 2.3 How select works in AngularJS

Listing 2.15 STEP 14 index.html

```
<div class="span6 body-content">
  <h2>Story</h2>
  <form class="form-horizontal">
    <!-- CODE OMITTED -->
    <div class="control-group">
      <label class="control-label" for="inputTitle">Title</label>
      <div class="controls">
        <input type="text"
          id="inputTitle"
          placeholder="Title"
          ng-model="currentStory.title">
      </div>
    </div>
    <div class="control-group">
      <label class="control-label" for="inputStatus">Status</label>
      <div class="controls">
        <select id="inputStatus"
          ng-model="currentStatus"
          ng-options="s.name for s in statuses"></select>
      </div>
    </div>
    <div class="control-group">
      <label class="control-label" for="inputType">Type</label>
      <div class="controls">
        <select id="inputType"
          ng-model="currentType"
          ng-options="t.name for t in types"></select>
      </div>
    </div>
    <!-- CODE OMITTED -->
  </form>
</div>
```

This now populates the select controls with the statuses and types array respectively. The next thing that we need to do is coordinate the statuses and types collections with the stories collection.

We are also going to set ng-model to currentStatus and currentType respectively so that we can programmatically update the select controls by setting those properties to the objects we want.

We are going to go on a small adventure for a minute and create a helper function to make things easier down the road. When you have an array of objects and you need to find a specific object based on a value, the most obvious approach would be to loop over that array until you find what you were looking for and move on to the next step. This is valid but it can get very resource intensive and it

is inefficient having the loop over the same collection over and over every time you need an object. When faced with this problem, I always create an index that keeps track of my objects by the property that I want to look them up by. In JavaScript this is nothing more than a named array with the format of myArray[object.property] = object. This way I can find my object simply by referencing it via myObject = myArray[someProperty].

You can see that I have create a helper function called buildIndex that takes a source array and the property you want to use to lookup the object and returns a named array primed for lookup.

NOTE: You may have noticed that I have not attached buildIndex to the $scope object. This is because I wanted to keep this method internal and not unnecessarily expose it to the view.

Listing 2.16 STEP 15 app.js

```
myModule.controller('MainCtrl', function($scope) {
  var buildIndex = function(source, property) {
      var tempArray = [];

      for(var i = 0, len = source.length; i < len; ++i) {
          tempArray[source[i][property]] = source[i];
      }

      return tempArray;
  };

    $scope.typesIndex = buildIndex($scope.types, 'name');
    $scope.statusesIndex = buildIndex($scope.statuses, 'name');

    $scope.setCurrentStory = function(story) {
        $scope.currentStory = story;

        $scope.currentStatus = $scope.statusesIndex[story.status];
        $scope.currentType = $scope.typesIndex[story.type];
    };

  // CODE OMITTED
});
```

For the sake of efficiency I am going to explain what is happening by focusing solely on the status feature. The principles are exactly the same for types and so I am going to extend the hand of trust and leave it up to you to draw the parallels.

We are going to build out an index called statuesIndex for the statuses array so that we can look up an individual status object with no looping. We do this by

calling the buildIndex method and passing in the statuses array and using the name property to index it the array.

Now it is time to put those indexes to good use and we do that in setCurrentStory. When someone selects a story, we need to find the appropriate status object by comparing the status property on the story to the name of the status objects in the status collection. And this is where the indexes earn their lunch money! Notice the incredibly concise syntax we are left with in $scope.currentStatus = $scope.statusesIndex[story.status].

Now when you select a story, the select controls will update to the proper values because they can locate the appropriate corresponding objects that exists in their individual data sources. We are able to do this quickly because we are using an index and not looping over and over on the collections themselves.

So now we are updating the form to the existing status and type for the story but what if we want to update it? That is the fair question and the next logical step. Let us create a method called setCurrentStatus on $scope that takes a status object as its parameter.

This method is pretty straightforward as it makes sure that currentStory is defined before trying to access it and then it sets the status on currentStory to the name of the status object.

Listing 2.17 STEP 16 app.js

```
myModule.controller('MainCtrl', function($scope) {
    // CODE OMITTED

    $scope.setCurrentStatus = function(status) {
        if(typeof $scope.currentStory !== 'undefined') {
            $scope.currentStory.status = status.name;
        }
    };

    $scope.setCurrentType = function(type) {
        if(typeof $scope.currentStory !== 'undefined') {
            $scope.currentStory.type = type.name;
        }
    };

    // CODE OMITTED
});
```

And then we call it that method via ng-change and send in the currentStatus object. The ng-change directive is similar to ng-click but is appropriate for controls

that involve selection.

Listing 2.18 STEP 17 index.html

```
<div class="control-group">
  <label class="control-label" for="inputStatus">Status</label>
  <div class="controls">
    <select id="inputStatus"
      ng-model="currentStatus"
      ng-options="l.name for l in statuses"
      ng-change="setCurrentStatus(currentStatus)"></select>
  </div>
</div>
<div class="control-group">
  <label class="control-label" for="inputType">Type</label>
  <div class="controls">
    <select id="inputType"
      ng-model="currentType"
      ng-options="t.name for t in types"
      ng-change="setCurrentType(currentType)"></select>
  </div>
</div>
```

And that completes round two of our application. Most of what we have covered is just familiar variations of the same theme.

2.4 Round Three: Clean Up and Pro Tips

We just cannot help ourselves. We are going to do one more lap and prep the application to be ready for extension for later chapters. This next section is not going to visibly change how the application works but it is going to illustrate some good techniques for building AngularJS applications that can scale.

2.4.1 Create a Helper Service

The buildIndex function is a pretty generic function and it is highly likely that I would want to use it somewhere else in the application. We suggest that general purpose functions be pulled out into a helper service so that any controller can use it.

Services can be created by calling the factory method on your AngularJS module and passing it a name and a factory function as I have done below. I am calling this service angelloHelper and exposing the buildIndex method via the revealing module pattern.

Listing 2.19 STEP 18 app.js

```
myModule.factory('angelloHelper', function() {
    var buildIndex = function(source, property) {
        var tempArray = [];

        for(var i = 0, len = source.length; i < len; ++i) {
            tempArray[source[i][property]] = source[i];
        }

        return tempArray;
    };

    return {
        buildIndex: buildIndex
    };
});
```

// NOTE The revealing module pattern is a variation of the module pattern which you can read about here http://addyosmani.com/resources/essentialjsdesignpatterns/book/#revealingmodulepatter

2.4.2 Create a Model Service

We are also going to extract the dummy data to another service that will hold our domain model. This is really useful because in doing so, we are completing hiding the implementation details of where we are getting our data. We will replace this data with actual service calls in later chapters and the rest of the application will be none the wiser.

Here we are creating an angelloModel service that exposes three methods to return statuses, types and stories.

Listing 2.20 STEP 19 app.js

```
myModule.factory('angelloModel', function() {
    var getStatuses = function() {
        var tempArray = [
            {name:'Back Log'},
            {name:'To Do'},
            {name:'In Progress'},
            {name:'Code Review'},
            {name:'QA Review'},
            {name:'Verified'},
            {name:'Done'}
        ];
        return tempArray;
    };

    var getTypes = function() {
        var tempArray = [
            {name:'Feature'},
            {name:'Enhancement'},
            {name:'Bug'},
            {name:'Spike'}
        ];
        return tempArray;
    };

    var getStories = function() {
        var tempArray = [
            {title:'Story 00',
             description:'Description pending.',
             criteria:'Criteria pending.',
             status:'To Do',
             type:'Feature',
             reporter:'Lukas Ruebbelke',
             assignee:'Brian Ford'},

            {title:'Story 01',
             description:'Description pending.',
             criteria:'Criteria pending.',
             status:'Back Log',
             type:'Feature',
             reporter:'Lukas Ruebbelke',
             assignee:'Brian Ford'},

            {title:'Story 02',
             description:'Description pending.',
             criteria:'Criteria pending.',
             status:'Code Review',
             type:'Enhancement',
             reporter:'Lukas Ruebbelke',
             assignee:'Brian Ford'},

            {title:'Story 03',
             description:'Description pending.',
```

```
            criteria:'Criteria pending.',
            status:'Done',
            type:'Enhancement',
            reporter:'Lukas Ruebbelke',
            assignee:'Brian Ford'},

           {title:'Story 04',
            description:'Description pending.',
            criteria:'Criteria pending.',
            status:'Verified',
            type:'Bug',
            reporter:'Lukas Ruebbelke',
            assignee:'Brian Ford'},

           {title:'Story 05',
            description:'Description pending.',
            criteria:'Criteria pending.',
            status:'To Do',
            type:'Spike',
            reporter:'Lukas Ruebbelke',
            assignee:'Brian Ford'}
        ];

        return tempArray;
    };

    return {
        getStatuses: getStatuses,
        getTypes: getTypes,
        getStories: getStories
    };
});
```

It is by now a running inside joke but the largest portion of code is by far the dummy data!

2.4.3 Consume the Services

Now that we have two new services we need to use them. We are going to use the dependency injection (DI) mechanism built into AngularJS to make angelloModel and angelloHelper available to the MainCtrl.

How do we do this? By adding them as parameters to the closure you created for the MainCtrl.

NOTE: We will do an anatomical deep dive into how DI works in AngularJS but for now it is a simple matter of adding whatever you want to expose as parameters.

From here we just initialize the properties on $scope by calling the appropriate service methods.

Here is the MainCtrl in its entirety:

Listing 2.21 STEP 20 app.js

```
myModule.controller('MainCtrl',
  function($scope, angelloModel, angelloHelper) {
    $scope.currentStory;

    $scope.types = angelloModel.getTypes();
    $scope.statuses = angelloModel.getStatuses();
    $scope.stories = angelloModel.getStories();
    $scope.typesIndex = angelloHelper
                          .buildIndex($scope.types, 'name');
    $scope.statusesIndex = angelloHelper
                          .buildIndex($scope.statuses, 'name');

    // CODE OMITTED
  });
```

2.5 Summary

And so as we bring this romantic evening to a close, I hope that you have seen how easy it is to get up and running with AngularJS. In just a short while, we have built an application that I would not be embarrassed to bring home to mother. It is something that you could actually use in the real world and we have managed to touch on some pretty important AngularJS concepts.

Let's review:

1. We covered the absolute minimum required code to get an AngularJS application off the ground.
2. We covered how to build controllers and services.
3. We covered how expose properties and methods to your view from your controller.
4. We covered how to exposed properties to your controller from your view via ng-model.
5. We covered how to call methods on the controller from the view using ng-click and ng-change.
6. We covered how to use ng-repeat and ng-options to let AngularJS do the templating for you.
7. And to top it all off, we covered how to use indexes to avoid looping and increase performance.

In the next chapter, in the next chapter we are going to discuss some of the high level architectural concerns behind building a large enterprise application that scales. We are also going to introduce the full version of Angello which is going to be quite impressive!

3

MVW The AngularJS Foundation

Now that we have taken AngularJS for a spin, it is time to get under the hood and see how the framework is put together. In this chapter we are going to cover the essential features that pretty much every AngularJS application has. There are going to be exceptions but as a matter of course if you are building a non-trivial application, you are going to need to use the pieces we are going to cover in this chapter.

"Where does my code go?" That is the inevitable question when approaching any new framework. In this chapter we are going to talk about the division of responsibility and how to organize your code in a way that makes AngularJS sense.

Chapter Objectives:

- Understand the main components that pretty much every AngularJS application has.
- Understand how to put these components together.
- Understand the AngularJS digest cycle so it is not so 'magical.'
- Understand how to set up and write a simple AngularJS unit test.

3.1 Introducing Angello

Allow us to introduce the application that we are going to use as the foundation for expounding and illustrating AngularJS concepts throughout the rest of this book. Drum roll please! Angello! One of the first really impressive web applications to surface on the Internet was Trello https://trello.com/ by Fog Creek Software http://www.fogcreek.com/. We wanted to pay homage to Trello by doing a variation of it in AngularJS called "Angello." See what we did there?

Angello is an agile project management application that allows users to create user stories, assign those stories to other users and track the status of the user story. You can see a picture of Angello in Figure 3.1 to get an idea of what we are going to be building. You can also download the entire project at http://angularjs-in-action.github.com/angello.

