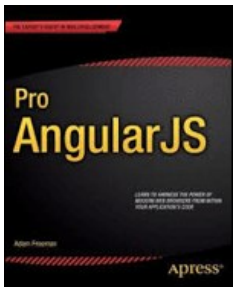

Pro AngularJS

by Adam Freeman

Chapter 3: Putting AngularJS in Context

Overview

In this chapter, I put AngularJS in context within the world of web app development and set the foundation for the chapters that follow. The goal of AngularJS is to bring the tools and capabilities that have been available only for server-side development to the web client and, in doing so, make it easier to develop, test, and maintain rich and complex web applications.

AngularJS works by allowing you to *extend* HTML, which can seem like an odd idea until you get used to it. AngularJS applications express functionality through custom elements, attributes, classes, and comments; a complex application can produce an HTML document that contains a mix of standard and custom markup.

The style of development that AngularJS supports is derived through the use of the *Model-View-Controller* (MVC) pattern, although this is sometimes referred to as Model-View- *Whatever*, since there are countless variations on this pattern that can be adhered to when using AngularJS. I am going to focus on the standard MVC pattern in this book since it is the most established and widely used. In the sections that follow, I explain the characteristics of projects where AngularJS can deliver significant benefit (and those where better alternatives exist), describe the MVC pattern, and describe some common pitfalls.

Understanding Where AngularJS Excels

AngularJS isn't the solution to every problem, and it important to know when you should use AngularJS and when you should seek an alternative. AngularJS delivers the kind of functionality that used to be available only to server-side developers, but entirely in the browser. This means that AngularJS has a lot of work to do each time an HTML document to which AngularJS has been applied is loaded—the HTML elements have to be compiled, the data bindings have to be evaluated, directives need to be executed, and so on, building support for the features I described in Chapter 2 and those that are yet to come.

This kind of work takes time to perform, and the amount of time depends on the complexity of the HTML document, on the associated JavaScript code, and—critically—on quality of the browser and the processing capability of the device. You won't notice any delay when using the latest browsers on a capable desktop machine, but old browsers on underpowered smartphones can really slow down the initial setup of an AngularJS app.

The goal, therefore, is to perform this setup as infrequently as possible and deliver as much of the app as possible to the user when it is performed. This means giving careful thought to the kind of web application you build. In broad terms, there are two kinds of web application: *round-trip* and *single-page*.

Understanding Round-Trip and Single-Page Applications

For a long time, web apps were developed to follow a *round-trip* model. The browser requests an initial HTML document from the server. User interactions—such as clicking a link or submitting a form—led the browser to request and receive a completely new HTML document. In this kind of application, the browser is essentially a rending engine for HTML content, and all of the application logic and data resides on the server. The browser makes a series of stateless HTTP requests that the server handles by generating HTML documents dynamically.

A lot of current web development is still for round-trip applications, not least because they require little from the browser, which ensures the widest possible client support. But there are some serious drawbacks to round-trip applications: They make the user wait while the next HTML document is requested and loaded, they require a large server-side infrastructure to process all of the requests and manage all of the application state, and they require a lot of bandwidth because each HTML document has to be self-contained (leading to a lot of the same content being included in each response from the server).

Single-page applications take a different approach. An initial HTML document is sent to the browser, but user interactions lead to Ajax requests for small fragments of HTML or data inserted into the existing set of elements being displayed to the user. The initial HTML document is never reloaded or replaced, and the user can continue to interact with the existing HTML while the Ajax requests are being performed asynchronously, even if that just means seeing a "data loading" message.

Most current apps fall somewhere between the extremes, tending to use the basic round-trip model enhanced with

JavaScript to reduce the number of complete page changes, although the emphasis is often on reducing the number of form errors by performing client-side validation.

AngularJS gives the greatest return from its initial workload as an application gets closer to the single-page model. That's not to say that you can't use AngularJS with round-trip applications—you can, of course—but there are other technologies that are simpler and better suit discrete HTML pages, such as jQuery. In Figure 3-1 you can see the spectrum of web application types and where AngularJS delivers benefit.

