CHAPTER 1
Introducing AngularJS

The Internet has come a long way since its inception. Consumption-oriented, non-interactive websites started moving toward something users interacted with. Users could respond, fill in details, and eventually access all their mail on websites. Concurrent usage, offline support, and so many other things became basic features, and the size and scope of client-side applications has kept on accelerating and increasing.

As applications have gotten bigger, better, and faster, so has the complexity a developer has to manage. A pure JavaScript/jQuery solution would not always have the right structure to ensure a rapid speed of development or long-term maintainability. Projects became heavily dependent on having a great software engineer to set up the initial framework. Even then, modularity, testability, and separation of concerns may not make it into a project. Testing and reliability were often pushed to the backburner in such cases.

AngularJS was started to fill this basic need. Could we provide a standard structure and meta-framework within which web applications could be developed reliably and quickly? Could the same software engineering concepts like testable code, separation of concerns, MVC (Model-View-Controller) (or rather, MVVM), and so on be applied to JavaScript applications? Could we have the best of both worlds—the succinctness of JavaScript and the pleasure of rapid, maintainable development? We think so, but we'll let you be the final judge as we walk through AngularJS throughout the rest of this book.

By the end of this chapter, we will build a basic AngularJS "hello world" example to get a sense of some common concepts and philosophies behind AngularJS. We will also see how to bootstrap and convert any HTML into an AngularJS application, and see how to use common data-binding techniques in AngularJS.

Introducing AngularJS

AngularJS is a superheroic JavaScript MVC framework for the Web. We call it superheroic because AngularJS does so much for us that we only have to focus on our core application and let AngularJS take care of everything else. It allows us to apply standard, tried-and-tested software engineering practices traditionally used on the server side in client-side programming to accelerate frontend development. It provides a consistent scalable structure that makes it a breeze to develop large, complex applications as part of a team.

And the best part? It's all done in pure JavaScript and HTML. No need to learn another new programming or templating language (though you do have to understand the MVC and MVVM paradigms of developing applications, which we briefly cover in this book). And how does it fulfill all these crazy and wonderful, seemingly impossible-to-satisfy promises?

The AngularJS philosophy is driven by a few key tenets that drive everything from how to structure your application, to how your applications should be hooked together, to how to test your application and integrate your code with other libraries. But before we get into each of these, let's take a look at why we should even care in the first place.

What Is MVC (Model-View-Controller)?

The core concept behind the AngularJS framework is the MVC architectural pattern. The Model-View-Controller pattern (or MVVM, which stands for Model-View-ViewModel, which is quite similar) evolved as a way to separate logical units and concerns when developing large applications. It gives developers a starting point in deciding how and where to split responsibilities. The MVC architectural pattern divides an application into three distinct, modular parts:

- The *model* is the driving force of the application. This is generally the data behind the application, usually fetched from the server. Any UI with data that the user sees is derived from the model, or a subset of the model.
- The *view* is the UI that the user sees and interacts with. It is dynamic, and generated based on the current model of the application.
- The *controller* is the business logic and presentation layer, which peforms actions such as fetching data, and makes decisions such as how to present the model, which parts of it to display, etc.

Because the controller is responsible for basically deciding which parts of the model to display in the view, depending on the implementation, it can also be thought of as a *viewmodel*, or a *presenter*.

At its core, though, each of these patterns splits responsibilities in the application into separate subunits, which offers the following benefits:

- Each unit is responsible for one and only one thing. The model is the data, the view is the UI, and the controller is the business logic. Figuring out where the new code we are working on belongs, as well as finding prior code, is easy because of this single responsibility principle.
- Each unit is as independent from the others as possible. This makes the code much more modular and reusable, as well as easy to maintain.

AngularJS Benefits

We are going to make some claims in this section, which we will expand on in the following section when we dive into how AngularJS makes all this possible:

- AngularJS is a Single Page Application (SPA) meta-framework. With client-side templating and heavy use of JavaScript, creating and maintaining an application can get tedious and onerous. AngularJS removes the cruft and does the heavy lifting, so that we can focus solely on the application core.
- An AngularJS application will require fewer lines of code to complete a task than a pure JavaScript solution using jQuery would. When compared to other frameworks, you will still find yourself writing less boilerplate, and cleaner code, as it moves your logic into reusable components and out of your view.
- Most of the code you write in an AngularJS application is going to be focused on business logic or your core application functionality, and not unnecessary routine cruft code. This is a result of AngularJS taking care of all the boilerplate that you would otherwise normally write, as well as the MVC architecture pattern.
- AngularJS's declarative nature makes it easier to write and understand applications. It is easy to understand an application's intent just by looking at the HTML and the controllers. In a sense, AngularJS allows you to create HTMLX (instead of relying on HTML5 or waiting for HTML6, etc.), which is a subset of HTML that fits your needs and requirements.
- AngularJS applications can be styled using CSS and HTML independent of their business logic and functionality. That is, it is completely possible to change the entire layout and design of an application without touching a single line of JavaScript.
- AngularJS application templates are written in pure HTML, so designers will find it easier to work with and style them.
- It is ridiculously simple to unit test AngularJS applications, which also makes the application stable and easier to maintain over a longer period of time. Got new

features? Need to make changes to existing logic? All of it is a breeze with that rock-solid bed of tests underneath.

- We don't need to let go of those jQueryUI or BootStrap components that we love and adore. AngularJS plays nicely with third-party component libraries and gives us hooks to integrate them as we see fit.

The AngularJS Philosophy

There are five core beliefs to which AngularJS subscribes that enable developers to rapidly create large, complex applications with ease:

Data-driven (via data-binding)
: In a traditional server-side application, we create the user interface by merging HTML with our local data. Of course, this means that whenever we need to change part of the UI, the server has to send the entire HTML and data to the client yet again, even if the client already has most of the HTML.