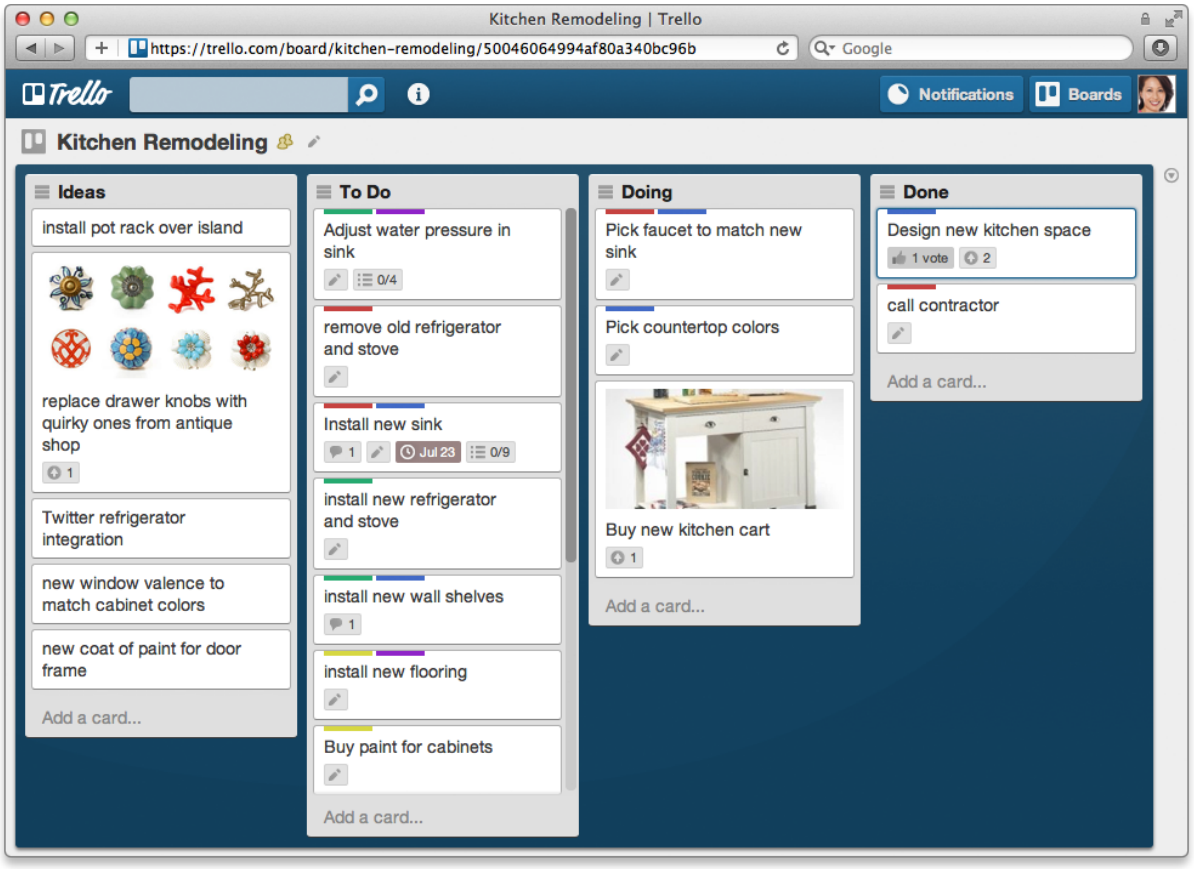

Figure 3.1 Angello in action

We will continue to add functionality to this Angello as we progress through AngularJS concepts so that by the end of the book we will have an application that multiple users can collaborate on user stories in real time and on a variety of devices.

In this chapter, we are going to lay a foundation by covering a lot of the same functionality we covered in the previous chapter but really elaborating on the underlying mechanisms that make these features possible. Specifically we are going to implement the CRUD (create, read, update, delete) operations of a user story including assigning the user story a status.

We are also going to cover some considerations and best practices along the way. To quickly recap

We should also consider adding the ability to create users and statuses. Although! We could work this into the next chapter.

3.2 ModelView or Whatever

If you recall the Figure 1-1 from Chapter 1 you will remember that AngularJS is an MVW framework. Because JavaScript is a functional, dynamic language some of the classical patterns start to break down. Instead of forcing AngularJS into an classical, imperative context it was decided to let it be whatever makes the most sense to the developer. Thus is it a Model View Whatever framework.

One possible interpretation that makes a lot of sense to developers is the MVVM or Model View ViewModel pattern. We are going to loosely frame the following sections around this interpretation. This is not something that we are dogmatic about and if it does not help then please do not get hung up on it. Focus on the individual parts and what they do.

It is easiest to think of the ViewModel as the $scope and controller working together as a single unit. The controller is responsible for setting up all the initial properties and methods for $scope and $scope is responsible for doing all the heavy lifting.

3.2.1 AUTHOR'S NOTE

I am a huge fan of the ViewModel pattern as I have seen how effective it is in the trenches of bad code. I was a Flex developer for many years and a lot of the same problems that AngularJS was designed to solve were the same problems that Flex developers were having years ago. One of the biggest pain points that I ran into doing Flex was how to keep logic out of the View and how to keep View concerns out of my Model. Eventually it was suggested that we implement a ViewModel pattern and it was like a knight in shining armor came riding up ready to save the day!

ViewModel allows you to separate your declarative markup and your imperative operations into their appropriate places. It makes testing the logic that

drives your view very easier which encourages writing more tests. It drastically reduces your code by having to write event handlers to try to make sense of what happened in your DOM which once again is mixing concerns.

3.2.2 Controllers

Every AngularJS has to have at least one controller and so this is a good place to start. Controllers are responsible for doing two things. First, they are responsible for setting up the initial state of the scope object and secondly they are responsible for adding behavior to that scope object. For instance, in our Angello project we need to have access to the stories that we have created. We accomplish this but using MainCtrl to create a property called stories on $scope. Having access to the stories illustrates the first responsibility of the controller but we also need to be able to create stories so that Angello is actually useful for managing projects. To do this we use MainCtrl again to create a method called createStory on $scope.

We are now going to show you the actual code to get this set up in Angello.

Step 1 Attach the Controller

You attach a controller to your application via the ng-controller directive and at the DOM element that you want the controller to own. Be judicious about where you attach your controller as it will define the portion of your page that the controller has domain over and no more. For instance, if you have a top level controller that you want to manage your entire page it would make sense to attach it to the body element of your page. On the other hand if you only want a controller to exercise control over a small portion of your page then only attach the controller to the outermost element of that portion.

```
<div id="masterWrapper" class="row" ng-controller="MainCtrl">
```

Step 2 Instantiate the Controller

You then instantiate the controller using the controller method on your AngularJS module. This keeps your controller off of the global scope object.

```
myModule.controller('MainCtrl',
    function($scope, angelloModel, angelloHelper) { });
```

Step 3 Build the Controller

You then can add properties and methods to the $scope object just like you would manipulate any JavaScript object.

```
myModule.controller('MainCtrl',
    function($scope, angelloModel, angelloHelper) {
        $scope.stories = angelloModel.getStories();

        $scope.createStory = function() {
            $scope.stories.push({title:'New Story',
                description:'Description pending.',
                criteria:'Criteria pending.',
                status:'Back Log',
                type:'Feature',
                reporter:'Pending',
                assignee:'Pending'});
        };
    });
```

Controller Best Practices

There are a few guiding principles around writing controllers but it can be summed up in a single sentence.

Controllers should be fine grained units of business logic that are easy to test.

Controllers should be fine grained. It is important to keep your controller small and focused specifically on the business logic of the view that it is responsible for. If you find that you are writing code that is of interest to more than one place in your application then consider it a good candidate to refactor to a service. There is a balance on how specific that you get with your controllers but always favor fine grained controllers over coarse grained controllers.

Controllers only contain business logic. It is important that controllers never manipulates the DOM as that dilutes your business logic by introducing declarative concerns making it harder to test. Having a proper grasp on the relationship that a View has with a ViewModel drastically reduces the need for any kind of DOM manipulation right out of the gates. The transition comes from seeing a View as representative and not reactive to the state of the application. In other words, that the View is only responsible for representing the state of the ViewModel and not reacting to the actions of the user and trying to figure out what to do next.

Controllers are easy to test. Controllers should be loosely coupled to the rest of the application and therefore they do not directly talk to other controllers. This makes it easy to write very small, specific tests around a single unit of responsibility.

It is also worth mentioning that controllers should not own your domain model. By "domain model" we are referring to code that actually manipulates state of your application as a whole. Stateful code belongs in services so that it is easy to share

across the application.

Consider talking about explicit and implicit creation of controllers

3.2.3 $scope

$scope is the lifeblood of any AngularJS application. It is responsible for storing application models and providing a context for the expressions that operate on those models. This is where we store important information that Angello needs like stories or define important functionality such as createStory.

$scope is basically a plain JavaScript object with some extra AngularJS functionality baked into it at compile time.

The three main characteristics of $scope: it has the ability to detect changes via $watch, the ability to explicitly propagate those changes via $apply provides a context in which to execute expressions on itself

We will get into how $digest, $apply and watch expression work in just a little bit but for now focus on the fact that when a property changes on your $scope object, it has the ability to inform your view of those changes.

To illustrate this, let us examine the code below.

```
myModule.controller('MainCtrl',
    function($scope, angelloModel, angelloHelper) {
        $scope.stories = angelloModel.getStories();

        // ...
    });
```

We are setting a property on $scope called stories. This property is now available to your view to bind to which we can do like this.

```
<article ng-repeat="story in stories>
    <h4>{{story.title}}</h4>
    <p>{{story.description}}</p>
</article>
```

When angelloModel.getStories() returns a value that updates $scope.stories, $scope will notify the ng-repeat directive that a change has occurred so that ng-repeat can update itself.

$scope also makes methods available for the view to call directly when an event has occurred. We have defined a createStory method on the $scope object and now it is available for use in the view.

```
$scope.createStory = function() {
    $scope.stories.push({ /* A NEW STORY OBJECT */ });
};
```

And now you can utilize that function like this.

```
<button ng-click="createStory()">New Story</button>
```

That is incredibly succinct and efficient! You are no longer having to wire up a bunch of event handlers to handle events on the DOM.

We have talked about controllers and scope and the important role they play but ultimately we need a way to visually represent what is going on in Angello. For instance, we need to be able to show the viewer what stories have been created and display some information about each story. This is where the View comes in!

3.3 View

The View is your HTML after it has been through the AngularJS compilation cycle. When an AngularJS application kicks off there is a two step process that happens to your HTML and JavaScript. The first step is the compile phase where AngularJS goes through your DOM and makes a note of all the directives by collecting all the link functions associated with each directive. The second step is the link phase where AngularJS injects the appropriate scope into each directive's link function setting up two-way data binding completing the link between scope and DOM.

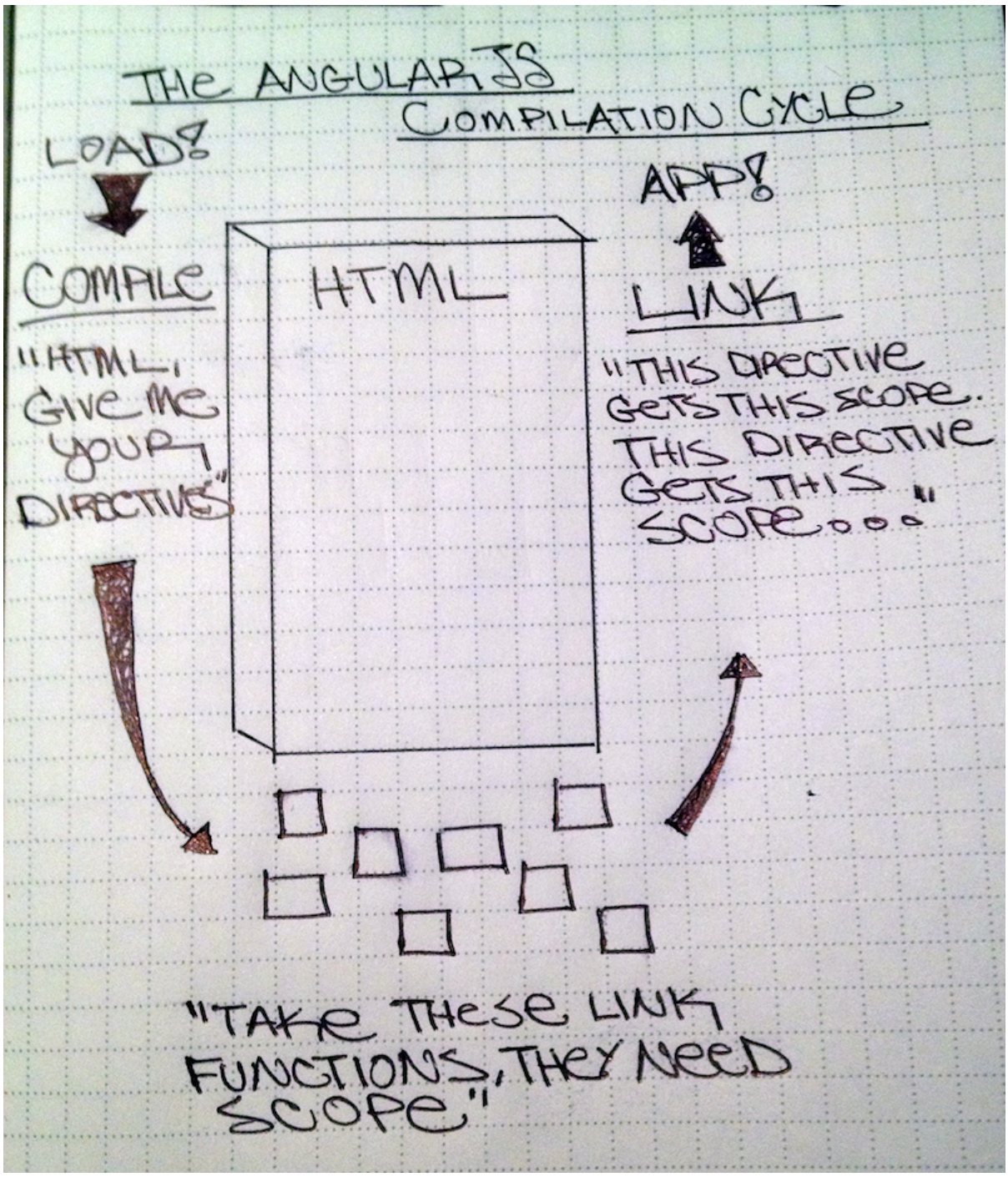

Figure 3.2 AngularJS Compilation

Once your HTML has been through the AngularJS compilation cycle... BAM! Superpowers! And that is your View.