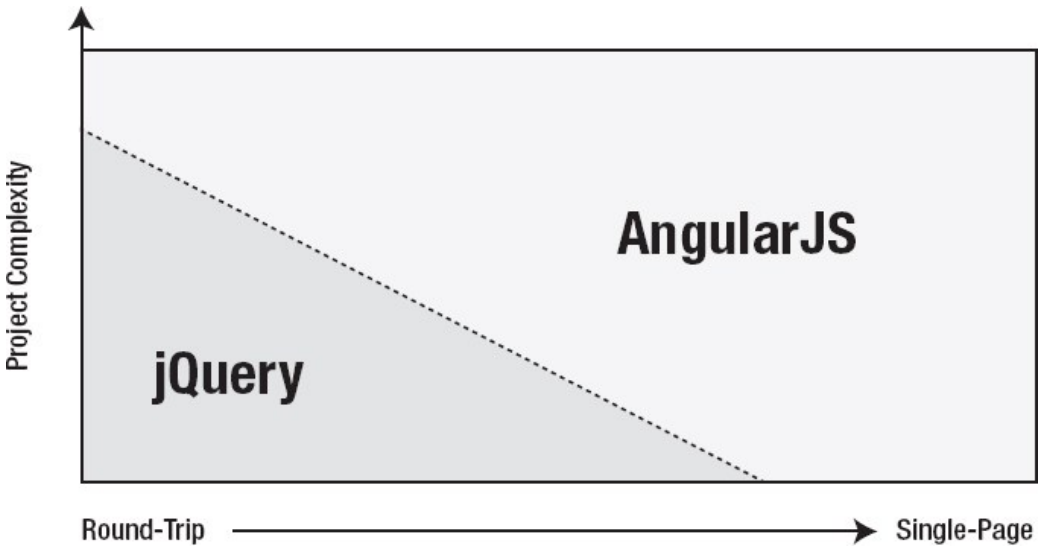

Figure 3-1: AngularJS is well-suited to single-page web apps

AngularJS excels in single-page applications and especially in complex round-trip applications. For simpler projects, jQuery or a similar alternative is generally a better choice, although nothing prevents you from using AngularJS in all of your projects.

There is a gradual tendency for current web app projects to move toward the single-page application model, and that's the sweet spot for AngularJS, not just because of the initialization process but because the benefits of using the MVC pattern (which I describe later in this chapter) really start to manifest themselves in larger and more complex projects, which are the ones pushing toward the single-page model.

Tip This may seem like circular reasoning, but AngularJS and similar frameworks have arisen because complex web applications are difficult to write and maintain. The problems these projects face led to the development of industrial-strength tools like AngularJS, which then enable the next generation of complex projects. Think of it less as circular reasoning and more of a virtuous circle.

AngularJS and Jquery

AngularJS and jQuery take different approaches to web app development. jQuery is all about explicitly manipulating the browser's Document Object Model (DOM) to create an application. The approach that AngularJS takes is to coopt the browser into being the foundation for application development.

jQuery is, without any doubt, a powerful tool—and one I love to use. jQuery is robust and reliable, and you can get results pretty much immediately. I especially like the fluid API and the ease with which you can extend the core jQuery library. If you want more information about jQuery, then see my book *Pro jQuery 2.0*, which is published by Apress and provides detailed coverage of jQuery, jQuery UI, and jQuery Mobile.

But as much as I love jQuery, it isn't the right tool for every job any more than AngularJS is. It can be hard to write and manage large applications using jQuery, and thorough unit testing can be a challenge.

One of the reasons I like working with AngularJS is that it builds on the core functionality of jQuery. In fact, AngularJS contains a cut-down version of jQuery called *jqLite*, which is used when writing custom directives (and which I describe in Chapters 15-17). And, if you add the jQuery to an HTML document, AngularJS will detect it automatically and use jQuery in preference to jqLite, although this is something you rarely need to do.

The main drawback of AngularJS is that there is an up-front investment in development time before you start to see results—something that is common in any MVC-based development. This initial investment is worthwhile, however, for complex apps or those that are likely to require significant revision and maintenance.

So, in short, use jQuery for low-complexity web apps where unit testing isn't critical and you require immediate results. jQuery is also ideal for enhancing the HTML that is generated by round-trip web apps (where user interactions cause a new HTML document to be loaded) because you can easily apply jQuery without needing to modify the HTML content generated by the server. Use AngularJS for more complex single-page web apps, when you have time for careful design and planning and when you can easily control the HTML generated by the server.

Understanding the MVC Pattern

The term *Model-View-Controller* has been in use since the late 1970s and arose from the Smalltalk project at Xerox PARC where it was conceived as a way to organize some early GUI applications. Some of the fine detail of the original MVC pattern was tied to Smalltalk-specific concepts, such as *screens* and *tools*, but the broader ideas are still applicable to applications, and they are especially well-suited to web applications.