  With client-side Single Page Applications (SPAs), we have an advantage. We only have to send from the server to the client the data that has changed. But the client-side code still has to update the UI as per the new data. This results in boilerplate that might look something like the following (if we were using jQuery). First, let's look at some very simple HTML:

```
Hello  <span id="name"></span>
```

  The JavaScript that makes this work might look something like this:

```
var updateNameInUI = function(name) {
    $('#name').text(name);
};

// Lots of code here...
// On initial data load
updateNameInUI(user.name);

// Then when the data changes somehow
updateNameInUI(updatedName);
```

  The preceding code defines a `updateNameInUI` function, which takes in the name of the user, and then finds the UI element and updates its `innerText`. Of course, we would have to be sure to call this function whenever the `name` value changes, like the initial load, and maybe when the user logs out and logs back in, or if he edits his name. And this is just one field. Now imagine dozens of such lines across your entire codebase. These kinds of operations are very common in a CRUD (Create-Retrieve-Update-Delete) model like this.

Now, on the other hand, the AngularJS approach is driven by the model backing it. AngularJS's core feature—one that can save thousands of lines of boilerplate code—is its data-binding (both one-way and two-way). We don't have to waste time funneling data back and forth between the UI and the JavaScript in an AngularJS application. We just bind to the data in our HTML and AngularJS takes care of getting its value into the UI. Not only that, but it also takes care of updating the UI whenever the data changes.

The exact same functionality in an AngularJS application would look something like this:

```
Hello <span>{{name}}</span>
```

Now, in the JavaScript, all that we need to do is set the value of the `name` variable. AngularJS will take care of figuring out that it has changed and update the UI automatically.

This is one-way data-binding, where we take data coming from the server (or any other source), and update the Document Object Model (DOM). But what about the reverse? The traditional way when working with forms—where we need to get the data from the user, run some processing, and then send it to the server—would look something like the following. The HTML first might look like this:

```
<form name="myForm" onsubmit="submitData()">
  <input type="text" id="nameField"/>
  <input type="text" id="emailField"/>
</form>
```

The JavaScript that makes this work might look like this:

```
// Set data in the form
function setUserDetails(userDetails) {
    $('#nameField').value(userDetails.name);
    $('#emailField').value(userDetails.email);
}

function getUserDetails() {
    return {
        name: $('#nameField').value(),
        email: $('#emailField').value()
    };
}

var submitData = function() {
  // Assuming there is a function which makes XHR request
  // Make POST request with JSON data
  makeXhrRequest('http://my/url', getUserDetails());
};
```

In addition to the layout and templating, we have to manage the data flow between our business logic and controller code to the UI and back. Any time the data

changes, we need to update the UI, and whenever the user submits or we need to run validation, we need to call the `getUserDetails()` function and then do our actual core logic on the data.

AngularJS provides two-way data-binding, which allows us to skip writing this boilerplate code as well. The two-way data-binding ensures that our controller and the UI share the same model, so that updates to one (either from the UI or in our code) update the other automatically. So, the same functionality as before in AngularJS might have the HTML as follows:

```
<form name="myForm" ng-submit="ctrl.submitData()">
  <input type="text" ng-model="user.name"/>
  <input type="text" ng-model="user.email"/>
</form>
```

Each input tag in the HTML is bound to an AngularJS model declared by the ng-model attribute (called directives in AngularJS). When the form is submitted, AngularJS hooks on by triggering a function in the controller called `submitData`. The JavaScript for this might look like:

```
// Inside my controller code
this.submitData = function() {
    // Make Server POST request with JSON object
    $http.post('http://my/url', this.user);
};
```

AngularJS takes care of the two-way data-binding, which entails getting the latest values from the UI and updating the `name` and `email` in the `user` object automatically. It also ensures that any changes made to the `name` or `email` values in the `user` object are reflected in the DOM automatically.

Because of data-binding, in an AngularJS application, you can focus on your core business logic and functionality and let AngularJS do the heavy lifting of updating the UI. It also means that it requires a shift in our mindset to develop an AngularJS application. Need to update the UI? Change the model and let AngularJS update the UI.

Declarative

A single-page web application (also known as an AJAX application) is made up of multiple separate HTML snippets and data stitched together via JavaScript. But more often than not, we end up having HTML templates that have no indication of what they turn into. For example, consider HTML like the following:

```
<ul class="nav nav-tabs">
  <li>Home</li>
  <li class="selected">Profile</li>
</ul>

<div class="tab1">
```

```
    Some content here
</div>
<div class="tab2">
    <input id="startDate" type="text"/>
</div>
```

Now, if you are used to certain HTML constructs or are familiar with jQuery or similar frameworks, you might be able to divine that the preceding HTML reflects a set of tabs, and that the second tab has an input field that needs to become a datepicker. But none of that is actually mentioned in the HTML. It is only because there is some JS and CSS in your codebase that has the task of converting these `li` elements into tabs, and the `input` field into a datepicker.

This is essentially the *imperative paradigm*, where we tell the application exactly how to do each and every action. We tell it to find the element with class `nav-tabs` and make it a tab component, then to select the first tab by default. We accomplish this entirely in our JavaScript code and not where the actual HTML needs to change. The HTML does not reflect any of this logic.

AngularJS instead promotes a *declarative paradigm*, where you declare right in your HTML what it is you are trying to accomplish. This is done through something that AngularJS calls *directives*. Directives basically extend the vocabulary of HTML to teach it new tricks. We let AngularJS figure out how to accomplish what we want it to do, whether it is creating tabs or datepickers. The ideal way to write the previous code in AngularJS would be something like the following:

```
<tabs>
  <tab title="Home">Some content here</tab>
  <tab title="Profile">
    <input type="text"
           datepicker
           ng-model="startDate"/>
  </tab>
</tabs>
```

The AngularJS-based HTML uses `<tab>` tags, which tells AngularJS to figure out how to render the tabs component, and declares that the `<input>` is a `datepicker` that is bound to an AngularJS model variable called `startDate`.

There are a few advantages to this approach:

- It's declarative, so just by looking at the HTML we can immediately figure out that there are two tabs, one of which has a datepicker inside of it.
- The business logic of selecting the current tab, unselecting the other tabs, and hiding and showing the correct content is all encapsulated inside the tab directive.

- Similarly, any developer who wants a datepicker does not have to know whether we are using jQueryUI, BootStrap, or something else underneath. It separates out the usage from the implementation so there is a clear separation of concerns.
- Because the entire functionality is encapsulated and contained in one place, we can make changes in one central place and have it affect all usages, instead of finding and replacing each API call manually.

