AngularJS IN DEPTH

David L. Aden
Jason L. Aden

MEAP

MANNING

MEAP Edition
Manning Early Access Program
AngularJS in Depth
Version 5

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

welcome

Thank you for purchasing the MEAP for *AngularJS in Depth*.

Working on this book has been a fascinating experience and, thanks to your interest and participation, we're sure things will get even more interesting. We're trying a few new things in this book in terms of how we describe Angular and will rely on you to tell us whether they work or whether we need to go back to the drawing board.

AngularJS in Depth is not intended to be your first book on Angular. For that, we suggest Manning's *AngularJS in Action*. What this book does intend to do is to help you move from familiarity with Angular to a higher level of Angular expertise.

It is easy to see how helpful Angular can be just by going through the introductory tutorial and it is relatively straightforward to use Angular to build simple single page applications. But, it has long been a challenge to make the move from basic familiarity to deep understanding. In part that is because Angular does so many things and employs advanced patterns such as dependency injection and a unique way to separate the model, view and controller.

But it is our belief that a large part of the difficulty has been Angular's documentation. Too often material that describes one feature of Angular uses other Angular terms, making it hard to break the infinite loop. Additionally, some parts of Angular, such as $scope, is used and demonstrated but the standard documentation doesn't examine it in detail so we never really understand it in depth or fully see how it relates to dependency injection or two-way binding.

The plan for this book is to begin by describing all the key parts of Angular and their relationships in a way that doesn't require you to already be an Angular expert. However, we will assume you have encountered common Angular terms and have some idea of what Angular does.

Chapters 1 and 2, which are being released now, are the on-ramp for the rest of the book. Chapter 1 provides background and details of the approach we will take. Chapter 2 covers those items that are prerequisites to understanding Angular. The rest of the book assumes you understand the concepts it presents.

Chapters 3 and 4 are where we step back to see the big picture of Angular's parts and their relationships and then dive into an explanation of how they work and what they are used for. The intention is to give a behind-the-curtain look at what makes Angular tick.

From there, we dive into advanced topics on directives, controllers, filters and animations.

Becoming a high-level Angular professional also means you need to know how to set up an Angular application, manage your code, coordinate team development, manage unit and end-to-end testing and successfully deploy code to testing and production environments. So, we'll take up those topics as well and cover the tools that exist to automate or smooth those processes.

Finally, since Angular 2.x will likely be released while this book is in progress, we will also take it up.

We hope you find this book useful and we are looking forward to receiving your feedback and using your suggestions to grow this book into something that helps you master Angular.

—Jason and David Aden

brief contents

1

The Path to Angular Professional

Angular burst onto the broader Javascript framework scene in 2013 bringing with it new concepts, a radical approach to building Single Page Applications (SPAs) and a couple of flavors of secret sauce. Even just its introductory demonstrations of two-way binding, directives and dependency injection were things most of us hadn't seen before. Merely showing two-way binding to other developers or customers could generate "wow" moments that could made developer converts or help to seal a customer deal.

If you're reading this book or considering buying it, you probably already know the above. So what does this book intend to accomplish?

Quite simply, its purpose is to help transition you from being a developer familiar with Angular to an Angular professional. This purpose could, of course, apply to developers who already are expert with parts of Angular but want to extend that expertise. However, it is not the book to pick up to learn Angular – that need is filled with "Angular in Action." This is the book to pick up when it's time to become more proficient with Angular and when you want to gain a deeper understanding of what lies behind the curtain, what it can accomplish and how to make it dance.

We begin by laying the groundwork for the rest of the book. We identify the role Angular plays, the areas of expertise required to be an Angular professional, the problems we choose when choose Angular and the approach that guides the rest of the book.

1.1 What is Angular?

Your interest in this book pretty much guarantees that you have an understanding of what Angular is. But if you need a good introduction to it and thereby get a full definition of it, the Manning book "Angular in Action" by Brian Ford and ?? does a great job of introducing Angular and all its key features. We're not going to attempt to boil that book into this small section but

it would be an oversight not to include at the beginning of this book a working definition for Angular.

A common way that has been used to define Angular is by comparison to other products. Many of those comparisons are available online by searching for Angular and whatever other product with which you are familiar. However, the value of such comparisons is largely limited to those people who have expertise in the other products. What we're looking for is a definition of Angular itself that doesn't depend on detailed knowledge of other products. Interestingly there has been some debate in the Angular community as to even the category or product into which Angular fits though the somewhat begrudging consensus seems to be that it fits best into the "framework" category.