The MVC pattern first took hold in the server-side end of web development, through toolkits like Ruby on Rails and the ASP.NET MVC Framework. In recent years, the MVC pattern has been seen as a way to manage the growing richness and complexity of client-side web development as well, and it is in this environment that AngularJS has emerged.

The key to applying the MVC pattern is to implement the key premise of a *separation of concerns*, in which the data model in the application is decoupled from the business and presentation logic. In client-side web development, this means separating the data, the logic that operates on that data, and the HTML elements used to display the data. The result is a client-side application that is easier to develop, maintain, and test.

The three main building blocks are the *model*, the *controller*, and the *view*. In Figure 3-2, you can see the traditional exposition of the MVC pattern as it applies to server-side development.

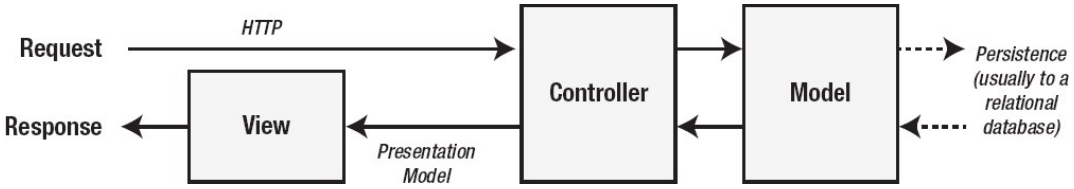

Figure 3-2: The server-side implementation of the MVC pattern

I took this figure from my *Pro ASP.NET MVC Framework* book, which describes Microsoft's server-side implementation of the MVC pattern. You can see that the expectation is that the model is obtained from a database and that the goal of the application is to service HTTP requests from the browser. This is the basis for round-trip web apps, which I described earlier.

Of course, AngularJS exists in the browser, which leads to a twist on the MVC theme, as illustrated in Figure 3-3.

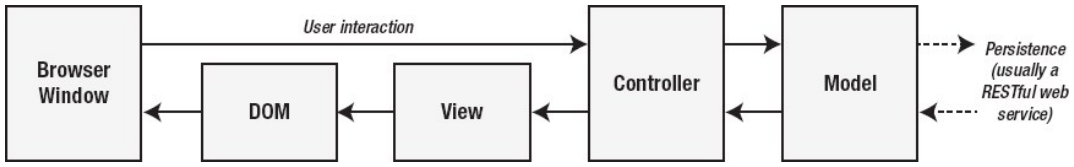

Figure 3-3: The AngularJS implementation of the MVC pattern

The client-side implementation of the MVC pattern gets its data from server-side components, usually via a RESTful web service, which I describe in Chapter 5. The goal of the controller and the view is to operate on the data in the model in order to perform DOM manipulation so as to create and manage HTML elements that the user can interact with. Those interactions are fed back to the controller, closing the loop to form an interactive application.

Tip Using an MVC framework like AngularJS in the client doesn't preclude using a server-side MVC framework, but you'll find that an AngularJS client takes on some of the complexity that would have otherwise existed at the server. This is generally a good thing because it offloads work from the server to the client, and that allows for more clients

to be supported with less server capacity.

Patterns and Pattern Zealots

A good pattern describes an approach to solving a problem that has worked for *other* people on *other* projects. Patterns are recipes, rather than rules, and you will need to adapt any pattern to suit your specific projects, just like a cook has to adapt a recipe to suit different ovens and ingredients.

The degree by which you depart from a pattern should be driven by experience. The time you have spent applying a pattern to similar projects will inform your knowledge about what does and doesn't work for you. If you are new to a pattern or you are embarking on a new kind of project, then you should stick as closely as possible to the pattern until you truly understand the benefits and pitfalls that await you. Be careful not to reform your entire development effort around a pattern, however, since wide-sweeping disruption usually causes productivity loses that undermine whatever outcome you were hoping the pattern would give.

Patterns are flexible tools and not fixed rules, but not all developers understand the difference, and some become *pattern zealots*. These are the people who spend more time talking about the pattern than applying it to projects and consider any deviation from their interpretation of the pattern to be a serious crime. My advice is to simply ignore this kind of person because any kind of engagement will just suck the life out of you, and you'll never be able to change their minds. Instead, just get on with some work and demonstrate how a flexible application of a pattern can produce good results through practical application and delivery.