3.3.1 Templates

In Angello, we want to be able display information without having to mark up the HTML by hand. This is a brave new world! The need for convenience becomes quite obvious when you need to display a large number of items that share the same visual characteristics. A list of user stories is a great example of this. Wouldn't it be nice if we could just define what we want one user story to look like and let AngularJS do the work of building out the rest of them? Well AngularJS is nice! And this is where templates swoop in to save the day!

Templates in AngularJS is just HTML. HTML is an incredibly fixed and limited technology and was never designed with the modern web in mind. Its entire genesis comes from the desire to approximate print design in the browser.

The good thing about this is that HTML does declarative markup really well. The bad thing about this is that it ONLY does declarative markup really well.

The positive side of this characteristic is that HTML is very good at declarative markup. Some JavaScript frameworks try to overcome this limitation by obscuring JavaScript away entirely. They do this by either pulling into an imperative context where all your layout is done in JavaScript or force you to do your layout in a completely different language and then compile it into HTML.

AngularJS takes another approach to this problem by simply embracing HTML and extending it instead of marginalizing what it does well. Another important benefit to this approach is that it is very easy to use AngularJS templates in an existing workflow and with established designers and developers. The learning curve is practically zero.

Let us give you an example. Our designer could deliver the markup for the form to create a user story without any concern whatsoever about we are going to do to it once we get our AngularJS claws into it.

Behold the HTML!

```
<div id="cardDetails" class="small-12 large-4 columns">
    <h2>Card Details</h2>
    <form class="form-horizontal">
        <div class="control-group">
            <label class="control-label" for="inputTitle">Title</label>
            <div class="controls">
                <input type="text" id="inputTitle" placeholder="Title">
            </div>
        </div>
        <!-- OMITTED -->
    </form>
    <br>
```

```
    <button>New Story</button>
</div>
```

And now all we have to do is go through and wire it up.

```
<div id="cardDetails" class="small-12 large-4 columns">
    <h2>Card Details</h2>
    <form class="form-horizontal">
        <div class="control-group">
            <label class="control-label" for="inputTitle">Title</label>
            <div class="controls">
                <input type="text"
                    id="inputTitle"
                    placeholder="Title"
                    ng-model="currentStory.title">
            </div>
        </div>
        <!-- OMMITTED -->
    </form>
    <br>
    <button ng-click="createStory()">New Story</button>
</div>
```

Seriously. That's it. Don't be jealous!

3.3.2 Two-Way Data Binding

When we select a user story in Angello and the information populates the form in the right column without a page refresh that is two-way data binding doing its thing. When you change a property in the form and the user story automatically updates that is more data binding!

Two-way data binding is the supermodel of the framework. It gets all the attention and rightly so. It is does something stunning to your application. Traditionally the majority of the code in most web applications was concerned with DOM manipulation and event handling.

If you've used another framework, you've probably made calls to some 'render' or 'update' function in your controller. You don't need to do this in Angular. Because the views are data-bound, Angular handles figuring out when the UI needs to change.

Two-way data binding can happen explicitly and implicitly. Explicitly is via string interpolation using the curly braces or implicitly via a directive.

Let us examine some code and point out both kinds of bindings.

```
<div
```

```
    id="column01"
    class="small-12
    large-2 columns"
    ng-repeat="status in statuses">

    <h3>{{status.name}}</h3>
    <article class="large-12 columns cardWrapper"
        ng-repeat="story in stories | filter:status.name"
        ng-click="setCurrentStory(story)">
        <h4>{{story.title}}</h4>
        <p>{{story.description}}</p>
    </article>
</div>
```

The explicit binding just pop out because they are surrounded with curly braces as in the case of `{{status.name}}` or `{{story.title}}`. When those properties on that object changes then the appropriate elements will update.

The implicit bindings come into play around the directives such as `ng-repeat="status in statuses"` where the binding is to statuses. When statuses change then the `ng-repeat` directive will pick up that change and update the DOM accordingly.

There is one other way to set up two-way data binding that occasionally comes into play if you are integrating an AngularJS application into a larger application that has a server side language. For instance, we have seen a Django application try to interpret AngularJS bindings as Django code and blow up.

The way around that is to use the ng-bind directive. So in this case, `<p>{{story.description}}</p>` becomes `<p ng-bind="story.description"></p>`.

In Angello, we need to separate our stories by their current status. So how do we tell AngularJS to give us all the stories that have a current status of "In Progress" so that we can put them in the "In Progress" column? AngularJS comes with a really great tool for this called filters to accomplish this.

3.3.3 Filters

Filters in AngularJS are super handy because they allow you to modify your data before it is rendered in the view. Another way to say this is that filters allow you to refine the stories collection to stories with a specific status before it is displayed to the user.

Let us jump into some code to drive the point home and then elaborate on it afterwards.

```
<div
    class="small-12 large-2 columns"
    ng-repeat="status in statuses">

    <h3>{{status.name}}</h3>
    <article class="large-12 columns cardWrapper"
        ng-repeat="story in stories | filter:{status:status.name}"
        ng-click="setCurrentStory(story)">
        <h4>{{story.title}}</h4>
        <p>{{story.description}}</p>
    </article>
</div>
```

This is a ridiculously succinct block of markup code that is responsible for rendering all the user stories sorted based on their statuses.

```
<div class="small-12
    large-2 columns"
    ng-repeat="status in statuses"></div>
```

The outer block of code is simply iterating over the the statuses collection and creating a column for each status.

```
<article class="large-12 columns cardWrapper"
    ng-repeat="story in stories | filter:{status:status.name}"
    ng-click="setCurrentStory(story)">
    <h4>{{story.title}}</h4>
    <p>{{story.description}}</p>
</article>
```

We want to show only the user stories that have the same status as the column they are in. This is done by adding a filter modifier to the end of the ng-repeat directive. The filter return returns a new array based on the expression that you pass into it. In this case, we are telling AngularJS to return all of the story items that have a status that matches the name of the status column it is in.

You can also pass in a string or a function into the filter modifier. Passing in a string works exactly the same as passing in an object but it will return any object that has a property that matches the filter. On complex objects that could have the same value for multiple properties this could yield unintended results. I did not realize this was going to be a problem until I called up my friend Backlog to test the app and it blew up! The lesson here is to use a string when you are dealing with simple arrays and objects.

3.3.4 Digest Cycle

So let us take a moment and answer the burning question of "How does AngularJS know when something in Angello has changed and it is time to update the bindings man!?" AngularJS works on a concept of dirty checking and it is one of the core tenants of how the entire framework operates.

We would like to put a disclaimer here that the next couple paragraphs are going to slant a bit towards the academic side. AngularJS works just fine without a deep understanding of what is going on at a molecular level. Feel free to skip this section but for the curious, read on!

Dirty checking is the simple process of comparing a value with its previous value and if it has changed then a change event is fired.

AngularJS performs dirty checking via a digest cycle that is controlled by $digest. $digest happens implicitly and you never have to call this directly. If you need to initiate a digest cycle, then use $apply as it calls $digest but has error handling mechanisms built around it.

During the compilation phase, $scope evaluates all of its properties and creates a watchExpression for each one. You can manually create watchExpressions but implicitly created watchExpressions are simple functions that compare the value of the property with the previous value using `angular.equals`.

It is during the $digest cycle that all watchExpressions for a scope object is evaluated. When a watchExpression detects that a $scope property has changed then a listener function is fired.

Occasionally a property is changed without AngularJS knowing about it and it is at that time you can manually kick off a digest cycle via `$apply`. The most common reason for this is that you have made an API call or a 3rd party library has done something that AngularJS needs to know about.

3.4 Model

In Angello, there are application models that are concerned with a very specific portion of the application such as what is the title of a specific story. In the AngularJS world, this is any property that is exposed via $scope and reachable by the rest of the application. This is not the Model with a capital M in Model View Whatever.

The Model in an AngularJS application is your domain model that holds the data that generally concerns your entire application. Your domain model will often communicate its state with a remote server so that it can be used elsewhere or at a

later time. With that said, the Model in an AngularJS application is nothing more than native JavaScript data structures such as arrays and objects that you expose via Services and dependency injection.

3.4.1 Services

A good example of a block of Angello code that belongs in a service is the general purpose helper functions. It is possible that we would want to construct an index of objects to make object lookup more efficient and we would not want to duplicate the code that generates that index over and over in every controller that needs to build one. The proper approach is to pull that function out into a service and make it available for anything in the app that needs it.

Services are created by registering them with the module they are going to operate in. The easiest way to create and register a service is to use the factory shortcut method on Module. This is a convenience method that actually calls the factory method on the $provider service. We will get to this in the next section.

Let us deconstruct the angelloHelper service and how it is put together.

```
myModule.factory('angelloHelper', function() {
    var buildIndex = function(source, property) {
        var tempArray = [];

        for(var i = 0, len = source.length; i < len; ++i) {
            tempArray[source[i][property]] = source[i];
        }

        return tempArray;
    };

    return {
        buildIndex: buildIndex
    };
});
```

We are registering the angelloHelper service with myModule via the factory function and providing it with a function that will executed when the service is requested for the first time. We are defining a method called buildIndex and returning it when the function is executed so that it is available for use.

To use the angelloHelper service we simply inject it into the controller that needs it. Examine the code below.

```
myModule.controller('MainCtrl',
    function($scope, angelloModel, angelloHelper) {
```

```
        // ...
    }
```

AngularJS is smart enough to know that the MainCtrl needs angelloHelper and provides it via dependency injection. From inside the controller we can call the buildIndex method on angelloHelper at will.

```
myModule.controller('MainCtrl',
    function($scope, angelloModel, angelloHelper) {
        //...

        $scope.typesIndex = angelloHelper.
            buildIndex($scope.types, 'name');

        $scope.statusesIndex = angelloHelper.
            buildIndex($scope.statuses, 'name');

        //...
    });
```

AUTHOR'S NOTE

Services are not particular magical in terms of what is happening under the hood. The most common implementation of a service is generally a module pattern or revealing module pattern. If you are interested in learning more about these pattern, we recommend starting with JavaScript The Good Parts by Douglas Crockford or JavaScript Patterns by Addy Osmani.

So now that we have establish how to create a service in Angello, how does the rest of the application use it? This is accomplished via dependency injection

3.4.2 Dependency Injection

We hinted at dependency injection up until this point and now it is time to dig into how it works within AngularJS. Dependency injection lives up to its name by doing exactly what its name implies. It injects the object that your code has a dependency on.

Traditionally this has been done in other languages by annotating the code to define dependencies. JavaScript as a language does not have an annotation mechanism built into it so we have to accomplish this either implicitly or explicitly in AngularJS. To do this implicitly, we simply evaluate function signatures using the toString method and then see if there is a matching service available. Explicitly, we use the $inject annotation to define what is going to be injected.

One of the most important features of dependency injection is that it makes

testing your code significantly easier. The reason being is that you can inject whatever you want in place of a fully functioning service. Do you have a service that calls a remote server? Create a fake or mock service to simulate that behavior so you can remove actually calling the server out of the testing equation.

We've gotten the app to a pretty good point. We can manually confirm everything is working right now in a few seconds by manually interacting with the app on our own. This is fine for small apps, but as our app grows and gains new features, it's going to be difficult and time consuming to verify that everything works. Furthermore, changes in web browsers, APIs, or other libraries may break your app. For these reasons, we think it's important to automatically test your app.

AUTHOR'S NOTE

Implicit dependency injection works fine until you minimize your code and then it falls apart. For production use, we recommend that you explicitly annotate your injections with $inject.

So how does AngularJS know what to actually provide to your object when it needs something? The $provider service of course! Services are registered with the $provider service when an AngularJS application is starting up. When a service is requested $provider returns the result of the factory function provided when the service was registered. It is important to note that services are effectively singletons and so the factory function is only called once and stored with the `$provider`.

3.5 Testing

We'll be using the Karma Test Runner to automatically test our app. We'll start by showing how to setup the Karma test runner to run unit tests in this app, then go through testing each piece.

3.5.1 Setting up Karma

To install Karma, you'll need Node.js, which you can get from the project site at http://nodejs.org/. Once you have Node.js installed, installing the Karma command line tool is simple. Just run `npm install -g karma` (note that you might need to run this command with `sudo` if you're using a Unix-based system) in your terminal.