In general usage, a framework is "an essential supporting structure of a building, vehicle, or object" or a "basic structure underlying a system, concept, or text."[1] Those definitions seem to apply quite well to Angular. Based on those definitions and the descriptions provided by the Angular team, our working definition for Angular is: it is an opinionated (it expects things to be set up in a specific way) application framework written in Javascript that runs in-browser with no additional plug-ins. It is used to build desktop and/or mobile applications and includes generalized services which:

- Interact with server-side data repositories
- Manage the client-side representation of data
- Implement interactive user interfaces, including the ability to extend HTML
- Allow the developer to extend the vocabulary of Angular itself by providing ways to define, configure, manage and access reusable code
- Support unit tests that don't require special "hooks" or alterations of the application code.

It delivers a structure on which applications -- their business logic, data and views (presentations) – are organized and built and has as a primary motivating intention the idea it should handle the basic application needs well enough that developers are freed up to focus on building applications rather than worrying about low level details. It is not intended to generate an application automatically but is intended to streamline every-day tasks encountered when building rich, interactive, browser-based web applications.

Now that we've identified what Angular is, let's take a look at what it takes to become an Angular professional.

[1] Definitions returned by Google search "define framework" on 6/15/14

1.2 On Becoming a Professional

Mastering Angular requires more than the ability to write some Angular code. It requires you understand everything involved in delivering a high performance, maintainable Angular application that plays nicely with the other parts of the application, the rest of the enterprise and the infrastructure on which it runs. To accomplish this you need to understand the problems Angular solves – those things it does easily and well – and, possibly even more important, the problems it creates.

Knowing the types of problems Angular creates and how to minimize, work around or overcome them is an integral part of practicing Angular at a professional level. We'll begin by identifying the broad categories of expertise needed to be an Angular professional.

1.2.1 Milestones on the Path

As mentioned, being a complete Angular professional requires more than just knowing some Angular. Of course, you must understand Angular conceptually and in detail, but the day-to-day work of an Angular professional involves skills that extend beyond Angular's borders. This book assumes four areas or zones of operation required of the Angular professional:

1. Knowing Angular itself. This encompasses understanding Angular's parts, how they relate and how to use them to build powerful applications.
2. Setting up and configuring Angular. How you organize and configure your Angular applications is important to make development easier, reduce maintenance headaches and accommodate future application growth. An application architecture that works for a small application may be a nightmare for an enterprise-class application.
3. The larger context. Knowing how to fit Angular into the larger application context, including whatever may be used for the backend data repository, helps write better Angular code.
4. Deploying your application. Establishing and managing the processes you use to coordinate front- and back-end developers, properly caring for the code, and correctly defining deploy policies and procedures impacts how well your application works and is received. The best application in the world will garner the harshest critics if it is not properly tested and deployed.

<<Diagram here showing Angular as a unit with key parts (1), organized into a file structure (2), interacting with other parts of the application (3) all managed by a set of processes and tools. Mmmm…will have to think about we might show that. ☺ >>

The above doesn't imply you need to be an expert at maintaining infrastructure or running code deploy tools or managing the data repository just to build Angular applications (though having that knowledge won't hurt). But it does mean that part of building professional Angular applications requires you to know something about the rest of the application, how your part interacts with it and the problems you might encounter along the way. You will likely be called on to help make decisions about deployment processes or infrastructure needs and will need

to be able to address those questions competently to ensure your application performs as advertised.

Most of this book will be taken up with the first two items – knowing and configuring Angular -- because those two are almost exclusively under your control. But, the other two items can't be ignored if you want to write Angular applications at a professional level. Let's expand quickly on each of the above to see where that rubber meets real world road.

KNOWING ANGULAR

It's probably obvious even trite to say you need to know Angular to work with it professionally. But we need to know what differentiates the beginning developer from the advanced developer. In our experience, the difference is that that advanced developer has a comfortable understanding of Angular's parts and how they fit together. They know both the big picture and the details. Usually, they have arrived at the "big picture" by going through the details. It's natural that as we learn more about Angular's parts and their relationships, new possibilities open that make our code more efficient, simpler or just plain cooler.