With this in mind, you will see that I follow the broad concepts of the MVC pattern in the examples in this book but that I adapt the pattern to demonstrate different features and techniques. And this is how I work in my own projects—embracing the parts of patterns that provide value and setting aside those that do not.

Understanding Models

Models—the *M* in *MVC*—contain the data that users work with. There are two broad types of model: *view models*, which represent just data passed from the controller to the view, and *domain models*, which contain the data in a business domain, along with the operations, transformations, and rules for creating, storing, and manipulating that data, collectively referred to as the *model logic*.

Tip Many developers new to the MVC pattern get confused with the idea of including logic in the data model, believing that the goal of the MVC pattern is to separate data from logic. This is a misapprehension: The goal of the MVC framework is to divide up an application into three functional areas, each of which may contain both logic *and* data. The goal isn't to eliminate logic from the model. Rather, it is to ensure that the model contains logic only for creating and managing the model data.

You can't read a definition of the MVC pattern without tripping over the word *business*, which is unfortunate because a lot of web development goes far beyond the line-of-business applications that led to this kind of terminology. Business applications are still a big chunk of the development world, however, and if you are writing, say, a sales accounting system, then your business domain would encompass the process related to sales accounting, and your domain model would contain the accounts data and the logic by which accounts are created, stored, and managed. If you are creating a cat video web site, then you still have a business domain; it is just that it might not fit within the structure of a corporation. Your domain model would contain the cat videos and the logic that will create, store, and manipulate those videos.

Many AngularJS models will effectively push the logic to the server side and invoke it via a RESTful web service because there is little support for data persistence within the browser and it is simply easier to get the data you require over Ajax. I describe the basic support that AngularJS provides for Ajax in Chapter 20 and for RESTful services in Chapter 21.

Tip There *are* client-side persistence APIs defined as part of the HTML5 standard effort. The quality of these standards is currently mixed, and the implementations vary in quality. The main problem, however, is that most users still rely on browsers that don't implement the new APIs, and this is especially true in corporate environments, where Internet Explorer 6/7/8 are still widely used because of problems migrating line-of-business applications to standard versions of HTML.

For each component in the MVC pattern, I'll describe what should and should not be included. The model in an application built using the MVC pattern *should*

- Contain the domain data
- Contain the logic for creating, managing, and modifying the domain data (even if that means executing remote logic via web services)
- Provide a clean API that exposes the model data and operations on it

The model *should not*

- Expose details of how the model data is obtained or managed (in other words, details of the data storage mechanism or the remote web service should not be exposed to controllers and views)
- Contain logic that transforms the model based on user interaction (because this is the controller's job)
- Contain logic for displaying data to the user (this is the view's job)

The benefits of ensuring that the model is isolated from the controller and views are that you can test your logic more easily (I describe AngularJS unit testing in Chapter 25) and that enhancing and maintaining the overall application is simpler and easier.

The best domain models contain the logic for getting and storing data persistently and for create, read, update, and delete operations (known collectively as CRUD). This can mean the model contains the logic directly, but more often the model will contain the logic for calling RESTful web services to invoke server-side database operations (which I demonstrate in Chapter 8 when I built a realistic AngularJS application and which I describe in detail in Chapter 21).

Understanding Controllers

Controllers are the connective tissue in an AngularJS web app, acting as conduits between the data model and views. Controllers add business domain logic (known as *behaviors)* to *scopes*, which are subsets of the model.

> **Tip** Other MVC frameworks use slightly different terminology. So, for example, if you are an ASP.NET MVC Framework developer (my preferred server-side framework), then you will be familiar with the idea of *action methods*, rather than *behaviors*. The intent and effect are the same, however, and any MVC skills you have developed through server-side development will help you with AngularJS development.

A controller built using the MVC *should*

- Contain the logic required to initialize the scope
- Contain the logic/behaviors required by the view to present data from the scope
- Contain the logic/behaviors required to update the scope based on user interaction

The controller *should not*

- Contain logic that manipulates the DOM (that is the job of the view)
- Contain logic that manages the persistence of data (that is the job of the model)
- Manipulate data outside of the scope

From these lists, you can tell that scopes have a big impact on the way that controllers are defined and used. I describe scopes and controllers in detail in Chapter 13.