To verify that Karma is now installed, run `karma` in your terminal. If all is well, you should be greeted with the following:

```
$ karma
```

```
Command not specified.
Karma - Spectacular Test Runner for JavaScript.

Usage:
  karma <command>

Commands:
  start [<configFile>] [<options>] Start the server / do single run.
  init [<configFile>] Initialize a config file.
  run [<options>] Trigger a test run.

Run --help with particular command to see its description and
available options.

Options:
  --help     Print usage and options.
  --version  Print current version.
```

Great! Now we need a config file. The `karma` command line tool can help us out here too. Let's run `karma init`, which will present us with a set of prompts to get us going. For the most part, we're going to use the default options in the prompts, but this is a good opportunity to look at how Karma works.

```
$ karma init

Which testing framework do you want to use ?
Press tab to list possible options. Enter to move to the next question
> jasmine
```

Karma supports multiple different testing libraries and assertion frameworks. These libraries basically define the way you write tests. "Jasmine" is one popular choice, and not coincidentally it's the default option for Karma.

```
Do you want to use Require.js ?
This will add Require.js adapter into files.
Press tab to list possible options. Enter to move to the next question.
> no
```

We're not going to use Require.js, so answer "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
>
```

Karma uses real browsers to run your tests. This is great, because it ensures that your app works even in the face of browser inconsistencies. For this prompt, choose whatever browser or browsers you want to develop with. We're going to use just Chrome for now.

```
Which files do you want to test ?
You can use glob patterns, eg. "js/*.js" or "test/**/*Spec.js".
Enter empty string to move to the next question.
> javascripts/**.js
> tests/**.js
WARN [init]: There is no file matching this pattern.
>
```

This option is for configuring which files Karma loads to run tests. We want to add two lines: `javascripts/**.js` and `tests/**.js`. The first will load up our application code, and the second is for out tests, which we have not written yet.

```
Any files you want to exclude ?
You can use glob patterns, eg. "**/*.swp".
Enter empty string to move to the next question.
>
```

We don't have to worry about excluding files, so hit enter and move along.

```
Do you want Karma to watch all the files and
run the tests on change ?

Press tab to list possible options.
> yes
```

One cool feature of Karma is that it can automatically re-run our tests whenever your applications' files change. We're a huge fan of this workflow, so let's answer "yes".

```
Config file generated at "some/dir/karma.conf.js".
```

At the end of these prompts, Karma will write a config file. The file is pretty well documented in just the comments, but you can check out the documentation

on http://karma-runner.github.io/ for more information. If you ever need to change the options aside from what we configured when we followed the prompts, just edit this file. Note that this is just a JavaScript file itself, so you can use the familiar JavaScript syntax. So for instance, if we wanted to also test in Firefox, we'd just add "Firefox" to the `browsers` array in this file.

With Karma now configured, let's try to run our tests. Run `karma start karma.conf.js`, and you should see the following:

```
$ karma start karma.conf.js
INFO [karma]: Karma server started at http://localhost:9876/
INFO [launcher]: Starting browser Chrome
WARN [watcher]: Pattern "some/dir/tests/**.js" does not match any file.
INFO [Chrome 27.0 (Mac)]: Connected on socket id M7rTQCAByo7HKHCAGwaj
Chrome 27.0 (Mac): Executed 0 of 0 SUCCESS (0.027 secs / 0 secs)
```

Also, you should notice that Chrome has started (if it was installed), and it looks something like this:

Figure 3.3 Karma running a Chrome browser for tests

What's going on here, exactly? As mentioned, the Karma test runner launches a browser that it uses to run the test. This browser will stay open as long as Karma is running. Feel free to minimize this window and forget about it, since the important

details about the test are reported via the command line interface.

With that done, it's time to start writing tests.

3.5.2 Testing Services

Let's start by writing a test for the angelloModel service. Make a directory, `tests`, and inside of it, a file called `angelloModelSpec.js`. We'll be writing our tests in this file.

We want to be able to organize our tests so we know where to look when a test fails. Jasmine (the testing framework we're going to be using) lets you do this with `describe`.

```
describe('Service: angelloModel', function () {
    // write tests for this service here
});
```

Better yet, you can nest `describe` blocks. We find that it's best to have one describe block per method on a service, with many tests for each method. So let's start with the getStatuses method:

```
describe('Service: angelloModel', function () {
    describe('#getStatuses', function () {
        // write tests for angelloModel.getStatuses here
    });
});
```

We're almost ready to start testing. The only thing left to do is load the relevant module and instantiate our service

```
describe('Service: angelloModel', function () {

    // load the service's module
    beforeEach(module('Angello'));

    var modelService;

    // Initialize the service
    beforeEach(inject(function (angelloModel) {
        modelService = angelloModel;
    }));

    describe('#getStatuses', function () {
        // write tests for angelloModel.getStatuses here
    });
});
```

Notice the beforeEach function being invoked here. This means we'll get a fresh instance of the module and the service before each of our tests. The beforeEach function is scoped to the `describe` block that it appears in, so the service will be reinstantiated before each test in not only the getStatus describe block, but also the other methods as we add them. We'll see how to use beforeEach and it's counterpart afterEach more soon to setup and teardown tests.

Now we're ready for an actual test. Much like we have `describe` blocks for grouping tests, we have `it` blocks for the test themselves. `it` is a function that takes a string describing the test as the first argument, and a function in which the test runs for the second argument. For our first test, let's just check that there are seven statuses returned by `angelloModel.getStatuses()`. Let's see what it looks like:

```
it('should return seven different statuses', function () {
    expect(modelService.getStatuses().length).toBe(7);
});
```

With our first test added, let's go back to the console and see how it went.

```
Chrome 27.0 (Mac): Executed 1 of 1 SUCCESS (0.129 secs / 0.013 secs)
```

Perfect! Let's break the code now for just a second. Comment out one of the statuses in our model service.

```
Chrome 27.0 (Mac) Service: angelloModel #getStatuses
should return seven different statuses FAILED
Expected 6 to be 7.
Error: Expected 6 to be 7.
  at null.<anonymous> (some/dir/tests/modelSpec.js:18:49)
Chrome 27.0 (Mac): Executed 1 of 1 (1 FAILED) (0.203 secs / 0.023 secs)
```

As you can see, this caused the test to fail, but it also let us know that there were only six models when seven were expected. This should make it easy to narrow down the source of a problem. We'll give some less contrived examples of how to use this shortly, but for now, uncomment the commented status so that the tests pass again.

Jasmine provides a lot of great ways to make our tests more readable.In

particular, this test shows how to check expected versus actual values with the `expect` API. `expect` returns an object with many different helper methods for checking the expected value. Here we used `toBe` to check equality, but there's also `toBeGreaterThan`, `toBeDefined`, `toBeFalsey` and many others.

Let's write another test to make sure that one of the statuses has the name "To Do." We'll have to pluck the "name" of each status out of the array of status objects, and then check that one is equal to "To Do." Here's how you would write this succinctly:

```
it('should have a status named "To Do"', function () {
    expect(modelService.
        getStatuses().
        map(function (status) { // get just the name of each status
            return status.name;
        }).
        toContain('To Do');
});
```

Save the file, and you should see the tests automatically run again:

```
Chrome 27.0 (Mac): Executed 2 of 2 SUCCESS (0.279 secs / 0.013 secs)
```

Perfect. Typically, we like to write tests before the implementation. The reason for this is that you can never be sure a test works until you see it first fail, then succeed after you add the feature or fix that the test is testing for. If you write tests after the implementation, it's good to test them out to make sure that they fail when their assertions are not met. Try commenting out the "To Do" status in our model, and both tests should fail. Then try uncommenting it and see both succeed.

Continuing in the same fashion, we can add some tests for the other methods of this service. Here's what the final tests for angelloModel looks like:

```
describe('Service: angelloModel', function () {

  // load the service's module
  beforeEach(module('Angello'));

  var modelService;

  // Initialize the service
  beforeEach(inject(function (angelloModel) {
    modelService = angelloModel;
  }));
```

```
  describe('#getStatuses', function () {

    it('should return seven different statuses', function () {
      expect(modelService.getStatuses().length).toBe(7);
    });


    it('should have a status named "To Do"', function () {
      expect(modelService.
        getStatuses().
        map(function (status) { // get just the name of each status
          return status.name;
        }).
        toContain('To Do');
    });

  });

  describe('#getTypes', function () {

    it('should return four different types', function () {
      expect(modelService.getTypes().length).toBe(4);
    });

    it('should have a type named "Bug"', function () {
      expect(modelService.
        getTypes().
        map(function (status) { // get just the name of each status
          return status.name;
        })).
        toContain('Bug');
    });

  });

  describe('#getStories', function () {

    it('should return six different stories', function () {
      expect(modelService.getStories().length).toBe(6);
    });

    it('should return stories that have a description property',
      function () {
        modelService.getStories().forEach(function (story) {
          expect(story.description).toBeDefined();
        });
      });

  });

});
```

We now have some rough guarantees about the angelloModel service. As we

develop our app and change the features of this service, we'll return back to these test to update or add to them. Although we've made great progress, there's still more to be done. Next we'll test the behavior of our controller.

3.5.3 Testing Controllers

Next, let's test our controller. In the same way we instantiated our service before each test, we need to instantiate our controller. Let's make a file in our `tests` directory called `MainCtrlSpec.js` and start it with the following:

```
describe('Controller: MainCtrl', function () {

  // load the controller's module
  beforeEach(module('Angello'));

  var MainCtrl,
    scope;

  // Initialize the controller and a mock scope
  beforeEach(inject(function ($controller) {
    scope = {};
    MainCtrl = $controller('MainCtrl', {
      $scope: scope
    });
  }));

});
```

Note that we are manually passing arguments into `MainCtrl` within the `beforeEach` block. By doing this, we can "mock out" different services, passing in a different implementation used just for tests. We'll discuss this more in a bit. For now, let's just test that the controller attaches models to the scope. Add these tests to the body of the `describe` after the `beforeEach` block:

```
it('should attach a list of story types to the scope', function () {
    expect(scope.types).toBeDefined();
});

it('should attach a list of story statuses to the scope', function () {
    expect(scope.statuses).toBeDefined();
});

it('should attach a list of stories to the scope', function () {
    expect(scope.stories).toBeDefined();
});
```

The tests run, and they should pass.