For example, suppose you are asked to build a complex directive. You find an open source directive that does most of what you need but lacks a few small features or needs to support a couple of additional events. The obvious answer is to customize the third-party directive so it implements all the features you need. A newer Angular developer would likely copy the directive's code to start, adding in whatever is needed to fulfill her specific needs. This leaves the problem that when the third-party library updates you'll have to examine it to see if and what you should migrated to your version.

With a deeper understanding of directives and the right situation, there's another approach: create a new directive that invokes the third party directive and grabs the object it returns. This gives you access to the directive's properties including the 'compile' method, which opens the door to access the directive's features and functionality. So, you get the original return from the compile method, add in the events you need to support and return the monkey-patched version of the directive. By creating your directive to wrap the original one, you've accessed the original's features without directly touching its code – a nice win for reusability that also should make updating less painful.

SETTING UP AND CONFIGURING ANGULAR

Some of the earliest available online Angular examples served the purpose of demonstrating a feature but, perhaps in the interest of keeping things simple, unfortunately also demonstrated bad practices for organizing code. One of the pluses for Angular is that it makes it relatively easy to move code around as an application grows but why not set things up the right way to begin with? You never know when the scope and organizational interest of an application might change. You might as well set up your applications to accommodate future growth, as long as it doesn't make life burdensome when you're first starting out.

For argument's sake, let's say you begin a side Angular project designed to streamline a key workflow within your department. You only work on it part time but several people start using it and their work speeds up noticeably. Happily, your boss recognizes it as a great

addition to the department and then decides it might make it possible to meet the unrealistically high annual targets. Suddenly, your part time application goes from part to full time. This is a good thing, it means your work is valuable and besides the side project was always more fun than the maintenance work you normally have to trudge through. Your boss assigns an additional developer, a part-time designer to address you developer-ugly interface and a person on the API team to take care of the backend so the data can move from a MySQL instance running on your desktop to a departmental database.

It's all very exciting but the first thing that hits you in the face is all the refactoring and code reorganization you didn't take the time to do when it was just a side project. Well, the project just became serious and the boatload of "todo's" littering your code provides absolutely no comfort. You'd prefer no one would ever see them.

Correctly setting up and configuring your Angular application at its inception helps prevent or at least lessen these types of problems. Of course no matter the planning, you always know more about an application after you've worked on it so the initial set-up may need to be adjusted. But, starting with an intelligent initial configuration does lessen future problems.

THE LARGER APPLICATION CONTEXT

The original vision for Angular included a backend repository and tools to interact with it but the team decided to focus on building just the front-end technology. That means someone else and some other set of code will likely managing the data, security, and external interactions on which your application depends. Almost any application, including the smallest, have dependencies or interactions with other things. Planning for and writing code with this larger context in mind is a vital element for building a professional Angular application.

Not that this has ever happened to us, but let's assume you're responsible for the Angular front-end of an enterprise application for which the API is also being developed. A decision is made, because the API will take a while, to create a fake API that is will mirror the final one both in terms of how you call it and the data it returns. You build your part, requesting information as you need it from the fake API. Everything moves forward happily.

Until the real API starts to arrive and things don't quite line up. The reality of the backend data, disconnects between the real and fake API's, and a failure to fully define the interactions beforehand lead to way more complications than there should be. Your estimates of the time it would take to migrate from the fake to the real API turn into oft-repeated in-house joke.

The problem here obviously isn't really with your Angular application, but with a mis-estimation of the context in which it will live. Understanding and working within that context is part of being a professional Angular developer.

PROCESSES FOR MANAGING ANGULAR APPLICATIONS

Building an application that runs nicely on your workstation and building an application that travels smoothly from development to testing to production are two very different activities. There are even significant differences between how you work on and deploy a brand new application and doing the same actions for an application once it is live and in use by the masses. The processes and tools for deploying your application to all the environments

accessed by downstream stakeholders and users are an important part of working on any application. Let's take a look at what could – and for us has – happened when this expertise is overlooked.

You're on a deadline. Your application has to be finished soon and there's marketing dollars already teed up for a couple of days after the target release. You're coding like mad and making good progress. To your surprise, it looks like you'll actually make the date.

Of course you haven't deployed to production yet because it's still being set up, but not to worry because it's similar to the test environment, just a bit bigger. The good news is the app performs well and load times aren't bad – and you're sure they'll only improve when the code is minified and concatenated.