Understanding View Data

The domain model isn't the only data in an AngularJS application. Controllers can create *view data* (also known as *view model data* or *view models)* to simplify the definition of views. View data is not persistent and is created either by synthesizing some aspect of the domain model data or in response to user interaction. You saw an example of view data in Chapter 2, when I used the `ng-model` directive to capture the text entered by the user in an `input` element. View data is usually created and accessed via the controller's scope, as I describe in Chapter 13.

Understanding Views

AngularJS views are defined using HTML elements that are enhanced and that generate HTML by the use of data bindings and directives. It is the AngularJS directives that make views so flexible, and they transform HTML elements into the foundation for dynamic web apps. I explain data bindings in detail in Chapter 10 and describe how to use built-in and custom directives in Chapters 10-17. Views *should*

- Contain the logic and markup required to present data to the user

Views *should not*

- Contain complex logic (this is better placed in a controller)
- Contain logic that creates, stores, or manipulates the domain model

Views *can* contain logic, but it should be simple and used sparingly. Putting anything but the simplest method calls or expressions in a view makes the overall application harder to test and maintain.

Understanding RESTful Services

As I explained in the previous chapter, the logic for domain models in AngularJS apps is often split between the client and the server. The server contains the persistent store, typically a database, and contains the logic for managing it. In the case of a SQL database, for example, the required logic would include opening connections to the database server, executing SQL queries, and processing the results so they can be sent to the client.

We don't want the client-side code accessing the data store directly—doing so would create a tight coupling between the client and the data store that would complicate unit testing and make it difficult to change the data store without also making changes to the client code as well.

By using the server to mediate access to the data store, we prevent tight coupling. The logic on the client is responsible for getting the data to and from the server and is unaware of the details of how that data is stored or accessed behind the scenes.

There are lots of ways of passing data between the client and the server. One of the most common is to use *Asynchronous JavaScript and XML* (Ajax) requests to call server-side code, getting the server to send JSON and making changes to data using HTML forms. (This is what I did at the end of Chapter 2 to get the to-do data from the server. I requested a URL that I knew would return the JSON content I required.)

> **Tip** Don't worry if you are not familiar with JSON. I introduce it properly in Chapter 5.

This approach can work well and is the foundation of *RESTful web services*, which use the nature of HTTP requests to perform create, read, update, and delete (CRUD) operations on data.

> **Note** REST is a style of API rather than a well-defined specification, and there is disagreement about what exactly makes a web service RESTful. One point of contention is that purists do not consider web services that return JSON to be RESTful. Like any disagreement about an architectural pattern, the reasons for the disagreement are arbitrary and dull and not at all worth worrying about. As far as I am concerned, JSON services *are* RESTful, and I treat them as such in this book.

In a RESTful web service, the operation that is being requested is expressed through a combination of the HTTP method and the URL. So, for example, imagine a URL like this one:

http://myserver.mydomain.com/people/bob

There is no standard URL specification for a RESTful web service, but the idea is to make the URL self-explanatory, such that it is obvious what the URL refers to. In this case, it is obvious that there is a collection of data objects called `people` and that the URL refers to the specific object within that collection whose identity is `bob`.

> **Tip** It isn't always possible to create such self-evident URLs in a real project, but you should make a serious effort to keep things simple and not expose the internal structure of the data store through the URL (because this is just another kind of coupling between components). Keep your URLs as simple and clear as possible, and keep the mappings between the URL format and the data store structure within the server.

The URL identifies the data object that I want to operate on, and the HTTP method specifies what operation I want performed, as described in Table 3-1.

Table 3-1: The Operations Commonly Performed in Response to HTTP Methods

Method	Description
`GET`	Retrieves the data object specified by the URL
`PUT`	Updates the data object specified by the URL
`POST`	Creates a new data object, typically using form data values as the data fields
`DELETE`	Deletes the data object specified by the URL

You don't have to use the HTTP methods to perform the operations I describe in the table. A common variation is that the POST method is often used to serve double duty and will update an object if one exists and create one if not, meaning that the PUT method isn't used. I describe the support that AngularJS provides for Ajax in Chapter 20 and for easily working with RESTful services in Chapter 21.

Idempotent HTTP Methods

You can implement any mapping between HTTP methods and operations on the data store, although I recommend you stick as closely as possible to the convention I describe in the table.