```
Chrome 27.0 (Mac): Executed 9 of 9 SUCCESS (0.293 secs / 0.024 secs)
```

If you recall, our controller depends on a few services. Because we are writing unit tests, we want to isolate the behavior of the controller from the behavior of the services that it depends on. To do this, we'll need to create mock services. We can leave `angelloModel` as is for now, since it simply returns hard-coded data. We could write a mock for `angelloHelper`, but the only method of this service, `buildIndex`, is relatively simple. As we add to these services, we'll need to think about mocking them out, but we can move on for now.

3.6 Summary

This chapter has laid a strong foundation for what goes into any AngularJS application. We covered the Model View Whatever in depth and highlighted the parts within each of those sections and how they are at play within Angello.

To summarize:

$scope and controller can be thought of as a ViewModel.

View is just HTML with AngularJS compiled into it.

Model is comprised of services.

Controllers are small, highly focused pieces of business logic that are generally concerned with a single view.

Controllers should not own the domain model. If something is needed by two controllers then make it available via a service.

Two-way databinding eliminates an incredible amount of code making it much easier to write and maintain your application.

4 Integrating with a Server

Chapter Objectives:

- Understand how to make asynchronous HTTP requests in AngularJS with $http
- Understand how to use RESTful web services with AngularJS through $resource.
- Understand promises in these APIs.
- Learn how to integrate WebSockets with Angular to make the app update in real-time.
- Understand how to test all of this.

At this point, Angello is looking pretty good from a UI standpoint. Users can create and organize tasks and comment on them. However, if a user refreshes the page, their data is gone! This app is still missing persistence. This lack of persistence also means that users cannot share tasks, comments, or statuses with each other. In order to persist and share data between users of our app, we need a central place to store this data. For this, we'll need to use a web server. For the first part of the chapter, we're going to concentrate on how we can use AngularJS to communicate with a server that stores Angello's data.

Maybe you've written your own web server with Django, Node.js, PHP, or one of the other numerous options. Even if you're not writing your own server, you may use AngularJS with some 3rd party API, for instance if you want users to be able to login to your app with their Twitter or Google accounts. In 4.1 we'll look at communicating with a server using AngularJS's service, `$http`

including login, retrieving user data, and finally, saving changes. In 4.1.1, we'll see how to use AngularJS with OAuth 2.0 to let users sign in with their Google account. Still, communicating with the server can be quite complicated. When a user logs in, they might get data for one task from the server, then get comment information, and finally update a timestamp representing the last time they logged in. Managing these different asynchronous actions can get quite complicated, especially when you consider that any or all of these might fail or throw errors along the way. In 4.4, we'll talk about using "promises" to organize all of these requests to the server so your code stays clean as you add features. Traditionally, web browsers communicate with web servers by sending a request for information and getting back a response containing the information. Because of this, in most web apps when one user does something, another user needs to refresh the page in order to see it. For Angello, we want to make sure users don't miss important comments. Furthermore, because the app is collaborative, we don't want one user to update a task, then have another user's update overwrite those changes because they didn't know someone else was also modifying the same information. In order to solve this problem, we want this part of the app to update in real-time. In 4.5, we'll go through how to do this using AngularJS with WebSockets.

At this point, Angello is looking pretty good from a UI standpoint. Users can create and organize tasks and comment on them. However, if a user refreshes the page, their data is gone! This app is still missing persistence. This lack of persistence also means that users cannot share tasks, comments, or statuses with each other. In order to persist and share data between users of our app, we need a central place to store this data. For this, we'll need to use a web server. For the first part of the chapter, we're going to concentrate on how we can use AngularJS to communicate with a server that stores Angello's data.

Maybe you've written your own web server with Django, Node.js, PHP, or one of the other numerous options. Even if you're not writing your own server, you may use AngularJS with some 3rd party API, for instance if you want users to be able to login to your app with their Twitter or Google accounts. In 4.1 we'll look a communicating with a server using AngularJS's `$http` service, including login, retrieving user data, and finally, saving changes. In 4.1.1, we'll see how to use

AngularJS with OAuth 2.0 to let users sign in with their Google account.

Still, communicating with the server can be quite complicated. When a user logs in, they might get data for one task from the server, then get comment

information, and finally update a timestamp representing the last time they logged in. Managing these different asynchronous actions can get quite complicated, especially when you consider that any or all of these might fail or throw errors along the way. In 4.4, we'll talk about using "promises" to organize all of these requests to the server so your code stays clean as you add features.

Traditionally, web browsers communicate with web servers by sending a request for information and getting back a response containing the information. Because of this, in most web apps when one user does something, another user needs to refresh the page in order to see it. For Angello, we want to make sure users don't miss important comments. Furthermore, because the app is collaborative, we don't want one user to update a task, then have another user's update overwrite those changes because they didn't know someone else was also modifying the same information. In order to solve this problem, we want this part of the app to update in real-time. In 4.5, we'll go through how to do this using AngularJS with WebSockets.

SIDEBAR HISTORY OF WEB DEV

Originally, asynchronous communication with a web server was called "AJAX," short for "Asynchronous JavaScript and XML." This was because XML was frequently used as the transport between web browsers and web servers. As web apps evolved, JavaScript Object Notation (or JSON) became the favored format (replacing XML), and is now the Lingua Franca of the web. Almost all web platforms (Node.js, Ruby on Rails, PHP, .NET) support parsing and encoding JSON, and AngularJS is no exception.

The technique of asynchronously communicating with JSON-encoded data is basically the same as "AJAX." There's no fancy acronym associated with this technique, and because it's become so ubiquitous, when you see "asynchronous HTTP request," you can safely assume that JSON is the implied format. Another now outdated technique, called "Comet" (a play on words with "AJAX") was often used to give servers push-like communication by holding onto the response for an HTTP request until something happened on the server. A client would then "poll" the server by regularly sending it requests, and the server would either let the request timeout or return something. This implementation of Comet is called "long polling." In 2011, WebSockets were introduced, which

made it easier for web servers to send push notifications. Using WebSockets is generally much simpler than Comet, both conceptually and in implementation. You'll see what this looks like in section 4.5.

We wanted to touch on these terms (AJAX, Comet, and long polling) in case you encountered them in other literature. While there might still be some merit in them, we want to use and showcase modern techniques used to accomplish the same thing.

By the end of the chapter, you'll be comfortable using AngularJS to make async calls to the server, using "promises" to simplify async code, and even integrate websockets into an app to allow it to update in real time.

4.1 Logging in with $http

Let's start with logging in, since that's the first thing your user will need to do. The basic idea behind logging in will be to issue an HTTP request to the server to check if the current user is logged in. If they are logged in, the user continues to the app. If not, we'll direct the user to a "login" route with a "username" and "password" form. A user then completes the form, and clicks submit. If the login is successful, the server returns a session token, and the app continues on. Otherwise, the user is prompted to try again.

AngularJS exposes a few different APIs to you to use to make asynchronous HTTP requests, but we're going to start with the simplest, called `$http`. `$http` is a service, so you can inject it into a controller just like you did earlier with `$scope`.

NOTE: By convention, services provided by AngularJS are prefixed with a dollar sign. These APIs are considered public and stable. Occasionally, you'll come across a method or property prefixed with two dollar signs, like `$scope.$$phase`. These "double dollar sign" methods are considered private, and might have breaking changes across different versions of AngularJS.

If we `GET /login`, let's assume the server will return true if logged in, and false otherwise.

```
// in controller
$http.get('/login').success(function (status) {
    $scope.status = 'You are ' +
    (res.loggedIn?'':'not ') +
    'logged in';
});
```

To display the result, let's add this to our view:

```
<p>{{status}}</p>
```

For this chapter, we've written a very simple HTTP server to get you going. It's available at `ch04/simple-server`. You run it using Node.js. The command from the `simple-server` directory is `node app.js`. We'll describe the API for the server as we use it.

Run the server, and open up the application in your browser. You shoul predictably see `you are not logged in` on the page. So far so good! Next let's implement logging in for the user.

According to the server's spec, if we POST the following JSON, we'll be logged in:

```
{
   "name": "brian",
   "password": "password"
}
```

Let's add a function that will make this request to the server, and store a session token if the login is successful. Back in the controller:

```
$scope.user = {};
$scope.login = function (user) {
      $http.post('login', $scope.user).success(function (res) {
            $scope.status = 'You are ' +
             (res.loggedIn?'':'not ') +
            'logged in';
      if (res.loggedIn) {
            $cookieStore.put('sessionToken', res.token);
      }
   });
};
```

And finally in the view:

```
<form ng-submit="login(user)">
   <p>User: <input ng-model="user.name"></p>
   <p>Password: <input type="password" ng model="user.password"></p>
   <input type="submit" value="Login">
</form>
```

Refresh the page to try the app again. Type "brian" for the user, and "password" for the password. Submit, and you should see the status change from "not logged" in to "logged in."

You've successfully implemented a login system, but now you want to do something with this information. You want users who are logged in to see the app, and users who have not logged in to be presented with a login dialog. To do this, you'll need to make use of AngularJS's route system. We will cover routes more thoughoutlly in Chapter 6, but the concept is faily easy. To do this, you'll use to redirect the user based `$location` on whether or not they are logged in.

Let's change this code in the controller accordingly:

```
$http.get('/login').success(function (res.loggedIn) {
      if (res.loggedIn) {
            $location.path('/login');
      }
});
```

But we would need to duplicate this behavior in each controller. This is problematic, because you don't want to repeat yourself. Additionally, the user's login status will need to be shared throughout the app. For instance, we'll need to see the user login state between different routes, and some services might not be available. In order to consolidate and share this information, let's make a service called `loginService`:

```
angular.module('Angello.login', []).
service('loginService', function ($http, $location) {
     $scope.login = function (user) {
          $http.post('login', $scope.user).success(function (res) {
               $scope.status = 'You are ' +
                     (res.loggedIn?'':'not ') +
                    'logged in';
               if (res.loggedIn) {
                    $cookieStore.put('sessionToken', res.token);
               }
          });
     };
var isLoggedIn = function () {
    $http.get('/login/status').success(function (status) {
        if (!status) {
            $location.path('/login');
        }
    });
};
// API
return {
    login: login,
    isLoggedIn: isLoggedIn
  };
});
```

Then, we can remove the code from the controller, instead just injecting loginService and exposing its methods to the scope:

```
$scope.login = loginService.login;
```

With a service dedicated to managing login, we can change the app's route based on whether or not they are logged in. We listen for the `$routeChangeStart` event, stop the route change if the user is not logged in, and redirect them to the login view. Let's add this to the existing code:

```
service('loginService', function ($http, $location) {
    $scope.login = function (user) {
        $http.post('login', $scope.user).success(function (res) {
            $scope.status = 'You are ' +
                 (res.loggedIn?'':'not ') +
                'logged in';
            if (res.loggedIn) {
                $cookieStore.put('sessionToken', res.token);
             }
        });
    };
var isLoggedIn = function () {
   $http.get('/login/status').success(function (status) {
      if (!status) {
         $location.path('/login');
      }
   });
};
// the user is stuck at '/login' until they are logged in
$rootScope.$on('$routeChangeStart', function (current, next) {
   if (!loginState && next !== '/login') {
      $location('/login');
   }
});
// API
return {
   login: login,
   isLoggedIn: isLoggedIn
};
```

Note that this redirecting merely prevents the user from seeing something before they are logged in, regardless of what route they come from. You still need to implement proper server-side security.

With this simple addition to our routing in place, lets try logging in again. Once logged in, you should be able to visit other routes:

TODO: screenshot

Great! This is a great way to use your own login system, but let's take a look at another way to accomplish the same thing: allowing users to login with their Google account using OAuth.

4.1.1 Logging in with OAuth

Login systems are great, but implementing them is kind of a pain. In order to prevent spammers from creating tons of accounts, you might require new users to register their email address. You want to prevent a malicious user from writing a script to create many new accounts, so you implement a CAPCHA system where a user types in some obfuscated string of characters. And even then, you still have to closely moderate the accounts for malicious behavior. That's a lot of work!

Individual logins are also a pain for users, since it gives them yet another account name and password to remember. For this reason, some developers choose to use the login APIs of Google, Facebook, or Twitter. These APIs are typically implemented with either OAuth or OpenID standards.

Let's say we're using "Sign in with Google." The basic idea is that your app redirects the user to the auth provider's website (in this case, google.com), they login, and the auth provider's site redirects back to yours with a login token that your app can use to communicate with the auth provider on behalf of the user. This might sound a bit convoluted, but it ensures that all parties are as secure as possible. Below is a quick illustration of OAuth when used with Google.

Figure x.y: Illustration of OAuth.

How do you implement this? There's a little bit of configuration we need to do before we get started. For security reasons, Google requires a "client side secret" and an "app id" when you make the first request. To set these up, you'll need to register your app with their API system. Visit https://developers.google.com/+/web/signin/ for more information.

In our LoginCtrl, write the following code, appropriately replacing "clientId" with the clientId you receive from Google when registering your application:

```
var clientId = 'your id here';
var authScope = 'https://www.googleapis.com/auth/userinfo.profile';
if ($routeParams.key) {
    // …
}
$scope.googleAuth = function () {
    $window.location =
```

```
            'https://accounts.google.com/o/oauth2/'
                'auth?response_type=token&client_id=' +
                clientId +
                '&redirect_uri=' + window.location.toString() +
                '&scope=' + scope;
    };
```

When a user tries to login, they will be directed to a page that looks like this:

TODO: screenshot

Upon correctly entering in their credentials, Google will redirect them back to our application, giving us an access token. This token will allow us to retrieve user data.

Although we will use Google's login system, we still need our own server to store user data for our app. In the next section, we'll cover how to do this.

4.2 Storing User Data on the Server

With the user able to login, we now need to persist data on the server, retrieving and updating it as we please. Let's see how to do this. First, we want to retrieve a list of stories from the server. In our controller, let's do this:

```
…
http.get('/stories').success(function (stories) {
    $scope.stories = stories;
});</para>
// make a new story
http.put('/story', {
...
})

// show update</para>

var updateStory = function (story) {
    http.post('/story/' + story.id, { }, function () {
        console.log('story successfully updated');
    });
};
// show delete
http.delete('/story' + story.id, function () {
    console.log('story deleted');
});
```

Testing HTTP

To test the HTTP calls, we can use the mock http backend service

See: http://docs.angularjs.org/api/ngMock.$httpBackend

```
describe('MainCtrl', function () {
    var $httpBackend;</para>
    // instantiate and configure the mock backend
    beforeEach(inject(function($injector) {
      $httpBackend = $injector.get('$httpBackend');
      $httpBackend.when('GET', '/auth.py')
        .respond({userId: 'userX'}, {'A-Token': 'xxx'});
    }));
    it('should ...', function () {
    });
});
```

4.2.1 Errors

The server communication is pretty much set now, but there's still something important missing: error handling. What if your server goes down? Or your user disconnects from the internet? With the code we've just written, the app will silently fail. The user won't know that there's a problem, which will probably be frustrating. Worse, it's hard for you to address bug reports of "nothing happens when I click on things."

In order to know when a request to the server fails, we can use the the '.error()' callback. Let's modify the code for retrieving stories to handle errors. Recall that before, our code looked like this:

```
…
http.get('/stories').success(function (stories) {
$scope.stories = stories;
});
```

We can use the chaining syntax to register a callback for errors:

```
…
http.get('/stories').success(function (stories) {
    $scope.stories = stories;
}).error(function (reason) {
    console.log('Error: ' + reason);
});
```

There are many strategies for dealing with server errors. For instance, you can try to recover by waiting a few seconds, trying again, waiting a bit longer, trying again, etc.