A couple of days before release you set up the grunt script to minify and you deploy to the production servers and the application promptly breaks. You dive into debugging and discover that some of your Angular code has dependencies that are not minification-safe. Three hours of frantic debugging later and the app is working, at least sort of. You turn on concatenation to lower the number of required downloads and again, the application promptly breaks.

Finally, at the end of a long evening the app works, sort of, though it's disconcerting that the unit tests won't pass against the minified/concatenated version.

You deploy, the site goes live and a few routine bugs are reported. You deploy a hotfix and production breaks. Do you revert, deploy un-minified/un-concatenated code or try to debug?

The point is simple: how you wrap up and deliver your Angular application so it arrives nicely packaged with a pretty bow on top is itself an expertise. How it is going to be packaged even impacts how you should write your Angular code. So, this area of expertise cannot be ignored.

Okay, the above covers the broad categories of knowledge required of an Angular professional. Now let's look at something that we as developers face and will face every day: problems.

1.2.2 Problems Angular Solves/Problems it Makes

Angular solves many problems encountered when writing single page applications. But it doesn't do away with problems – if it did, there would be no need for this book, or, really for our jobs. Every technology choice comes with its own set of problems so it is a good idea to know what problems you are selecting when you select a technology. You should pick a technology that give you problems you don't mind having or, even better, that you enjoy solving. The first section below quickly covers the problems Angular solves. These tend to be well-known because they are usually the reason we picked (or someone paying our salary) picked Angular in the first place.

Equally important, though, is identifying the problems that Angular gives us since we'll need to deal with these on a day-to-day basis. The second section below highlights the problems we choose when we choose Angular and, of course, much of the rest of the book will deal with these problems in detail.

We do not spend any time in the following defining the Angular terms, which rubs us the wrong way as we don't like using technical terminology without defining the terms. As we cover in the final section of this chapter, we believe that using terms that haven't yet been defined is one of the problems that has made it difficult for developers to learn Angular as expertly as they would have liked. Hopefully, you'll recognize most of the terms in the following and rest assured these terms will be thoroughly defined in Chapter 3 and beyond.

"SOLVES"

The most familiar list of the problems Angular solves is right on the Angular home page which lists Angular's key features. In brief, the problems Angular solves include:

TWO-WAY BINDING

This is one of the first things demonstrated to show how cool Angular is and it does solve a real-world problem intrinsic to large Javascript applications: keeping track of all the events. Modern web applications respond to many user and server events. If you manually create all the code to listen for and handle those events, the real task of making the application meet the business requirements can get lost in the details. Angular takes care of creating and managing most of the events you would otherwise have to code by hand.

DEPENDENCY INJECTION

Many positive things can be said about dependency injection and we'll talk about it at length in this book, but the bottom line is it gives you a convenient way to create once and register objects and services that can be re-used throughout the application. Angular handles all the administrative tasks involved in creating, registering and delivering the re-usable code when and how you need it. Without DI, code reuse is a nice idea but the mechanics of making it work can involve too much manual tracking and administration.

DIRECTIVES

Often cited as one of the key features of Angular, directives solve the "problem" of HTML. HTML has limitations and additions or changes to it take time to make their way through the standards process, not to mention the fact that the rate of implementation varies browser to browser (Internet Explorer we are looking at you). Angular solves this problem by giving us the capability to extend HTML right now when we need to.

MANAGING SERVER-SIDE AND CLIENT-SIDE DATA

Any web application worth the name deals with data and usually has to manage data on both the server and client sides. Keeping the data in sync and building client-side models that correctly reflect server-side data can be time consuming. Angular reduces the amount of manual work required to build and manage those data models.

TESTING

Any professional application has associated unit tests. If the framework you use isn't built to facilitate unit testing, you quickly get into problems. Angular was built from the beginning assuming you would write unit tests and so supports that process.

"MAKES"

As mentioned above, the list of problems you choose when selecting Angular is at least as important as the problems it solves. If you've spent any time working on an Angular application, one or more of the following will be familiar to you, beginning with the "makes problem" side of the features listed above.

TWO-WAY BINDING

Two-way binding is great up until the moment there's too much of it. Complex pages with lots of internal dependencies or large pages that display a ton of data can lead to situations in which two-way binding becomes a performance problem. Paging data, schemes to bind some values only once, rendering pages as display-only except for small sections that selectively become interactive only when needed, capping the number of bindings on a page, etc. are all approaches to handling the problem of "too much" two-way binding. New browser capabilities promise to ameliorate this problem further in future versions of Angular but there will probably never come a time when you don't have to be smart about how you use two-way binding.