Separate your concerns

AngularJS adopts a Model-View-Controller (MVC)-like pattern for structuring any application. If you think about it, there are three parts to your application.

There is the actual data that you want to display to the user, or get the user to enter through your application. This is the model in an AngularJS project, which is mostly pure data, and represented using JSON objects.

Then there is the user interface or the final rendered HTML that the user sees and interacts with, which displays the data to the user. This is the view.

Finally, there is the actual business logic and the code that fetches the data, decides which part of the model to show to the user, how to handle validation, and so on —core logic specific to your application. This is the controller for an AngularJS application.

We think MVC or an MVC-like approach is neat for a few solid reasons:

- There is a clear separation of concerns between the various parts of your application. Need some business logic? Use the controller. Need to render something differently? Go to the view.
- Your team and collaborators will have an instant leg up on understanding your codebase because there is a set structure and pattern.
- Need to redesign your UI for any reason? No need to change any JavaScript code. Need to change how something is validated? No need to touch your HTML. You can work on independent parts of the codebase without spilling over into another.
- AngularJS is not completely MVC; the controller will never have a direct reference to the view. This is great because it keeps the controller independent of the view, and also allows us to easily test the controller without needing to instantiate a DOM.

Because of all of these reasons, MVC allows you develop and scale your application in a way that is easy to maintain, extend, and test.

Dependency Injection

AngularJS is the one of the few JavaScript frameworks with a full-fledged Dependency Injection system built in. Dependency Injection (discussed in Chapter 5) is the concept of asking for the dependencies of a particular controller or service, instead of instantiating them inline via the `new` operator or calling a function explicitly (for example, `DatabaseFactory.getInstance()`). Some other part of your code becomes responsible (in this case, the *injector*) for figuring out how to create those dependencies and provide them when asked for.

This is helpful because:

- The controller, service, or function asking for the dependency does not need to know how to construct its dependencies, and traverse further up the chain, however long it might be.
- It's explicit, so we immediately know what we need before we can start working with our piece of code.
- It makes for super easy testing because we can replace heavy dependencies with nicer mocks for testing. So instead of passing an `HttpService` that talks to the real server, we pass in a `MockHttpService` that talks to a server created in memory.

  Dependency Injection in AngularJS is used across all of its parts, from controllers and services to modules and tests. It allows you to easily write modular, reusable code so that you can use it cleanly and simply as needed.

Extensible

We already mentioned directives in the previous section when we talked about AngularJS's declarative nature. Directives are AngularJS's way of teaching the browser and HTML new tricks, from handling clicks and conditionals to creating new structure and styling.

But that is just the built-in set of directives. AngularJS exposes the same API that it uses internally to create these directives so that anyone can extend existing directives or create their own. We can develop robust and complex directives that integrate with third-party libraries like jQueryUI and BootStrap, to name a few, to create a language that is specific to our needs. We'll see how to create our own directives in Chapter 11.

The bottom line is that AngularJS has a great core set of directives for us to get started, and an API that allows us to do everything AngularJS does and more. Our imagination is really the only limit for creating declarative, reusable components.

Test first, test again, keep testing

A lot of the benefits that we mentioned previously actually stem from the singular focus on testing and testability that AngularJS has. Every bit and piece of AngularJS

is designed to be testable, from its controllers, services, and directives to its views and routes.

Between Dependency Injection and the controller being independent of references to the view, the JS code that we write in an AngularJS application can easily be tested. Because we get the same Dependency Injection system in our tests as in our production code, we can easily instantiate any service without worrying about its dependencies. All of this is run through our beautiful, insanely fast test runner, Karma.

Of course, to ensure that our application actually works end to end, we also have Protractor, which is a WebDriver-based end-to-end scenario runner designed from the ground up to be AngularJS-aware. This means that we will not have to write any random waits and watches in our end-to-end test, like waiting for an element to show or waiting for five seconds after a click for the server to respond. Protractor is able to hook into AngularJS and figure out when to proceed with the test, leaving us with a suite of solid, deterministic end-to-end tests.

We will start using Karma, and talk about how to set it up and get started in Chapter 3, and Protractor in Chapter 14. So there really is no excuse for your AngularJS application not to be completely tested. Go ahead, you and your teammates will thank yourself for it.

Now that you have had a brief overview of what makes AngularJS great, let's see how to get started with writing your own AngularJS applications.

Starting Out with AngularJS

Starting an AngularJS application has never been easier, but even before we jump into that, let's take a moment to answer a few simple questions to help you decide whether or not AngularJS is the right framework for you.

What Backend Do I Need?

One of the first questions we usually get is regarding the kind of a backend one would need to be able to write an AngularJS application. The very short answer is: there are no such requirements.

AngularJS has no set requirements on what kind of a backend it needs to work as a Single-Page Application. You are free to use Java, Python, Ruby, C#, or any other language you feel comfortable with. The only thing you do need is a way of communicating back and forth with your server. Ideally, this would be via XHR (XML HTTP requests) or sockets.

CHAPTER 2

Basic AngularJS Directives and Controllers

We saw in Chapter 1 how to create a very simple and trivial AngularJS application, which was basically the "hello world" of the AngularJS world. In this chapter, we will expand on that example.

We explore AngularJS modules and controllers, and create our very own controllers. Then we use these controllers to load data or state into our application, and manipulate the HTML to perform common tasks such as displaying an array of items in the UI, hiding and showing elements conditionally, styling HTML elements based on certain conditions, and more.

AngularJS Modules

The very first thing we want to introduce is the concept of *modules*. Modules are AngularJS's way of packaging relevant code under a single name. For someone coming from a Java background, a simple analogy is to think of modules as packages.

An AngularJS module has two parts to it:

- A module can define its own controllers, services, factories, and directives. These are functions and code that can be accessed throughout the module.
- The module can also depend on other modules as *dependencies*, which are defined when the module is instantiated. What this means is that AngularJS will go and find the module with that particular name, and ensure that any functions, controllers, services, etc. defined in that module are made available to all the code defined in this module.

In addition to being a container for related JavaScript, the module is also what AngularJS uses to bootstrap an application. What that means is that we can tell AngularJS what

module to load as the main entry point for the application by passing the module name to the `ng-app` directive.