```
…
var attempts = 0;
var getStories = function () {
```

```
    http.get('/stories').success(function (stories) {
        $scope.stories = stories;
        attempts = 0;
    }).error(function (reason) {
        attempts += 1;
        if (attempts < 10) {
            setTimeout(getStories, 5000 * attempts);
        } else {
            // display an error message
            console.log('error: ' + reason);
        }
    });
};
```

You might save the changes to localStorage and sync them to the server when a connection returns. Consider our "updateStory" function from earlier:

```
var updateStory = function (story) {
    http.post('/story/' + story.id, story, function () {
        console.log('story successfully updated');
    });
};
```

Let's see how we would implement this strategy so that if the initial request to post this fails, we try again.

```
var updateStory = function (story) {
    http.post('/story/' + story.id, story, function () {
        console.log('story successfully updated');
    }).error(function (reason) {
        // localStorage call
        console.log('story saved locally.');
    });
};
```

Besides recovering from errors, we want to display some information to the user. There are several ways that one might do this, but we want to highlight one technique. We like to use $rootScope.$broadcast from the service, and let the current route's controller choose how to handle the error.

Let's see how to implement this with the original updateStory. We start with this:

```
var updateStory = function (story) {
    http.post('/story/' + story.id, story, function () {
        console.log('story successfully updated');
    });
};
```

And then add the broadcast in the error handler:

```
var updateStory = function (story) {
    http.post('/story/' + story.id, story, function () {
        console.log('story successfully updated');
    }).error(function (reason) {
        $rootScope.broadcast('http:error', reason);
    });
};
```

Then, in our controllers, we want to listen for these broadcasts. We do that using `$scope.$on` like so:

```
$scope.on('error:http', function (ev, reason) {
    $scope.error(reason);
});

$scope.error = function (reason) {
    $scope.errorMessage = reason;
    // clear the message after 3 seconds (3000 ms)
    $timeout(function () {
        $scope.errorMessage = '';
    }, 3000);
};
```

How should we represent this visually?

```
// in the HTML
<div ng-show="errorMessage">
    <p>{{errorMessage}}</p>
    <a href ng-click="errorMessage">Close</a>
</div>
```

The reason for this is that there may be different ways to visually represent the error depending on the route. You may not want to show errors at all. Consider a user who is not logged in. An error with retrieving account information wouldn't concern them. For a logged in user, you might not want to display error information about this. In an administrative or debugging context, you might want more thorough logs. Decoupling the errors with `$broadcast` and `$on` allows you to handle each of these cases separately.

todo: screenshot

Testing Error Behavior

We saw earlier how to test http calls.

So now you've seen each part of CRUD: retrieving, creating, updating, and deleting models. As a bonus, we also saw how to do batch CRUD operations to multiple models.

We could duplicate this code for our other collections of models: statuses, comments, etc. But that would be really repetitive.

Using a convention called REST, we can eliminate a ton of repetitive code. Better yet, AngularJS provides a service called `$resource` that makes this a snap.

4.3 Getting RESTful with $resource

As you noticed in section 4.1, the code for making async HTTP requests was getting quite repetitive. When you're writing repetitive code, it's time to consider how you can refactor and simplify. One way to simplify is to adopt a convention for these requests. We mentioned REST in Chapter 3. As a reminder, REST is a convention for handling HTTP requests based on the request verb (GET, POST, UPDATE, or DELETE). The server Angello uses just happens to use REST. What a coincidence!

To do this, we're going to use an AngularJS service called `$resource`. Here's the quick version:

```
// make a new resource for stories
var Story = $resource('/:story');

// get all the stories
Story.query(function (stories) {
   $scope.stories = stories;
});

// update a story
$scope.stories[0].name = $scope.newName;
$scope.stories[0].$save();

// create a new story
var myNewStory = new StoryResource({
   name: 'this one is new!',
   user: 'Luke',
…
});

// save the story on our server
myNewStory.$save();
```

Using this technique, we significantly simplified our code!

Because not all apps use a RESTful backend, not all of them need $resource. To keep the size of required JS imports as small as possible, $resource is distributed in it's own file. In order to use $resource this in your application, be sure to include the `angular-resource.js` file in your `index.html`:

```
<script src="path/to/angular-resource.js"></script>
```

Let's use $resource to create the rest of our models in this app:

To recap, for servers that implement REST, $resource is a great way to be productive when interfacing with them. For other servers, you can use $http as described in 4.1 to manually setup and control requests.

4.4 Simplifying Control Flow with Promises

What are promises? Simply put, promises represent a value that is in transit. Promises are an abstraction around asynchronous behavior. For our app, we're going to use promises to represent the result of an asynchonous HTTP request. Let's look at some of the code we wrote earlier:

Example of Promises versus Callbacks.

History of Web Dev Futures vs Promises Event Emitters: (node.js-style) Promises/A: http://wiki.commonjs.org/wiki/Promises/A End History of Web Dev

$http.

How/When do you use them? When you have some combination of async operations, one after another...

It also makes it possible to re-use asynchronous operations easier. Consider getting a list of stories. You might want to get this list after a user logs in, or after a user changes from one view to another.

This is a bit facetious, because it turns out that you've been using promises all along! When you did `$http.(....).success()`, you were just binding a final "success handler" function. Similarly, `$http.(...).error()` was binding an "error handler" function.

Let's look at the flow for ...

Changing promises.

When you wanted to perform multiple asynchronous calls to the server one after another, you probably nested them like this:

```
$http( … ).success(function (res1) {
    $http( … ).success(function (res2) {
        $http( … ).success(function (res3) {
            // do something with res3
        });
    });
});
```

But you could have instead written this:

```
$http( … ).then(function (res1) {
    return $http( … )
}).then(function (res2) {
    return $http( … )
}).success(function (res3) {
    // do something with both results
});
```

What's the advantage here?

What happens when something on the chain 'fails'? The promise is said to be "rejected," and ....

What if you need to do multiple HTTP requests, but not necessarily one after another?

```
$q.all([
    $http( … ),
    $http( … )
]).success(function (results) {
    // do something with results
});
```

Here `results` will be an array with each item corresponding to the request in the array we passed to `$q.all`.

$resource also uses promises. Let's see how we can rewrite some of the code to ...

4.4.1 Reusing behavior with promises

You can wrap a function that returns a promise so that it can be more easily reused:

```
var getLoginStatus = function () {
    return $http( … )
}

var doLogin = function () {
    return $http( … )
}
getLoginStatus.
    then(doLogin).
    success(function () { … })
```

4.5 Security and CORS

AngularJS takes security very important and does its best to ensure that apps are safe by default. One such feature is Cross-Origin Resource Sharing (or CORS). Let's say you host an app on http://example.com. You wouldn't want someone to be able to GET or POST `http://example.com/user/sensitive_data` from . Thus browsers restrict `http://malicious-site.com` these types of communication, except in cases where some CORS policy explicitly allows the command. But you might want to allow other apps to connect to some public API, such as `http://example.com/api/something`.

Let's say that you want to use the API for a website like GitHub to grab information about your favorite open source project, AngularJS. In order to do this, you'll have to tell Github the domain of your app on their developer's site. This will cause GitHub's API to serve the resonses with the correct CORS headers.

And what if you wanted to write a web API that could be used by an AngularJS app? Just like GitHub, you'll have to make sure that users who sign up to use your API give a domain, and that this domain is correctly added to the allowed origins header in the resonses back from your server. For more in-depth information on CORS, we recommend reading the spec on W3C's site (url: http://www.w3.org/TR/cors/).

When possible, it's greatly preferred to use HTTPS for productions apps. The advantage of HTTPS is that it helps prevent "middle-man" attacks where someone intercepts communication between a user's browser and your app and is able to get sensitive information.

While we've done our best to outline security measures for the client-side, remember that your application is only as secure as its weakest link. This means you still have to consider security on your production web server, database, hosting account, etc. This subject is outside the scope of this book, but here are some useful resources:

4.6 Websockets

To recap, we've just seen how to communicate with a server over HTTP, as well as some considerations for security. Now we want our comment system to update automatically in real-time. Unfortunately, async HTTP requests don't let the server update the client when something has changed.

We could poll the server for changes by sending a request to get the

comments every so often, but that's not a very graceful solution. A lot of the time, the server won't have new infomation for the client, meaning that the additional HTTP requests are wasted. Furthermore, there's a tradeoff between the time between polls and the additional load on the web server. We want to give the server the ability to efficiently send messages to the client. Fortunately, a relatively new technology called WebSockets makes this much easier. WebSockets create a persistent two-way connection between client and server so that messages can be sent either way. AngularJS provides a module for async comm `$http` over HTTP, and `$resource` on top of that for using a RESTful server. What if you want to use WebSockets? Angular has no built in support for them, but you can extend Angular to add support pretty easily. Let's start with a factory:

```
angular.module('angello.websocket', []).
factory('socket', function ($rootScope) {

    var socket = new WebSocket();
    var listeners = {};

    socket.onmessage = function (message) {
        $rootScope.$apply(function () {
            listeners[message.name].forEach(function (listener) {
                listener(message.data);
            });
        });
};

return {
    on: function (name, fn) {
        listeners[name] = listeners[name] || [];
        listeners[name].push(fn);
    },
    send: function (message) {
        socket.send(JSON.stringify(message));
        }
    };
});
```

In our controller, we want to know when something has updated. To do that, we simply add the following code:

```
controller('MyCtrl', function ($scope, socket) {

    socket.on('myEvent', function (data) {
        $scope.data = data;
    });

});
```

We can now listen for server events and emit messages back to the server, but what happens when we change route? Try it and you'll see that although the route has changed, the events on the scope keep firing. We need to fix this so that as a user navigates through the route, only the relevant events from the server affect our application.

AngularJS has an event emitting system, so what if we were just to hook Socket.IO into that? Great idea!

```
// connect to the same place this is being hosted from
var socket = new WebSocket();

socket.onmessage = function (event) {

    // $apply notifies AngularJS that models may change
    $rootScope.$apply(function () {

        // broadcast the event to AngularJS scopes
        $rootScope.$broadcast('socket:' + event.name, event.data);
    });
});
```

Back to Angello, we now have some decisions to make. We can keep some of the asyncronous communication using HTTP, and some using WebSockets. This approach is fine, but it means that not all of our app will be real-time. We could convert all of the HTTP requests to use web sockets instead. This has the drawback of removing support for older browsers that do not have WebSockets. Another option would be to build an abstraction layer around `$http` and our`socket` service. This is a good solution, but it might be complex. Perhaps the best solution is to use a "shim" or "polyfill," which attempts to add WebSockets to browsers without them. This last technique is roughly what a project called "Socket.IO" does. Socket.IO will fall back to some of the techniques described earlier (like long polling) in the absense of WebSocket support.

5 Directives

Chapter Objectives:

- Understand what a directive is and why they are helpful.
- Understand the different kind of directives and what they are best suited for.
- Understand the main components that make up a directive.
- Discuss a few directives we are using in Angello.

5.1 Intro to Directives

Welcome to the directives chapter. We are going to get into one of the most powerful and important features of the AngularJS framework. In this chapter we are going to build three directives for Angello and discuss the techniques and reasoning behind each one. We will start out slow and work our way up in complexity to some really neat things you can use in your own web application.

5.1.1 What are Directives?

The major claim that AngularJS makes is that it is "HTML Enhanced for Web Apps". What does that mean to enhance HTML?

HTML was born from a mentality rooted in print media which was quite appropriate at the time. Browsers were limited and the best you could hope for was layout content on a page much like you would a magazine or a newspaper. But fast forward to the modern browser and HTML is incredibly limited and fixed when it comes to performing modern tasks like handling dynamic content, interactions, animations, etc.

AngularJS solves this limitation by allowing you to define your own HTML

behavior with directives. Directives are essentially custom HTML tags and attributes that you can create to do some very clever things. And by "clever", we mean "anything you want".

5.1.2 Why We NEED Directives

HTML is a fixed language in the sense that you get what is on the spec and that is that. And because it is fixed it is broken before even get started. Countless developers have tore their hair out working around the limitations that HTML has imposed upon them. Most of the time this comes by augmenting HTML with other technologies like CSS and JavaScript.

So why do we need directives. For your sanity. That's why. All kidding aside, when you can find a way to elegantly extend HTML to overcome its limitations you are now in a position to write modern web applications without resorting to circus tricks to get it done.

5.1.3 Why We WANT Directives

Setting the need to overcome the limitations of HTML aside for a moment, there is something elegant and artistic about writing code that is expressive and describes the domain you are in and the problems you are trying to solve. My absolute favorite feature of directives is that it allows you to turn your HTML into a Domain Specific Language.

We are building a project management board that tracks user stories. Wouldn't it be convenient if you had a tag called user-story that you could use in your markup. Would you have any question in your mind what that tag did. That is the beauty of directives!

5.2 Directives 101: A Quick Foundation

We are going to quickly lay the foundation as to what generally goes into a directive so we can start building our own as quickly as possible. Directives generally have three parts in them which are the `controller` function, `link` function and the Directive Definition Object (DDO). A directive will always have a DDO but may only have a link or controller function depending on the context.

FIGURE 5-1 When you condense directives into three main parts things get much simpler

The DDO is the foundation of the directive. It tells AngularJS how the directive should be handled during the compilation cycle and what it should. The DDO is

where you can set things like such as how it is going to be marked up in the HTML, how its scope is going to interact with the outside world and if it is going to use the existing HTML or load new HTML into the directive.

The controller function works just like controllers in the rest of your application. It is responsible for setting initializing state for the directive and defining functionality for it as well.

This is also where you would interact with a service if the directive needs to perform an action outside of its narrow area of focus.

The link function is where any DOM manipulation in your application goes. This is also where you put any initialization and interaction code for 3rd party plugins here. For instance, we are going to integrate with a jQuery plugin in our second example and this is where we initialize that plugin. The link function is also where you would capture any events emitted by the 3rd party plugin and process it for the rest of the AngularJS application.

5.2.1 The User Story Directive

Our first directive is going to a user story directive that is purposefully simplistic so that we can get our feet wet with the basic structure of how a directive is put together.

FIGURE 5-2 The user story directive

Create the Directive

Defining a directive is very similar to defining a controller or service in that you give it a name and a factory function to be return when the directive is needed. For instance, this is the basic syntax for creating a controller. Notice you are giving it a name as the first parameter of the controller method call and then a factory function as the second parameter.