DEPENDENCY INJECTION

Dependency Injection is great but it sometimes suffers from not being enough. Because Angular only creates one injector (the service that facilitates code reuse), the names of the registered objects have to be unique. This can be a problem for large or multi-part applications. Sometimes it would be helpful to have specialized services available on a sub-application basis without worrying about name collisions. Angular also only allows you to instantiate modules once – at application start-up. It might be useful in large applications or highly dynamic applications to instantiate services at runtime based on user needs.

DIRECTIVES

There seems to be an agreement that directives are difficult to understand and configure. So much so that Angular 2.0 has as one of its design goals simplification of directive configuration.

Directives can also cause problems in terms of understanding how an application is put together. One of their great advantages is they can hide complexity but when it comes to debugging or working with an application that someone else wrote, tracing down which directives are doing what and how they're interacting can be like tracing strands in a bowl of spaghetti – it could get messy. For example, supposed you are new to an Angular application and have been tasked with tracing down an error on a complex page. You open the code and find that no naming convention is used to distinguish home-grown directives. You stare at a block of code wondering whether a comment, a class, an attribute or an element might be a home-grown directive. And, once you find it, it's not unusual to find additional directives used inside the first one.

You feel like you're tracing a bunch of GOTO statements, a sure sign that debugging is not going to be fun. Throw in some third-party library directives such as UI-Bootstrap which add their own behavior, often somewhat out of sight of the code you're focusing on, and things can

become seriously complicated. What is needed is a good directive mapper – a way to analyze or visualize the directives and sub-directives at play on a page or on a section of a page.

DATA MODELS

Angular's mechanisms for marshaling data from the server to the client and keeping both sides in sync is useful, but doesn't always do the job. The built-in service ($resource) often comes up short and is either replaced or teams write their own low-level code to manage their unique data.

Although all Javascript applications are subject to memory leaks, Angular memory leaks can be particularly challenging to trace down. Because of the way Angular applications are organized and code is modularized, and the need exists to interact with third-party libraries, it is easy to create references to memory structures that persist long after you assume they will. Some tools exist to help solve this, but on one recent project, one of the senior developers spent a week "living in the Chrome profiler" trying to trace down the worst of the application's memory leaks and wasn't able to find them all.

INTEGRATION WITH THIRD-PARTY LIBRARIES

The borderline between Angular and third-party applications sometimes causes problems, if only because it can be confusing to know which code is causing which effect. We have also seen applications that suffered from "library overload." It's easy to understand how that can happen. You're working on an application and need a control. AngularUI provides it so you add it to the mix. A bit later you need something else and add in Select2. Still later you find other nice controls and add them in. Six months down the road you have five different libraries which not only bloated the initial download, but sometimes interact in odd ways.

VALIDATION

The Angular form validation is very cleverly written but has some limitations. For example, if you have two controls on a page that have validation dependencies between them – such as an action taken on control A should make control B required – built-in validation does not handle the situation well. We have also seen complex requirements for error messages that that walk the user through a set of errors spread across the page. Such systems required a tremendous amount of additional coding on top of validation to make them work.

AND MORE???

NOTE: When we get to MEAP, might want to pose the question to the readers to get a list of additional common problems they encounter with Angular.

1.3 Our Rationale (or Our Approach)

This section could arguably appear in the very beginning of this book, in the "front matter", prior to Chapter One. But we wanted to place it here because we feel it is an important and needed disclosure which could get lost in the pre-content descriptions of why this book was written, who it was written for and the coding conventions it uses. Those items are important, but our concern is they are not read in depth by a significant percentage of a technical book's

audience (not, of course, that we have ever skipped reading those parts of other books). We approached writing this book with a particular emphasis and felt it only fair to highlight that emphasis early on.

This book is intended to fill a perceived need in the Angular community: how to make the leap from being somewhat familiar with Angular to working with it at a professional level. Part of making that jump successfully encompasses the items outside of Angular discussed in section 1.2.1 Milestones on the Path.

But a lot is specific to Angular because it is an opinionated framework. Angular is not just a set of generalized tools that somehow or another can be cobbled together into an application. It serves best when you think about things in specific ways and favors certain approaches to solving development problems. One of the things we admired most about Angular when getting started is how tightly integrated it is. It consists of a fantastic set of functions, features and conventions that together deliver something bigger and better than its individual pieces.