Let's clear this up with the help of a few examples.

This is how we define a module named `notesApp`:

```
angular.module('notesApp', []);
```

The *first argument* to the module function in AngularJS is the *name of the module*. Here, we define a module named `notesApp`. The *second argument* is an *array of module names* that this module depends on. Do note the empty square brackets we pass as the second argument to the function. This tells AngularJS to create a new module with the name `notesApp`, with no dependencies.

This is how we define a module named `notesApp`, which depends on two other modules: `notesApp.ui`, which defines our UI widgets, and `thirdCompany.fusioncharts`, which is a third-party library for charts:

```
angular.module('notesApp',
    ['notesApp.ui', 'thirdCompany.fusioncharts']);
```

If we want to load an existing module that has already been defined in some other file, we use the the `angular.module` function with just the first argument, as follows:

```
angular.module('notesApp');
```

This line of code tells AngularJS to find an *existing* module named `notesApp`, and to make it available to use, add, or modify in the current file. This is how we refer to the same module across multiple files and add code to it.

There are two common mistakes to watch out for:

- Trying to define a module, but forgetting to pass in the second argument. This would cause AngularJS to try to look up a module instead of defining one, and we would get an error ("No module found").
- Trying to load a module from another file to modify, but the file that defines the module has not been loaded first. Make sure the file that defines the module is loaded first in your HTML before you try to use it.

Now that the module has been defined, how do we use it? We can of course add our functionality to it, and modularize our codebase into distinct sections. But more importantly, we can tell AngularJS to use these modules to bootstrap our application. The `ng-app` directive takes an *optional argument*, which is the *name of the module to load* during bootstrapping.

Let's take a look at a complete example to make sense of this:

```
<!-- File: chapter2/module-example.html -->
<html ng-app="notesApp">
<head><title>Hello AngularJS</title></head>
<body>
  Hello {{1 + 1}}nd time AngularJS

<script
  src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.19/angular.js">
</script>
<script type="text/javascript">
  angular.module('notesApp', []);
</script>
</body>
</html>
```

This example defines a module (note the empty array as the second argument), and then lets AngularJS bootstrap the module through the `ng-app` directive.

Creating Our First Controller

We saw how to create modules, but what do we do with them? So far, they have just been empty modules.

Let's now take a look at controllers. Controllers in AngularJS are our workhorse, the JavaScript functions that perform or trigger the majority of our UI-oriented work. Some of the common responsibilities of a controller in an AngularJS application include:

- Fetching the right data from the server for the current UI
- Deciding which parts of that data to show to the user
- Presentation logic, such as how to display elements, which parts of the UI to show, how to style them, etc.
- User interactions, such as what happens when a user clicks something or how a text input should be validated

An AngularJS controller is almost always directly linked to a view or HTML. We will never have a controller that is not used in the UI (that kind of business logic goes into services). It acts as the gateway between our model, which is the data that drives our application, and the view, which is what the user sees and interacts with.

So let's take a look at how we could go about creating a controller for our `notesApp` module:

```
<!-- File: chapter2/creating-controller.html -->
<html ng-app="notesApp">
<head><title>Hello AngularJS</title></head>
<body ng-controller="MainCtrl">
  Hello {{1 + 1}}nd time AngularJS
```

```
<script
  src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.19/angular.js">
</script>
<script type="text/javascript">
  angular.module('notesApp', [])
    .controller('MainCtrl', [function() {
      // Controller-specific code goes here
      console.log('MainCtrl has been created');
    }]);
</script>
</body>
</html>
```

We define a controller using the `controller` function that is exposed on an AngularJS module. The `controller` function takes the name of the controller as the first argument, which in the previous example is creatively named `MainCtrl`. The second argument is the actual controller definition, of what it does and how it does it.

But there is a slight twist here, which is the array notation. Notice that we have defined our `controller` definition function inside an array. That is, the first argument to the controller function on the module is the name of the controller (`MainCtrl`), and the second argument is an array. The array holds all the dependencies for the controller as string variables, and the last argument in the array is the actual controller function. In this case, because we have no dependencies, the function is the only argument in the array. The function then houses all the controller-specific code.

We also introduce a new directive, `ng-controller`. This is used to tell AngularJS to go instantiate an instance of the controller with the given name, and attach it to the DOM element. In this case, it would load `MainCtrl`, which would end up printing the `console.log()` statement.

Dependency Injection Syntax and AngularJS

The notation that we have used is one of the two ways in which we can declare AngularJS controllers (or services, directives, or filters). The style we have used (and will use for the remainder of the book), which is also the recommended way, is *safe-style of Dependency Injection*, or declaration. We could also use:

```
angular.module('notesApp', [])
    .controller('MainCtrl', function() {
});
```

and it would work similarly, but it might cause problems when we have a build step that minifies our code. We will delve into this more when we introduce Dependency Injection in Chapter 5.

Now, for our first AngularJS application with a controller, we are going to move the "hello world" message from the HTML to the controller, and get and display it from the controller. Let's see how this would look:

```
<!-- File: chapter2/hello-controller.html -->
<html ng-app="notesApp">
<head><title>Notes App</title></head>
<body ng-controller="MainCtrl as ctrl">
  {{ctrl.helloMsg}} AngularJS.
  <br/>
  {{ctrl.goodbyeMsg}} AngularJS

<script
  src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.19/angular.js">
</script>
<script type="text/javascript">
  angular.module('notesApp', [])
    .controller('MainCtrl', [function() {
       this.helloMsg = 'Hello ';
       var goodbyeMsg = 'Goodbye ';
  }]);
</script>
</body>
</html>
```

If we run this application, our UI should look something like Figure 2-1.