```
myModule.controller('MainCtrl', function ($scope) {
  // ...
});
```

And when you define a directive, the pattern is the same. We are giving this directive a name of userstory and then we will start to build out the factory function as we progress.

```
myModule.directive('userstory', function () {
  // ...
});
```

And now let us fill out the directive with a link function, controller function and a definition object to lay the foundation for the rest of the functionality in the directive.

```
// app.js
myModule.directive('userstory', function () {
  var linker = function (scope, element, attrs) {
    // Pending
  };
  var controller = function ($scope) {
    // Pending
  };
  return {
    restrict: 'A',
    controller: controller,
    link: linker
  };
});
```

The Directive Definition Object

The definition object is really just an extension of the module pattern where you are returning an object to be instantiated during an AngularJS compilation cycle. The difference is that there is a specific API available to tell AngularJS exactly how the directive should behave. In the example above, we are stating that we want to restrict the directive to only be used as an attribute as noted by the line `restrict: 'A'`. And then we are indicating that we want to use the linker function and controller function as the link and controller functions respectively.

AngularJS is incredibly powerful with almost infinitely possibilities of what you can do but the 80/20 rule definitely applies. Nowhere is this more true than in the case of directives. There are some pretty exotic options you can invoke from the definition object but they are generally relegated to edge cases and rarely seen in the wild. For the sake of space and sanity we are going to endeavor to stay within the confines of what is reasonably useful.

The Link Function

Remember when we talked about the link function being the primary place to do DOM manipulation. Everything you need to accomplish this is delivered to you via an AngularJS care package in the form of the function parameters. The `scope` parameter is simply the scope of the current instantiation of the directive you are working with. It is worth mentioning that this is the same scope object as the `$scope` parameter in the controller function. The `element` parameter is the

element that the directive is declared on but wrapped in a jQuery object. The `attrs` parameter is an array of all of the attributes on the element that the directive was declared on.

NOTE: AngularJS ships with a subset of jQuery out of the box but if you have included jQuery in your project then AngularJS defers to that instance

The Controller Function

The controller function works almost exactly like a controller you would define on your application. Just as you want to segment DOM manipulation to the link function, you want to keep imperative logic in the controller. Because the link function and the controller function share the same scope object it is not uncommon to call a function in the controller from the link function. The only difference worth noting is that services are injected into the directive and are then available to the controller as opposed to injecting services into stand-alone controllers directly.

Use the Directive

Now that we have the skeleton of our user story directive in place, let's go ahead and actually use the directive in our markup.

This is the HTML that visually represents the user story currently.

```
// main.html
<li ng-repeat="story in stories | filter:{status:status.name}"
  class="story"
  ng-click="setCurrentStory(story)">

  <article class="{{story.type}}">
    <div class="modal-header">
      <h3>{{story.title}}</h3>
    </div>
    <div class="modal-body">
      <p>{{story.description}}</p>
    </div>
  </article>
</li>
```

This is the same HTML with the `userstory` directive defined on it.

```
// main.html
<li userstory
  ng-repeat="story in stories | filter:{status:status.name}"
  class="story"
  ng-click="setCurrentStory(story)">

  <article class="{{story.type}}">
```

```
    <div class="modal-header">
      <h3>{{story.title}}</h3>
    </div>
    <div class="modal-body">
      <p>{{story.description}}</p>
    </div>
  </article>
</li>
```

Notice the difference. Without turning into a cheerleader, being able to extend the user story HTML to include all of the extra functionality we are going to define in a single attribute is a really powerful feature!

Add DOM Event Handlers to Directive Link Function

Speaking of functionality, let's actually doing something with the directive already. We are going to start with the link function and add a slight fade when the user mouses over the user story and then restore it back when the user mouses out.

If you go to the jQuery website and look up the .mouseover() function, you will see a snippet of code that looks like this.

```
$('#outer').mouseover(function() {
  $('#log').append('<div>Handler for
    .mouseover() called.</div>');
});
```

We are doing to do something similar but you will soon see that the task is actually much easier within the link function.

```
// app.js
myModule.directive('userstory', function () {
  var linker = function (scope, element, attrs) {
    element.mouseover(function () {
      element.css({ 'opacity': 0.9 });
    }).mouseout(function () {
      element.css({ 'opacity': 1.0 })
    });
  };
  var controller = function ($scope) {
    // Pending
  };
  return {
    restrict: 'A',
    controller: controller,
    link: linker
  };
});
```

Because the element object is already a jQuery wrapped object, you can attach the event handler directly to the element object without having to query the DOM. You also do not have to worry about the DOM element being ready since the directive does not fire until the element has been added to the page and the compilation cycle has ran. It is like a premium valet service just for your DOM!

With that said, we are chaining two events together like this

```
element.mouseover(function () {
  // ...
}).mouseout(function () {
  // ...
});
```

and then setting the opacity in the event handler.

Create a Delete Story Method on AngelloModel

We are going to shift gears and add the ability to delete a story by clicking the delete button on the user story.

Practically speaking, an object should not be responsible for moving itself from the collection that it lives in and so we need to extend AngelloModel to accommodate this functionality since it owns all of the stories for the application.

NOTE: JavaScript has fairly limited functionality when it comes to manipulating arrays and collections and there are a some really great libraries to help bridge the gap. We are using Sugar in Angello which you can read about here: `http://sugarjs.com/`. Two other very popular libraries are Underscore `http://underscorejs.org/` and Lo-Dash `http://lodash.com/`.

```
// app.js
myModule.factory('AngelloModel', function ($rootScope) {
  // Code omitted
  var deleteStory = function (id) {
    stories.remove(function (s) {
      return s.id == id;
    });
  };
  return {
    // Code omitted
    deleteStory: deleteStory
  };
});
```

We have created the `deleteStory` method that accepts an id parameter. Using Sugar, we use the `remove` method to loop over the array and remove the story that matches the `id` parameter. We then make it available to the outside world by exposing it in the return object.

Create a Delete Story Method on Directive Controller

And now we need to create a `deleteStory` method and make it available to the directive. This is simply a matter of adding a `deleteStory` method on the `$scope` object and from there we call the `deleteStory` method we created on `AngelloModel` and pass the `id` parameter to it.

```
myModule.directive('userstory', function (AngelloModel) {
  // Code omitted
  var controller = function ($scope) {
    $scope.deleteStory = function (id) {
      AngelloModel.deleteStory(id);
    };
  };
  return {
    // Code omitted
    controller: controller,
  };
});
```

Also notice that all we had to do to access `AngelloModel` was to inject it into the factory method when we declared our directive. Dependency injection is a wonderful thing indeed.

Add Button to Call Delete Story on Click

The last thing we need to do is write up the `deleteStory` method on `$scope` to the actual view. This is familiar territory as we simply add a `button` with `ng-click` defined to call `deleteStory` and pass `story.id` in as the parameter.

```
<li userstory
  ng-repeat="story in stories | filter:{status:status.name}"
  class="story"
  ng-click="setCurrentStory(story)">
  <article class="{{story.type}}">
    <div class="modal-header">
      <button type="button"
        class="close"
        ng-click="deleteStory(story.id);">×</button>
      <h3>{{story.title}}</h3>
    </div>
```

```
    <div class="modal-body">
      <p>{{story.description}}</p>
    </div>
  </article>
</li>
```

Hooray. The user story directive is done. Time for a quick review before we move on.

We defined a directive including the three main parts: link function, controller function and the definition object.

We explored the parameters of the link function and why they make DOM manipulation so breezy.

We talked about how a controller function works pretty much like an application controller but shares $scope with the link function.

We showed how the directive definition object is used to define the directive. Well yeah. And more importantly that the 80/20 rule definitely applies here. Onward and upward!

5.3 A More Advanced Directive

So being able to fade HTML on mouseover is pretty awesome but how about something a bit more ambitious. You got it. We are going to flex our DOM chops and create a sortable directive that allows the user to drag a user story from one status column to another.

FIGURE 5-3 The sortable directive

The Sortable Directive

The sortable directive is essentially a wrapper around the jQuery UI sortable plugin with hooks to do Angello specific things.

The first the we need to do is to download the appropriate jQuery UI build to suit our needs.

Just Enough jQuery UI

It is a common reaction among developers when you start talking about jQuery UI to slam on the breaks and gasp "But. That library is huuuuuuuuge!" and that is a completely valid point if you were grabbing the entire library. The jQuery UI team are clever cats indeed and they allow you to do custom builds of jQuery UI with just the functionality that you need.

You can go to `http://jqueryui.com/download/` and select just the components that you want. In the case of the sortable directive, we only needed the sortable plugin and its supporting cast of characters. The custom jQuery UI

download tipped the scales at a trim 37 KB which is something we can live with.

FIGURE 5-4 Just enough jQuery UI to get the job done.

We then need to include the jQuery UI file in `index.html` like so.

```
// index.html
<script src="lib/jquery-ui-1.10.3.custom.min.js"></script>
```

Create the Sortable Directive

Now that the preliminary groundwork is done, it is time to actually create the sortable directive.

This follows the same pattern as before with us declaring the directive with the name `sortable` and a factory function that has a link function and a definition object to start out with.

```
myModule.directive('sortable', function () {
  var linker = function (scope, element, attrs) {
    element.sortable({
      items: 'li',</para>
      connectWith: '.list'</para>
    });
  };
  return {
    restrict: 'A',
    link: linker
  };
});
```

Within the link function, we are going to initialize the sortable plugin by called `sortable()` on the `element` object. We are going to send in a configuration object to the plugin with two options: `items: 'li'` and `connectWith: '.list'`. This tells the plugin that we only want to be able to drag `li` elements and items can be dragged from one list to another as long as it has a `.list` class.

NOTE: For a full list of configuration options for the sortable plugin checkout the documentation at `http://api.jqueryui.com/sortable/`.

Use the Sortable Directive

We will then insert the sortable directive in our HTML by adding it to the status columns.

```
// main.html
<ul class="list"
  sortable
```

```
  ng-repeat="status in statuses">

  <h3>{{status.name}}</h3>
  <hr>
  <li userstory
    ng-repeat="story in stories | filter:{status:status.name}"
    class="story"
    ng-click="setCurrentStory(story)">
    <!-- Omitted -->
  </li>
</ul>
```

Now that we have seen the HTML, the sortable plugin configuration should make sense. The user story elements are in fact `li` elements and that is what we want to be able to sort and the status columns have the `list` class and that is what we want to be able to sort between.

Technically we have accomplished the ability to drag user stories from one column to another but unfortunately our work is far from done. As a matter of course, you never want your DOM to hold the state of your application. This makes it hard to have a single source of truth and even harder to test. So what we need to do is capture the act of dragging a user story from one status column to another and update our domain model appropriately.

Update the User Story Status

jQuery UI thankfully has an event for just about event for just about every facet of dragging and dropping between lists. The event that we want to hook into is the `receive` event which is described in the jQuery UI API as "This event is triggered when a connected sortable list has received an item from another list." Bingo!