I'm sure we're not the only developers who, as we continue to learn more about it, are impressed with the clever way it was architected, how needs seem to be anticipated and solved. For example, when you begin working with Angular expressions it's obvious that it would be helpful to have a way to reliably and "angularingly" parse those expressions. And lo and behold, there is it – the $parse service which takes any legal Angular expression and turns it into a function to retrieve its value or carry out an action. Angular takes the generalization solution a step further. Not only does it give you a function to retrieve the expression's value within a given context, when feasible it also provides a way to assign a value to the expression.

Angular is filled with those kind of clever solutions to general and specific needs.

This is one of Angular's strengths but we believe it is also one of the things that makes it sometimes difficult to learn. It is so well integrated conceptually that it has been difficult to define one part without referring to another as yet undefined part. There are few things more frustrating than specialized definitions that depend on other specialized definitions that refer back to the one you started with – or to five other as-yet undefined definitions.

Integrated, not tightly coupled (sidebar)

We should note that despite the tight conceptual and actual integration of Angular it is not true that the Angular team is trying to build a tightly coupled monolith. To the contrary, they are working to break Angular into replaceable pieces so teams can swap out parts of it to better serve their application's needs. Understanding the integration will still be important, perhaps even more so, when we have the option to swap out maor pieces since we'll need to understand what it is we're giving up to get something else. (Remember all technology choses come with their own set of problems.)

This book has as its goal to help you move from general understanding of Angular to a deep understanding of its twigs and berries, Kibbles and Bits or whatever other phrase you want to use to characterize its inner workings. As part of that we hope to present specific data on how to solve coding problems, how to organize your code, etc. That is important and likely one of the reasons you are considering or have bought this book. But we don't believe it is the most important thing this book can accomplish.

The most important contribution we'd like to make is based on our belief about the great things you can and will do with Angular. Knowledge is only valuable when possessed and used by people and in this case, by developers like you. So we believe that any effort that helps you to better understand the basic parts of Angular and how they relate one to another is an investment in the people who will do great work and is the most powerful way to advance its use.

Sometimes people brush off basic definitions as unimportant or something "everyone knows" but it is often the basic definitions that drift along, not fully understood or wrongly understood, that cause the most intractable difficulties. It is those "little" or basic misunderstandings that make the larger subject complicated or obtuse, or at least seem that way.

As mentioned in the "Why this Book", the original Angular documentation was something of an Everest to climb. But probably its most basic fault was that it left basic terms undefined or even obscured their meaning. Failing to understand the basic terms and their relationships obscured the big picture.

No clearer example of this is needed than the continuing confusion surrounding the term "provider." Not only was it not clearly and comprehensively defined such that you didn't have to already know the rest of Angular, it was unfortunately (as the Angular team itself has noted) badly overloaded. So, as you probably know, one type of "provider" is a "provider", and the second "provider" means something different from the first one – the first is a general term referring to a broad class of things while the second is a specific type of that broad class. Many others have contributed to clarify what it means and the general community understanding today is much higher than when Angular was first released. But, of course, there are always new people picking up Angular who in their turn face the problem of working out exactly what a "provider" is and which of the different definitions is being used when the word appears in other definitions such "factory."

While we would like to state emphatically and confidently that the definitions we're going to provide (I can't believe I used that word here!) in this book are the best, last and final word when it comes to Angular definitions, I'm sure that won't be the case.

But what is important to understand as a reader is that our attempt is going to be to give you basic definitions and that we think definitions are important. Good definitions not only tell you what a thing is, but where something fits in to the universe in which it lives and, perhaps more important, what the item being defined is used for and what effects it creates or doesn't.

This is where you come in. While we will present best practices in this book, we believe that its biggest impact will not be measured by the usefulness of the recipes or techniques it

contains but by the degree to which we succeed in giving you basic, workable definitions that help advance your understanding of Angular because that will, in turn, advance your use of it to that elusive “next level.”

And with that we wrap up the introductory chapter. Next, we’ll dive into the things you need to know to get the most out of the rest of the book. Chapter 2 covers those things that are the backdrop to understanding Angular. You may already know many of the items covered. If so, skip those sections or if all of it is old hat, feel free to skip the entire chapter. Chapter 2 covers several topics that entire chapters or, in some cases, entire books have been written about but we feel it is important to lay out a high level definition of them to ensure we’re all starting from the same place speaking the same language.