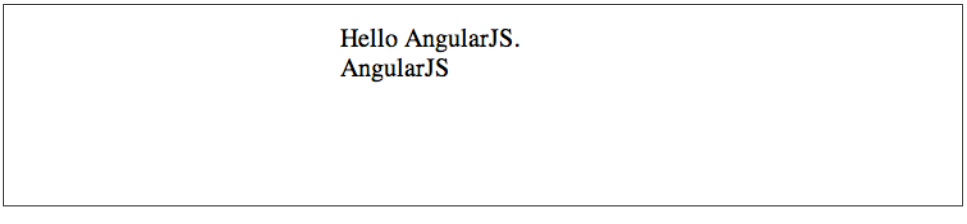

Figure 2-1. "Hello world" controller example screenshot

Yes, we only see "Hello AngularJS." The "Goodbye" message is not printed in the UI. Let's dig into the example to see if we can clarify what is happening:

- We defined our `notesApp` module as we saw before.
- We created a controller called `MainCtrl` using the controller function on the module.
- We defined the variable `helloMsg` on the controller's instance (using the `this` keyword), and the variable `goodbyeMsg` as a local inner variable in the controller's instance (using the `var` keyword).

- We used this controller in the UI through the use of another directive: `ng-controller`. This directive allows us to associate an instance of a controller with a UI element (in this case, the `body` tag).
- We also gave this particular instance of the `MainCtrl` a name when we used `ng-controller`. Here, we called it `ctrl`. This is known as the `controllerAs` syntax in AngularJS, where we can give each instance of the controller a name to recognize its usage in the HTML.
- We then referred to the `helloMsg` and `goodbyeMsg` variables from the controller in the HTML using the double-curly notation.

By now, it should be obvious that variables that were defined on the `this` keyword in the controller are accessible from the HTML, but local, inner variables are not.

Furthermore, any variable defined on the controller instance (on *`this`* in the controller, as opposed to declaring variables with the *`var`* keyword like *`goodbyeMsg`*) can be accessed and displayed to the user via the HTML. This is basically how we funnel and expose data from our controller and business logic to the UI.

Getting Data to the HTML

Changing `ctrl.goodbyeMsg` to `goodbyeMsg` in the HTML will not help either. We will not get the value of the `goodbyeMsg` variable from the controller to the UI without declaring it on the controller instance using the `this` keyword.

Anything that the user needs to see, or the HTML needs to use, needs to be defined on `this`. Anything that the HTML does not directly access should not be put on `this`, but should rather be saved as local variables in the controller's scope, similar to `goodbyeMsg`.

$scope Versus controllerAs Syntax

If you used AngularJS prior to 1.2, you might have expected the `$scope` variable to be injected into the controller, and the variables `helloMsg` and `goodbyeMsg` to be set on it. In AngularJS 1.2 and later, there is a new syntax, the `controllerAs` syntax, which allows us to define the variables on the controller instance using the `this` keyword, and refer to them through the controller from the HTML.

The advantage of this over the earlier syntax is that it makes it explicit in the HTML which variable or function is provided by which controller and which instance of the controller. So with a complicated, nested UI, you don't need to play a game of "Where's Waldo?" to find your variables in your codebase. It becomes immediately obvious because the controller instance is present in the HTML.

Let's take a look at one more example before we delve into how AngularJS works and accomplishes this:

```
<!-- File: chapter2/controller-click-message.html -->
<html ng-app="notesApp">
<head><title>Notes App</title></head>
<body ng-controller="MainCtrl as ctrl">
  {{ctrl.message}} AngularJS.

  <button ng-click="ctrl.changeMessage()">
    Change Message
  </button>
<script
  src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.19/angular.js">
</script>
<script type="text/javascript">
  angular.module('notesApp', [])
    .controller('MainCtrl', [function() {
      var self = this;
      self.message= 'Hello ';
      self.changeMessage = function() {
        self.message = 'Goodbye';
      };
  }]);
</script>
</body>
</html>
```

What has changed here?

- We now have only one binding, which is in the `ctrl.message` variable.
- There is a button with the label "Change Message." There is a built-in directive on it, `ng-click`, to which we pass a function as an argument. The `ng-click` directive evaluates any expression passed to it when the button is clicked. In this case, we tell AngularJS to trigger the controller's `changeMessage()` function.
- The `changeMessage()` function in the controller sets the value of message to "Goodbye."
- Also, as good practice, we avoid referring to the `this` keyword inside the controller, preferring to use a proxy `self` variable, which points to `this`. The following note has more information on why this is recommended.

What we will see in play is that the app starts with "Hello AngularJS," but the moment we click the button, the text changes to "Goodbye AngularJS." This is the true power of data-binding in AngularJS. Here are a few things of note from the example:

- The controller we wrote has no direct access to the view or any of the DOM elements that it needs to update. It is pure JavaScript.

- When the user clicked the button and `changeMessage` was triggered, we did not have to tell the UI to update. It happened automatically.
- The HTML connects parts of the DOM to controllers, functions, and variables, and not the other way around.

This is one of the core principles of AngularJS at play here. An AngularJS application is a data-driven app. We routinely say "The model is the truth" in an AngularJS application. What this means is that our whole aim in an AngularJS application should be to manipulate and modify the model (pure JavaScript), and let AngularJS do the heavy lifting of updating the UI accordingly.

Before we talk about how AngularJS accomplishes this, let's take a look at one more example that will help clarify things.

this in JavaScript

People used to languages like Java have trouble getting their heads around the `this` keyword in JavaScript. One of the insane and crazy (and downright cool) things about JavaScript is that the `this` keyword inside a function can be overridden by whoever calls the function. Thus, the `this` outside and inside a function can refer to two completely different objects or scopes.

Thus, it is generally better to assign the `this` reference inside a controller to a proxy variable, and always refer to the instance through this proxy (`self`, for example) to be assured that the instance we are referring to is the correct one.

If you want to read more about this, as well as understand a bit more about the craziness that can be JavaScript, do check out Kyle Simpson's *You Don't Know JS: this & Object Prototypes* (O'Reilly, 2014).

Working with and Displaying Arrays

We have seen how to create a controller, and how to get data from the controller into the HTML. But we worked with very simplistic string messages. Let's now take a look at how we would work with a collection of data; for example:

```
<!-- File: chapter2/ng-repeat-example-1.html -->
<html ng-app="notesApp">
<head><title>Notes App</title></head>
<body ng-controller="MainCtrl as ctrl">

  <div ng-repeat="note in ctrl.notes">
    <span class="label"> {{note.label}}</span>
    <span class="status" ng-bind="note.done"></span>
  </div>
```

```
<script
  src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.19/angular.js">
</script>
<script type="text/javascript">
  angular.module('notesApp', [])
    .controller('MainCtrl', [function() {
      var self = this;
      self.notes = [
        {id: 1, label: 'First Note', done: false},
        {id: 2, label: 'Second Note', done: false},
        {id: 3, label: 'Done Note', done: true},
        {id: 4, label: 'Last Note', done: false}
      ];
    }]);
  </script>
</body>
</html>
```

We introduced a few new concepts in this example. Before we delve into those, Figure 2-2 shows how the HTML and JS would look when we run it.

