O'REILLY®

AngularJS Up & Running

ENHANCED PRODUCTIVITY
WITH STRUCTURED WEB APPS

Shyam Seshadri & Brad Green

O'REILLY®

AngularJS: Up and Running

If want to get started with AngularJS, either as a side project, an additional tool, or for your main work, this practical guide teaches you how to use this meta-framework step-by-step, from the basics to advanced concepts. By the end of the book, you'll understand how to develop a large, maintainable, and performant application with AngularJS.

Guided by two engineers who worked on AngularJS at Google, you'll learn the components needed to build data-driven applications, using declarative programming and the Model–view–controller pattern. You'll also learn how to conduct unit tests on each part of your application.

- Learn how to use controllers for moving data to and from views
- Understand when to use AngularJS services instead of controllers
- Communicate with the server to store, fetch, and update data asynchronously
- Know when to use AngularJS filters for converting data and values to different formats
- Implement single-page applications, using ngRoute to select views and navigation
- Dive into basic and advanced directives for creating reusable components
- Write an end-to-end test on a live version of your entire application
- Use best practices, guidelines, and tools throughout the development cycle

"I'm hardly more than an amateur JavaScript developer and I had zero problems understanding this book. I appreciate how it started at the very beginning—the *why* of AngularJS—and slowly worked its way up from there. The complimentary code repository was a *huge* help as well!"

—**Marc Amos**
frontend developer

Shyam Seshadri, owner/CEO of Fundoo Solutions in Mumbai, splits his time between working on innovative and exciting new products for the Indian markets, and consulting about and running workshops on AngularJS.

Brad Green, an engineering manager at Google, works on the AngularJS project and directs Accessibility as well as Support Engineering. Brad also worked on the early mobile web at AvantGo, and founded and sold startups.

PROGRAMMING/JAVASCRIPT

US $39.99 CAN $41.99

ISBN: 978-1-491-90194-6

9 781491 901946 53999

Twitter: @oreillymedia
facebook.com/oreilly

AngularJS: Up And Running

Shyam Seshadri and Brad Green

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo O'REILLY®

AngularJS: Up And Running

by Shyam Seshadri and Brad Green

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Simon St. Laurent and Brian MacDonald
Production Editor: Kara Ebrahim
Copyeditor: Gillian McGarvey
Proofreader: Kim Cofer
Indexer: Judy McConville
Cover Designer: Ellie Volckhausen
Interior Designer: David Futato
Illustrator: Rebecca Demarest

September 2014: First Edition

Revision History for the First Edition:

2014-09-05: First release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491901946* for release details.

ISBN: 978-1-491-90194-6

[LSI]

Table of Contents

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.

If your server has REST or API endpoints that provide JSON values, then it makes your life as a frontend developer even easier. But even if your server doesn't return JSON, that doesn't mean you should abandon AngularJS. It's pretty easy to teach AngularJS to talk with an XML server, or any other format for that matter.

Does My Entire Application Need to Be an AngularJS App?

In a word, no. AngularJS has a concept (technically, a directive, but we'll get to that in the next section) called `ng-app`. This allows you to annotate any existing HTML element with the tag (and not just the main `<html>` or `<body>` tag). This tells AngularJS that it is only allowed to work on, control, and modify that particular section of the HTML.

This makes it pretty simple to start with a small section of an existing application and then grow the part that AngularJS controls over time gradually.

A Basic AngularJS Application

Finally, with all that out of the way, let's get to some code. We'll start with the most basic of AngularJS applications, which just imports the AngularJS library and proves that AngularJS is bootstrapped and working:

```
<!-- File: chapter1/basic-angularjs-app.html -->
<!DOCTYPE html>
<html ng-app>

<body>
  <h1>Hello {{1 + 2}}</h1>

<script
  src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.19/angular.js">
</script>
</body>

</html>
```

Examples in This Book

All the examples in this book are hosted at its GitHub repository. The latest updated and correct version will always be available there in case you run into any issues when running the example from the book. Each example will also give the filename so that you can find it in the GitHub repository. Each chapter has its own folder to make it easier to find examples from the book.

There are two parts to starting an AngularJS application:

Loading the AngularJS source code
: We have included the unminified version directly from the Google Hosted Libraries, but you could also have your own local version that you serve. The Google CDN hosts all the latest versions of AngularJS that you can directly reference it from, or download it from the AngularJS website.

Bootstrapping AngularJS
: This is done through the `ng-app` directive. This is the first and most important directive that AngularJS has, which denotes the section of HTML that AngularJS controls. Putting it on the `<html>` tag tells AngularJS to control the entire HTML application. We could also put it on the `<body>` or any other element on the page. Any element that is a child of that will be handled with AngularJS and be annotated with directives, and anything outside would not be processed.

Finally, we have our first taste of AngularJS one-way data-binding. We have put the expressions "1+2" within double curly braces. The double curly is an AngularJS syntax to denote either one-way data-binding or AngularJS expressions. If it refers to a variable, it keeps the UI up to date with changes in the value. If it is an expression, AngularJS evaluates it and keeps the UI up to date if the value of the expression changes.

If for any reason AngularJS had not bootstrapped correctly, we would have seen `{{1 + 2}}` in the UI, but if there are no errors, we should see Figure 1-1 in our browser.

Figure 1-1. Screenshot of a basic AngularJS application

AngularJS Hello World

Now that we've seen how to create an AngularJS application, let's build the traditional "hello world" application. For this, we will have an input field that allows users to type in their name. Then, as the user types, we will update the UI with the latest value from the text box. Sound complicated? Let's see how it would look:

```
<!-- File: chapter1/angularjs-hello-world.html -->
<!DOCTYPE html>
<html>

<body ng-app>
  <input type="text"
         ng-model="name"
         placeholder="Enter your name">
  <h1>Hello <span ng-bind="name"></span></h1>
```

```
<script
  src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.19/angular.js">
</script>
</body>

</html>
```

We have added two new things from the last example, and kept two things:

- The AngularJS source code is still the same. The `ng-app` directive has moved to the `<body>` tag.
- We have an `input` tag, with a directive called `ng-model` on it. The `ng-model` directive is used with input fields whenever we want the user to enter any data and get access to the value in JavaScript. Here, we tell AngularJS to store the value that the user types into this field in a variable called `name`.
- We also have another directive called `ng-bind`. `ng-bind` and the double-curly notation are interchangeable, so instead of `<span ng-bind="name"></span>`, we could have written `{{ name }}`. Both accomplish the same thing, which is putting the value of the variable `name` inside the tag, and keeping it up to date if it changes.

The end result is captured in Figure 1-2.

Figure 1-2. AngularJS "hello world" example screenshot

Conclusion

We wrote two very simple AngularJS applications. The first demonstrated how to create a very simple AngularJS application, and the second showcased the power of AngularJS two-way data-binding. The best part was that we were able to do that without writing a single line of JavaScript. The same application in pure JavaScript would require us to create listeners and jQuery DOM manipulators. We were able to do away with all of that.

We also went over the basic philosophies of AngularJS and some of its benefits and how it differs from existing solutions. In the next chapter, we become familiar with some of the most common pieces of AngularJS, such as common directives, working with controllers, and using services.