If you depart from the normal approach, make sure you honor the nature of the HTTP methods as defined in the HTTP specification. The GET method is *nullipotent*, which means that the operations you perform in response to this method should only retrieve data and not modify it. A browser (or any intermediate device, such as a proxy) expects to be able to repeatedly make a GET request without altering the state of the server (although this doesn't mean the state of the server won't change between identical GET requests because of requests from other clients).

The PUT and DELETE methods are *idempotent*, which means that multiple identical requests should have the same effect as a single request. So, for example, using the DELETE method with the `/people/bob` URL should delete the `bob` object from the `people` collection for the first request and then do nothing for subsequent requests. (Again, of course, this won't be true if another client re-creates the `bob` object.)

The POST method is neither nullipotent nor idempotent, which is why a common RESTful optimization is to handle object creation *and* updates. If there is no `bob` object, using the POST method will create one, and subsequent POST requests to the same URL will update the object that was created.

All of this is important only if you are implementing your own RESTful web service. If you are writing a client that consumes a RESTful service, then you just need to know what data operation each HTTP method corresponds to. I demonstrate consuming such a service in Chapter 6 and explain the AngularJS support for REST in detail in Chapter 21.

Common Design Pitfalls

In this section, I describe the three most common design pitfalls that I encounter in AngularJS projects. These are not coding errors but rather problems with the overall shape of the web app that prevent the project team from getting the benefits that AngularJS and the MVC pattern can provide.

Putting the Logic in the Wrong Place

The most common problem is logic put into the wrong component such that it undermines the MVC separation of concerns. Here are the three most common varieties of this problem:

- Putting business logic in views, rather than in controllers
- Putting domain logic in controllers, rather than in model
- Putting data store logic in the client model when using a RESTful service

These are tricky issues because they take a while to manifest themselves as problems. The application still runs, but it will become harder to enhance and maintain over time. In the case of the third variety, the problem will become apparent only

when the data store is changed (which rarely happens until a project is mature and has grown beyond its initial user projections).

> **Tip** Getting a feel for where logic should go takes some experience, but you'll spot problems earlier if you are using unit testing because the tests you have to write to cover the logic won't fit nicely into the MVC pattern. I describe the AngularJS support for unit testing in Chapter 25.

Knowing where to put logic becomes second nature as you get more experience in AngularJS development, but here are the three rules:

- View logic should prepare data only for display and never modify the model.
- Controller logic should never directly create, update, or delete data from the model.
- The client should never directly access the data store.

If you keep these in mind as you develop, you'll head off the most common problems.

Adopting the Data Store Data Format

The next problem arises when the development team builds an application that depends on the quirks of the server-side data store. I recently worked with a project team that had built their client so that it honored the data format quirks of their server-side SQL server. The problem they ran into—and the reason why I was involved—was they needed to upgrade to a more robust database, which used different representations for key data types.

In a well-designed AngularJS application that gets its data from a RESTful service, it is the job of the server to hide the data store implementation details and present the client with data in a suitable data format that favors simplicity in the client. Decide how the client needs to represent dates, for example, and then ensure you use that format within the data store—and if the data store can't support that format natively, then it is the job of the server to perform the translation.

Clinging to the Old Ways

One of the most powerful features of AngularJS is that it builds on jQuery, especially for its directives feature, which I describe in Chapter 15. The problem this presents, however, is that it makes it easy to notionally use AngularJS for a project but really end up using jQuery behind the scenes.

This may not seem like a design problem, but it ends up distorting the shape of the application because jQuery doesn't make it easy to separate the MVC components, and that makes it difficult to test, enhance, and maintain the web app you are creating. If you are manipulating the DOM directly from jQuery in an AngularJS app, then you have a problem.

As I explained earlier in the chapter, AngularJS isn't the right tool for every job, and it is important you decide at the start of the project which tools you are going to use. If you are going to use AngularJS, then you need to make sure you don't fall back to sneaky jQuery shortcuts that, in the end, will cause endless problems. I return to this topic in Chapter 15 when I introduce jqLite, the AngularJS jQuery implementation, and throughout Chapters 15-17, when I show you how to create custom directives.

Summary

In this chapter, I provided some context for AngularJS. I described the kinds of project where it makes sense to use AngularJS (and where it doesn't), I explained how AngularJS supports the MVC pattern for app development, and I gave a brief overview of REST and how it is used to express data operations over HTTP requests. I finished the chapter by describing the three most common design problems in AngularJS projects. In the next chapter, I provide a quick primer for HTML and the Bootstrap CSS framework that I use for examples throughout this book.