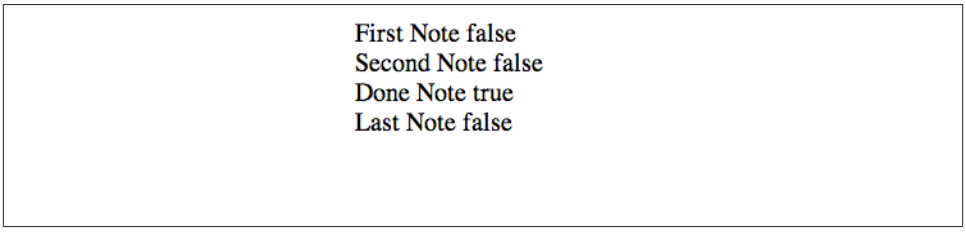

Figure 2-2. ng-repeat example screenshot

First things first, we removed the `message` variable and introduced an array of JSON objects in our `MainCtrl`. We exposed this on the controller instance with the name `notes`.

In our HTML, we used a directive called `ng-repeat`. `ng-repeat` is one of the most versatile and heavily used directives of AngularJS, because it allows us to iterate over an array or over the keys and values of an object and display them in the HTML.

When we use the `ng-repeat` directive, the contents of the element on which the directive is applied is considered the *template* of the `ng-repeat`. AngularJS picks up this template, makes a copy of it, and then applies it for each instance of the `ng-repeat`. In the previous case, the `label` and `status span` elements were repeated four times, once for each item in the `notes` array.

The `ng-repeat` is basically the same as a `for each` loop in any programming language, so the syntax is similar:

```
ng-repeat="eachVar in arrayVar"
```

We'll cover more details about `ng-repeat` in the following section.

The next point of interest is the template that we used for the `ng-repeat`. Inside the context of the `ng-repeat`, we now have a new variable, `note`, which is not present in our controller. This is created by `ng-repeat`, and each instance of the `ng-repeat` has its own version and value of `note`, obtained from each item of the array.

The final thing to note is that we used the double-curly notation to print the note's label, but used a directive called `ng-bind` for the note's `done` field. There is no functional difference between the two; both take the value from the controller and display it in the UI. Both of them also keep it data-bound and up to date, so if the value underneath changes, the UI will change automatically. We can use them interchangeably, because the expression between the double curly braces will directly drop into the `ng-bind`.

Using ng-bind Versus Double Curlies

The advantage `ng-bind` has over the double-curly notation is that it takes AngularJS time to bootstrap and execute before it can find and replace all the double curly braces from the HTML. That means, for a portion of a second while the browser starts, you might see flashing double curly braces in the UI before AngularJS has the chance to kick in and replace them. This is only for the very first page load, and not on views loaded after the first load. You will not have that issue with `ng-bind`. You can also get around that issue with the `ng-cloak` directive.

Waiting for AngularJS to Load

AngularJS has a directive called *ng-cloak*, which is a mechanism to hide sections of the page while AngularJS bootstraps and finishes loading. `ng-cloak` uses the following CSS rules, which are automatically included when you load *angular.js* or *angular.min.js*:

```
[ng\:cloak], [ng-cloak], [data-ng-cloak], [x-ng-cloak],
    .ng-cloak, .x-ng-cloak {
  display: none !important;
}
```

After this, any section or element that needs to be hidden in your HTML needs to have `class="ng-cloak"` added to it. This applies the preceding CSS and hides the element by default. After AngularJS finishes loading, it goes through your HTML and removes `ng-cloak` from all these elements, thus ensuring that your UI is shown after AngularJS has finished bootstrapping.

You can apply `ng-cloak` on the `body` tag, but it is often better to add it on smaller sections so that your application can load progressively instead of all at once.

Do note that the `ng-cloak` styling is loaded as part of the *angular.js* source code. So if you load your AngularJS library at the very end of your HTML (as you should), the style will not be applied to the HTML until AngularJS has finished loading. Thus it is often a good idea to include the preceding CSS as part of your own CSS to ensure it is loaded upfront before your HTML starts displaying.

With this example in place, let's dig in and understand how AngularJS is working behind the scenes:

1. The HTML is loaded. This triggers requests for all the scripts that are a part of it.
2. After the entire document has been loaded, AngularJS kicks in and looks for the `ng-app` directive.
3. When it finds the `ng-app` directive, it looks for and loads the module that is specified and attaches it to the element.
4. AngularJS then traverses the children DOM elements of the root element with the `ng-app` and starts looking for directives and bind statements.
5. Each time it hits an `ng-controller` or an `ng-repeat` directive, it creates what we call a *scope* in AngularJS. A scope is the context for that element. The scope dictates what each DOM element has access to in terms of functions, variables, and the like.
6. AngularJS then adds watchers and listeners on the variables that the HTML accesses, and keeps track of the current value of each of them. When that value changes, AngularJS updates the UI immediately.
7. Instead of polling or some other mechanism to check if the data has changed, AngularJS optimizes and checks for updates to the UI only on certain events, which can cause a change in the data or the model underneath. Examples of such events include XHR or server calls, page loads, and user interactions like click or type.

In our previous example with the `ng-repeat`, we have a template that shows the note's `label` and `status`. That template accesses a variable called `note`, which is created in our `for each` loop that is the `ng-repeat`. Now, if each template accessed the same context, each instance would show the same text. But to ensure that each template instance of the `ng-repeat` shows its own value, each `ng-repeat` also gets its own scope with a variable called `note` defined in it, which is specific to it.

Also note that while the `ng-repeat` instances each get their own scope, they still have access to the parent scope. If there were a function in our controller that we wanted to access within the `ng-repeat`, we could still do that.