```
myModule.directive('sortable', function () {
  var linker = function (scope, element, attrs) {
    var status = scope.status.name; // Get the status of the column
    element.sortable({
      items: 'li',
      connectWith: '.list',
      receive: function (event, ui) {
        var curScope = angular.element(ui.item).scope();
        scope.$apply(function () {
          curScope.story.status = status; // Update the status
        });
      }
    });
  };
  return {
    restrict: 'A',
```

```
    link: linker
  };
});
```

The first thing we need to do is to create a reference to the status of the column the sortable directive is declared on. We are accomplishing that with this bit of code here: `var status = scope.status.name;`.

The next thing we need to do is to add an event handler for the receive event which we are doing here: `receive: function (event, ui) { /* ... */ }`.

Now that we are hooked into the receive event, we need to get the scope of the item that was dropped. We are using `angular.element` to get the `scope` of `ui.item`like so:

```
var curScope = angular.element(ui.item).scope();
```

Almost there. One more thing left in this code example to cover and that is to update the `currentScope.story.status` to the `status` of the column it was dropped on. Because all of this is happening in within the realm of jQuery UI and not AngularJS we need to wrap our model mutation in `scope.$apply` so that AngularJS knows to perform a digest cycle. This is why `curScope.story.status = status;` is within `scope.$apply(function () { /* .. */ });`.

Addressing the Disconnect Between DOM and Model Part 1

Another interesting aspect of this directive thus far is that you can drag a user story to another column and before another user story and in the DOM it is represented in that order but the actual `AngelloModel.stories` array is completely oblivious to this "reality". We need to reconcile this disparity but updating the model first and letting the DOM reflect the data.

We need to a way to essentially tell Angello "Hey we dropped this story after this previous story and so go ahead and insert it after the previous story." And so to accomplish this we are going to create an `insertStoryAfter` method on `AngelloModel`.

```
myModule.factory('AngelloModel', function ($rootScope) {
  // Code omitted
  var insertStoryAfter = function(story, prevStory) {
```

```
    stories = stories.remove(function(t) {
      return t['id'] == story.id;
    });
    stories = stories.add(story, stories.findIndex(prevStory) + 1);
  };

  return {
    // Code omitted
    insertStoryAfter: insertStoryAfter</para>
  };
});
```

This method takes two arguments `story` and `prevStory` which are the story that was dropped and the story previous to that. Based on those two objects we have enough information to manipulate the stories array to accurately reflect the event that has occurred.

We are going to use the Sugar `remove` method to remove the `story` that was dropped from its current position in the array. We are then going to use the Sugar `findIndex` method to find the index of the `prevStory`. From there, we use Sugar `add` to insert the `story` object into the index of the `prevStory` plus one.

NOTE: Operations like these are a great candidate for writing rigorous unit tests around.

Addressing the Disconnect Between DOM and Model Part 2

Now that we have a method in place to update the model we need to make it available by injecting `AngelloModel` into the directive and then actually calling `AngelloModel.insertStoryAfter` at the appropriate time.

The first step is as simple as adding AngelloModel as a parameter to the factory function when you declare the directive.

```
<para>myModule.directive('sortable', function (AngelloModel) {
  var linker = function (scope, element, attrs) {
    var status = scope.status.name; // Get the status of the column

    element.sortable({
      items: 'li',
      connectWith: ".list",
      receive: function (event, ui) {
        var prevScope = angular.element(ui.item.prev()).scope();
        var curScope = angular.element(ui.item).scope();
        scope.$apply(function () {
          AngelloModel
            .insertStoryAfter(curScope.story, prevScope.story);
          curScope.story.status = status; // Update the status
        });
```

```
      }
    });
  };
  return {
    restrict: 'A',
    link: linker
  };
});
```

We already have `curScope` and so all we need is the scope of the item previous to `ui.item` so that we have `access` to both of the story objects we need. Thankfully, jQuery has a `prev()` method to get the previous child item in the DOM. And so with that knowledge in hand, we can modify the code we used to get `curScope` to set `prevScope` like so:

```
angular.element(ui.item.prev()).scope();
```

And now that we have both scope objects, we can can pass their story objects for processing like this: `AngelloModel.insertStoryAfter(curScope.story, prevScope.story);`

And that completes our foray into the sortable directive. Let us review real quick what we covered.

We downloaded a custom build of jQuery UI that included just what we needed to make sortable work.

We showed how to instantiate the jQuery sortable plugin in our link function and pass it configuration options to control how it behaves.

We added a `received` event handler to respond to the sortable event and perform further actions.

We wrapped the code that changed AngularJS data in a `scope.$apply` so that AngularJS knew to perform a digest cycle.

We talked about not letting the DOM hold state in case it got out of sync with the model but rather capturing the event and processing it in the model and then letting the DOM reflect the new state of the model.

We showed some more ways to manipulate data structures using Sugar.js and a way to get a previous sibling using the jQuery method `prev()`.

5.4 Integrating with 3rd Party Libraries Again!

So far we have written two directives that relied on jQuery and jQuery UI which has been a really pleasant experience so far but let's try for something a bit more ambitious.

We are going to build a directive that integrates with Flot and displays user story statistics.

NOTE Flot is a gorgeous graphing library that is built in JavaScript. You can read more about Flot at `http://www.flotcharts.org/`.

FIGURE 5-5 We may not have a lot but we've got a Flot!

Install Flot

The first thing we need to do is to install Flot in our application. Flot comes with a core library and then you add a plugin for the visualization you want to accomplish. In our case, we want to use the categories plugin because we want to segment our data based on categories.

We are going to make Flot available to our application by adding references to the appropriate files in the code below.

```
// index.html
<script src="lib/flot/jquery.flot.js"></script>
<script src="lib/flot/jquery.flot.categories.js"></script>
```

The next two steps are going to go by pretty quick now that we have established muscle memory but they are necessary steps to getting started.

Build the Directive

Call us creatures of habit but we are going to start this directive the same way we kicked off the other two directives with the basic skeleton.

```
// app.js
myModule.directive('chart', function () {
  var linker = function (scope, element, attrs) {
    // Link goes here
  };
  var controller = function ($scope) {
    // Controller goes here
  };
  return {
    restrict: 'A',
    controller: controller,
    link: linker
  };
});
```

Use the Directive

We are going to use the chart directive in two places and so we need to add it the chart directive in two places.

```
// dashboard.html
<div id="content">
  <h3>User Stories by Status</h3>

  <div class="chart-container">
    <div chart class="chart-placeholder"></div>
  </div>
  <h3>User Stories by Type</h3>

  <div class="chart-container">
    <div chart class="chart-placeholder"></div>
  </div>
</div>
```

Massage Our Data

Technically the directive is working but it doesn't actually do much yet. The interesting challenge with Flot integration is that Flot expects a very specific data structure to properly render the chart.

Below you can see the data structure that it expects which was pulled from the Flot sample files.

```
var data = [
  ["January", 10],
  ["February", 8],
  ["March", 4],
  ["April", 13],
  ["May", 17],
  ["June", 9]
];
```

To make this work with the user stories, we need to come up with a data structure that looks like array below.

```
var data = [
  ["Log", 1],
  ["To Do", 2],
  ["In Progress", 0],
```

```
  ["Code Review", 1],
  ["QA Review", 0],
  ["Verified", 1],
  ["Done", 1]
];
```

We are going to delve into a pretty heavy utility function that is going to produce this very data structure. Bare with us and it will make sense in in just moment and balance will be restored in the universe.

We are going to unpack a pretty heavy utility function that is used to produce the data structure that Flot needs. It is easiest to articulate what it does in natural language before we start to look at the code. In a nutshell we are telling `parseDataForCharts` to "loop over Array A and on each iteration get the value that exists at Property A and when you have that value go to Array B and count how many times that value occurs on Property B m'kay?"

```
myModule.directive('chart', function () {
  var parseDataForCharts = function(sourceArray, sourceProp,
    referenceArray, referenceProp) {

    var data = [];
    referenceArray.each(function (r) {
      var count = sourceArray.count(function (s) {
          return s[sourceProp] == r[referenceProp];
      });
      data.push([r[referenceProp], count]);
    });

    return data;
  };
  // Code omitted
});
```

The two main pieces of this function are the Sugar each method that iterates over the referenceArray and the Sugar count method that counts the matches between `sourceArray[ sourceProp ]` and `referenceArray[referenceProp]`. From there it is a matter of of pushing the result in the right format into the data array for the return statement. Flot expects a format of of `[['property', number], ['property', number], etc]` which we are honoring on this line `data.push([r[referenceProp], count]);`.

This was the part of the directive that got the most mindshare when it was being

written and now that we have our data in the format we need it is going to be pretty much a matter of wiring up the pieces.

It Is Time We Had the "Isolated Scope Talk"

Scope by default prototypically inherits from its parent and if we were to reference a property on the child scope that it did exist, AngularJS would walk up the prototype chain until it found it. This is a non-issue in most cases but there are times when you do want to isolate the directive's scope entirely from its parent scope to completely mitigate potential side effects.

AngularJS allows you to accomplish this via isolated scope which creates an ironclad perimeter around the directive's scope and then it is the responsibility of the developer to define exactly how the directive will communicate with the outside world. This essentially provides an API for your directive with clearly defined channels of communication.

There are three types of isolated scope; attribute isolated scope, binding isolated scope and expression isolated scope. Attribute isolated scope binds on a single attribute and the communication is only from the parent to the child. The value that you define is interpreted as a string and therefore is really only suitable for simple values. Binding defined isolated scope enables two communication between the parent and child scope and can bind to collections and objects as well as simple values. This is the most common type of isolated scope and it what most of the built in AngularJS directives use. Expression isolated scope works by allowing the child to execute an expression on the parent. Although not as common, expression isolated scope is a great way to dynamically attach behavior to your application by letting the parent define the expression to be execute when the child calls the expression defined in the isolated scope.

NOTE: This was an elevator pitch for isolated scope and the topic warrants an entire discussion dedicated to it. Check out Appendix X for a more thorough examination of isolated scope and how it works.

Let us examine an instance of isolated scope as it relates to our project before we go any further.

```
myModule.directive('chart', function () {
  // Code omitted
  return {
    restrict: 'A',
    controller: controller,
    link: linker,
    scope: {
      sourceArray: '=',
```

```
      referenceArray: '='
    }
  };
});
```

We want to bind to `sourceArray` and `referenceArray` so that if they change, we will know about it in the directive. Conversely, if we modified the arrays in the the directive, we would want the outside world to know as well. Isolated scope is accomplished on the definition object by passing in an object with the properties you want to expose and some special syntax to define the kind of isolation you want. In our case, we want binding isolated scope and so we are going to use an = sign to indicate this.

NOTE: Attribute isolated scope is defined with an @ symbol, binding isolated scope is defined with an equals sign ("=") and expression isolated scope is defined with an ampersand ("&"). If the property name you are isolating is the same to the outside world as what you are using internally, no other configuration is necessary. If for some reason you wanted to use a different name internally then the format is as follows `externalProperty: '=internalProperty'`.

We then exercise our right to isolated scope in the HTML as seen in the markup below.

```
<div id="content">
  <h3>User Stories by Status</h3></para>

  <div class="demo-container">
    <div chart class="demo-placeholder"
      source-array="stories"
      source-prop="status"
      reference-array="statuses"
      reference-prop="name">
    </div>
  </div>
  <h3>User Stories by Type</h3>
  <div class="demo-container">
    <div chart class="demo-placeholder"
      source-array="stories"
      source-prop="type"
      reference-array="types"
      reference-prop="name">
    </div>
  </div>
</div>
```

We told the outside world to put any array in the `source-array` attribute

and the directive will internally treat it as `sourceArray`.

NOTE: AngularJS converts JavaScript camelcase into snakecase in HTML. This is why in the directive we are using `sourceArray` and on the HTML it is `soure-array`.

You may have noticed that we are also defining source-prop and reference-prop on the directive element but we have not set up isolated scope around these properties. This was a design decision that did not warrant isolated scope because those properties only need to be read once and it is not worth incurring the cost of binding in any direction. We will read them from the `attrs` array in the next section.

Grand Finale. Breathe Life Into Flot

And now that we have created communication channels with all of the data that we need, it is time to lock this down in style. It is time to actual hook up Flot.

```
myModule.directive('chart', function () {
  // Code omitted
  var linker = function (scope, element, attrs) {
    scope.data = parseDataForCharts(scope.sourceArray,
      attrs['sourceProp'],
      scope.referenceArray,
      attrs['referenceProp']);

    if(element.is(":visible")){
      $.plot(element, [ scope.data ], {
        series: {
          bars: {
            show: true,
            barWidth: 0.6,
            align: "center"
          }
        },
        xaxis: {
          mode: "categories",
          tickLength: 0
        }
      });
    }
  };

  // Code omitted
});
```

The first thing we do is parse the data via our `parseDataForCharts` method passing in our isolated scope arrays and the properties values we are reading off of the `attr` array. We set the result of that method call to the `data`

property on `scope` so that we can use it when we spin up the Flot chart.

The one caveat about Flot is that the element is is drawing in has to be visible or it completely falls apart. That is why we are using `element.is(':visible')` as a condition for proceeding any further. And then we instantiate Flot with `$.plot(element, [ scope.data ], { });` passing in `element, scope.data` and the appropriate configuration object.

NOTE:

When this directive was being created we used the configuration object from the Flot sample files and it works right out of the box. For fun, we encourage you to explore the different options that Flot has available such as mouse interaction, colors, etc.

Stop. Review time!

We performed some more array wizardry with Sugar to get our data into a format that Flot could use.

We talked about isolated scope and the benefits it provides.

We instantiated Flot to show two completely different sets of data.

5.5 Conclusion

And we have crossed the finish line with three directives under our belt built from the ground up. While we sacrificed covering the entire academic tome of directives in favor of illustrating practical, working examples, we really hope that you have started to see the immense power of directives and dig deeper.