In summation, AngularJS creates scopes or context for various elements in the DOM to ensure that there is no global state and each element accesses only what is relevant to it. These scopes have a parent-child relation by default, which allows children scopes to access functions and controllers from a parent scope.

More Directives

Let's now build on our example, and add more functionality to our ongoing application:

```
<!-- File: chapter2/more-directives.html -->
<html ng-app="notesApp">
<head>
  <title>Notes App</title>
  <style>
    .done {
      background-color: green;
    }
    .pending {
      background-color: red;
    }
  </style>
</head>

<body ng-controller="MainCtrl as ctrl">

  <div ng-repeat="note in ctrl.notes"
       ng-class="ctrl.getNoteClass(note.done)">
      <span class="label"> {{note.label}}</span>
      <span class="assignee"
            ng-show="note.assignee"
            ng-bind="note.assignee">
      </span>
  </div>

<script
  src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.19/angular.js">
</script>
<script type="text/javascript">
  angular.module('notesApp', []).controller('MainCtrl', [
    function() {
      var self = this;
      self.notes = [
        {label: 'First Note', done: false, assignee: 'Shyam'},
        {label: 'Second Note', done: false},
        {label: 'Done Note', done: true},
        {label: 'Last Note', done: false, assignee: 'Brad'}
      ];

      self.getNoteClass = function(status) {
        return {
            done: status,
```

```
                pending: !status
            };
        };
    }]);
</script>
</body>
</html>
```

We added two more directives in this example. Let's take a look at what they are and what they do:

`ng-show`
: There are two directives in AngularJS that deal with hiding and showing HTML elements: `ng-show` and `ng-hide`. They inspect a variable and, depending on the truthiness of its value, show or hide elements in the UI, respectively. In this case, we say show the `assignee` span if `note.assignee` is `true`. AngularJS treats `true`, nonempty strings, nonzero numbers, and nonnull JS objects as truthy. So in this case, we get to see the `assignee` span if the note has an assignee.

`ng-class`
: The `ng-class` directive is used to selectively apply and remove CSS classes from elements. There are multiple ways of using `ng-class`, and we will talk about what we feel is the most declarative and cleanest option. The `ng-class` directive can take strings or objects as values. If it is a string, it simply applies the CSS classes directly. If it is an object (which we are returning from the function in the controller), AngularJS takes a look at each key of the object, and depending on whether the value for that key is true or false, applies or removes the CSS class.

In this case, the CSS class `done` gets added and `pending` gets removed if `note.done` is `true`, and `done` gets removed and `pending` gets added if `note.done` is `false`.

Notice also that `ng-bind`, `ng-show`, and most of these directives can directly refer to a variable on the controller or call a function to get the value, as we did with the `ng-class`. We can pass variables and arguments to the function as normal by directly referring to the variable. No need for the double-curly syntax.

Working with ng-repeat

The `ng-repeat` directive is one of the most versatile directives in AngularJS, and can be used for a whole variety of situations and requirements. We saw how we can use it to repeat an array in the previous examples. In this section, we will explore some of the other options we have when using the `ng-repeat` directive.

ng-repeat Over an Object

Just like we used the `ng-repeat` directive to show an array of elements in the HTML, we can also use it to show all the keys and values of an object:

```
<!-- File: chapter2/ng-repeat-object.html -->
<html ng-app="notesApp">
<head><title>Notes App</title></head>
<body ng-controller="MainCtrl as ctrl">

  <div ng-repeat="(author, note) in ctrl.notes">
    <span class="label"> {{note.label}}</span>
    <span class="author" ng-bind="author"></span>
  </div>

<script
  src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.19/angular.js">
</script>
<script type="text/javascript">
  angular.module('notesApp', [])
    .controller('MainCtrl', [function() {
      var self = this;
      self.notes = {
        shyam: {
          id: 1,
          label: 'First Note',
          done: false
        },
        Misko: {
          id: 3,
          label: 'Finished Third Note',
          done: true
        },
        brad: {
          id: 2,
          label: 'Second Note',
          done: false
        }
      };
    }]);
  </script>
</body>
</html>
```

In this example, we have intentionally capitalized `Misko` while leaving `brad` and `shyam` lowercase. When we use the `ng-repeat` directive over an object instead of an array, the keys of the object will be sorted in a case-sensitive, alphabetic order. That is, uppercase first, and then sorted by alphabet. So in this case, the items would be shown in the HTML in the following order: Misko, brad, shyam.

The `ng-repeat` directive takes an argument in the form `variable in arrayExpression` or `(key, value) in objectExpression`. When used with an array, the items will be in the order in which they exist in the array.

Helper Variables in ng-repeat

The `ng-repeat` directive also exposes some variables within the context of the HTML template that gets repeated, which allows us to gain some insight into the current element:

```
<!-- File: chapter2/ng-repeat-helper-variables.html -->
<html ng-app="notesApp">
<head><title>Notes App</title></head>
<body ng-controller="MainCtrl as ctrl">

  <div ng-repeat="note in ctrl.notes">
    <div>First Element: {{$first}}</div>
    <div>Middle Element: {{$middle}}</div>
    <div>Last Element: {{$last}}</div>
    <div>Index of Element: {{$index}}</div>
    <div>At Even Position: {{$even}}</div>
    <div>At Odd Position: {{$odd}}</div>

    <span class="label"> {{note.label}}</span>
    <span class="status" ng-bind="note.done"></span>
    <br/><br/>
  </div>

<script
  src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.19/angular.js">
</script>
<script type="text/javascript">
  angular.module('notesApp', [])
    .controller('MainCtrl', [function() {
      var self = this;
      self.notes = [
        {id: 1, label: 'First Note', done: false},
        {id: 2, label: 'Second Note', done: false},
        {id: 3, label: 'Done Note', done: true},
        {id: 4, label: 'Last Note', done: false}
      ];
    }]);
</script>
</body>
</html>
```

In this example, we use the same array that we did in the example with the `ng-repeat` over the array of items. The only difference is that we now display more state about the item being repeated in the HTML. For each item, we display which index the item is in, and whether it is the first, middle, last, odd, or even item.

Each of the $ prefixed variables we use within the context of the `ng-repeat` are provided by AngularJS, and refer to the state of the repeater for that particular element. They include:

- `$first`, `$middle`, and `$last` are Boolean values that tell us whether that particular element is the first, between the first and last, or the last element in the array or object.
- `$index` gives us the index or position of the item in the array.
- `$odd` and `$even` tell us if the item is in an index that is odd or even (we could use this for conditional styling of elements, or other conditions we might have in our application).

Do note that in the case of an `ng-repeat` over an object, all of these list items exist and are still applicable, but the index of the item may or may not correspond to the order in which we declare the keys in the object. This is because of the way AngularJS `ng-repeat` sorts the keys of the object alphabetically, as we saw in the "ng-repeat Over an Object" on page 28.

Track by ID

By default, `ng-repeat` creates a new DOM element for each value in the array or object that we iterate over. But to optimize performance, it caches or reuses DOM elements if the objects are exactly the same, according to the hash of the object (calculated by AngularJS).

In some cases, we might want AngularJS to reuse the same DOM element, even if the object instance does not hash to the same value. That is, if we have objects coming from a database and we do not care about the exact object properties, we want AngularJS to treat two objects with the same ID as identical for the purpose of the repeat. For this purpose, AngularJS allows us to provide a tracking expression when specifying our `ng-repeat`:

```
<!-- File: chapter2/ng-repeat-track-by-id.html -->
<html ng-app="notesApp">
<body ng-controller="MainCtrl as ctrl">

  <button ng-click="ctrl.changeNotes()">Change Notes</button>
  <br/>
  DOM Elements change every time someone clicks
  <div ng-repeat="note in ctrl.notes1">
    {{note.$$hashKey}}
    <span class="label"> {{note.label}}</span>
    <span class="author" ng-bind="note.done"></span>
  </div>

  <br/>
```

```
  DOM Elements are reused every time someone clicks
  <div ng-repeat="note in ctrl.notes2 track by note.id">
    {{note.$$hashKey}}
    <span class="label"> {{note.label}}</span>
    <span class="author" ng-bind="note.done"></span>
  </div>

<script
  src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.19/angular.js">
</script>
<script type="text/javascript">
  angular.module('notesApp', [])
    .controller('MainCtrl', [function() {
      var self = this;
      var notes = [
        {
          id: 1,
          label: 'First Note',
          done: false,
          someRandom: 31431
        },
        {
          id: 2,
          label: 'Second Note',
          done: false
        },
        {
          id: 3,
          label: 'Finished Third Note',
          done: true
        }
      ];
      self.notes1 = angular.copy(notes);
      self.notes2 = angular.copy(notes);
      self.changeNotes = function() {
        notes = [
          {
            id: 1,
            label: 'Changed Note',
            done: false,
            someRandom: 4242
          },
          {
            id: 2,
            label: 'Second Note',
            done: false
          },
          {
            id: 3,
            label: 'Finished Third Note',
            done: true
          }
```

```
        ];
        self.notes1 = angular.copy(notes);
        self.notes2 = angular.copy(notes);
      };
    }]);
</script>
</body>

</html>
```

Here we have two arrays, `notes1` and `notes2`, which are identical in all respects. Both of them are shown in the UI using the `ng-repeat` directive. The difference is that one uses the plain vanilla `ng-repeat` (`ng-repeat="note in ctrl.notes1"`) and the other uses the track by ID version (`ng-repeat="note in ctrl.notes2 track by note.id"`).

We also included an `ng-click`, which we used before. This allows us to trigger a function in our controller whenever someone clicks that element. In this case, we call `changeNotes()` on our controller. The function changes the `notes` arrays to a new array.

Now we can see that the `hashKeys` and the DOM elements in the first `ng-repeat` are getting changed every time we click a button. In the second `ng-repeat`, there is no `$$hashKey` that AngularJS needs to generate, because we tell it what the unique identifier is for each element. So the DOM elements are reused based on the ID of the object.

Do not use any variables that start with `$$` in your application. AngularJS uses them to denote private variables that it uses for its own purposes, and does not guarantee their presence or continued working across different versions of AngularJS. If you find yourself reaching out to a $$ variable, stop! You need to rethink your approach.

We would use the `track-by` expression to optimize DOM manipulation in our application. This would generally be on the IDs of objects returned from our databases, to ensure AngularJS reuses DOM elements even if we fetch the data multiple times from the server.

ng-repeat Across Multiple HTML Elements

An uncommon requirement, but something that still pops up every now and then, is the ability to repeat multiple sibling HTML elements that may not be in a single container element. For example, think of the case where we need to repeat two table rows (`<tr>`) for each item in our array, maybe one as a header row and one as a child row.

For these kinds of situations, AngularJS provides the ability to mark where our `ng-repeat` starts and tell which HTML element is considered part of the `ng-repeat`. It does so through the use of `ng-repeat-start` and `ng-repeat-end` directives:

```
<!-- File: chapter2/ng-repeat-across-elements.html -->
<html ng-app="notesApp">
<body ng-controller="MainCtrl as ctrl">

  <table>
    <tr ng-repeat-start="note in ctrl.notes">
      <td>{{note.label}}</td>
    </tr>
    <tr ng-repeat-end>
      <td>Done: {{note.done}}</td>
    </tr>
  </table>


<script
  src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.19/angular.js">
</script>
<script type="text/javascript">
  angular.module('notesApp', [])
    .controller('MainCtrl', [function() {
      var self = this;
      self.notes = [
        {id: 1, label: 'First Note', done: false},
        {id: 2, label: 'Second Note', done: false},
        {id: 3, label: 'Finished Third Note', done: true}
      ];
    }]);
</script>
</body>

</html>
```

In this example, we are creating a table to display the list of notes. For each note, we want a row where we display the label of the note, followed by a second table row where we display the status of the note. This could very well contain the author information, when it was created, and so on. Because we can't have a wrapper element around the `tr` table row element, we can use the `ng-repeat-start` and `ng-repeat-end` directives.

We mark the first `tr` as where our `ng-repeat` starts, and use our traditional `ng-repeat` expression as the argument. We then define our template, which contains the first element with the label, and then move on to the second table row element. We mark this element as where our repeater ends by using the `ng-repeat-end` directive.

AngularJS will then ensure that it creates both `tr` elements for each element in the array we are repeating.