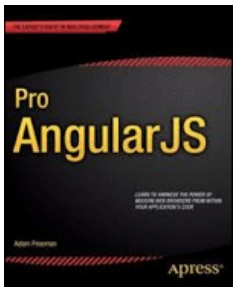

Pro AngularJS

by Adam Freeman

Chapter 9: The Anatomy of an AngularJS App

Overview

AngularJS applications follow the MVC pattern that I described in Chapter 3, but the development process itself relies on a wider range of building blocks. There are, of course, the headline building blocks—the model, the views, and the controllers—but there are lots of other moving parts in an AngularJS app as well, including *modules, directives, filters, factories*, and *services*. During development you use these smaller components to flesh out the bigger MVC pattern.

The different types of AngularJS component are tightly integrated, and to demonstrate one feature, I'll often have to use several others. I have to start somewhere, but I don't want you to have to wait until the end of the book to know what all the components do, especially since part of the joy of developing with AngularJS is the way you reshape your application using different building blocks as the functionality grows and becomes more complex.

I have written this chapter so that I have a context in which to talk about the broader AngularJS environment, written from the perspective of the top-level component: the *module*. I explain the different roles that modules play in an AngularJS application, show you how to create them, and demonstrate how they act as gateways to the major AngularJS features that you will come to know and love.

In the chapters ahead, I'll rely on features that I have yet to describe in detail; when that happens, you can return to this chapter to put those features in context, see how they fit into the wider AngularJS application, see a simple example, and learn where in the book I provide more details. Think of this chapter as a thumbnail sketch that provides a preview and a context for the rest of the book. There are some important concepts that set the foundation for AngularJS development, such as *dependency injection* and *factory functions*, and a lot of references to other chapters where more detailed descriptions can be found, including *why* a specific type of component is useful and *when* it should be applied. Table 9-1 summarizes this chapter.

Table 9-1: Chapter Summary

Problem	Solution	Listing
Create an AngularJS module.	Use the `angular.module` method.	1, 2
Set the scope of a module.	Use the `ng-app` attribute.	3
Define a controller.	Use the `Module.controller` method.	4, 8
Apply a controller to a view.	Use the `ng-controller` attribute.	5, 7
Pass data from a controller to a view.	Use the `$scope` service.	6
Define a directive.	Use the `Module.directive` method.	9
Define a filter.	Use the `Module.filter` method.	10
Use a filter programmatically.	Use the `$filter` service.	11
Define a service.	Use the `Module.service`, `Module.factory`, or `Module.provider` method.	12
Define a service from an existing object or value.	Use the `Module.value` method.	13
Add structure to the code in an application.	Create multiple modules and declare dependencies from the module referenced by the `ng-app` attribute.	14-16
Register functions that are called when modules are loaded.	Use the `Module.config` and `Module.run` methods.	17

Preparing the Example Project

I am going to return to a simple project structure for the examples in this part of the book. Remove the contents of the `angularjs` folder and add the `angular.js`, `bootstrap.css`, and `bootstrap-theme.css` files as described in Chapter 1. Create an HTML file called `example.html` and set the content to match Listing 9-1.

Listing 9-1: The Contents of the example.html File

```
<!DOCTYPE html>
<html ng-app="exampleApp" >
<head>
    <title>AngularJS Demo</title>
    <link href="bootstrap.css" rel="stylesheet" />
    <link href="bootstrap-theme.css" rel="stylesheet" />
    <script src="angular.js"></script>
    <script>

        var myApp = angular.module("exampleApp", []);

        myApp.controller("dayCtrl", function ($scope) {
            // controller statements will go here
        });

    </script>
</head>
<body>
    <div class="panel" ng-controller="dayCtrl">
        <div class="page-header">
            <h3>AngularJS App</h3>
        </div>
        <h4>Today is {{day || "(unknown)"}}</h4>
    </div>
</body>
</html>
```

The listing contains the plumbing for a minimal AngularJS app, which I'll explain in the sections that follow. The view for this app contains a data binding expression that isn't set up at the moment, so I have used the JavaScript `||` operator, which I described in Chapter 5, to display the value of the `day` variable if it is defined and the string `(unknown)` otherwise. Figure 9-1 shows how this HTML document is displayed by the browser.

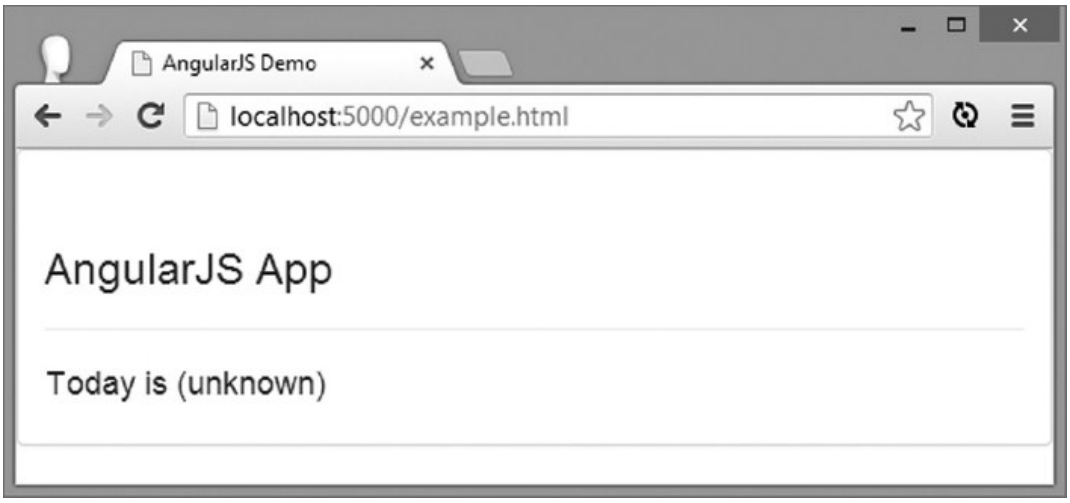

Figure 9-1: Displaying the example HTML document in the browser

Working with Modules

Modules are the top-level components for AngularJS applications. You can actually build simple AngularJS apps without needing to reference modules at all, but I don't recommend doing so because simple applications become complex applications over time, and you'll just end up having to rewrite the application when it becomes unmanageable. Working with modules is easy, and the handful of additional JavaScript statements that you need to set up and manage modules is a worthwhile investment. Modules have three main roles in an AngularJS app:

- To associate an AngularJS application with a region of an HTML document
- To act as a gateway to key AngularJS framework features
- To help organize the code and components in an AngularJS application

I explain each of these functions in the sections that follow.

Setting the Boundaries of an AngularJS Application

The first step when creating an AngularJS app is to define a module and associate it with a region of the HTML document. Modules are defined with the `angular.module` method. Listing 9-2 shows the statement from the example that creates the module for the example app.

Listing 9-2: Creating a Module

```
…
var myApp = angular.module("exampleApp", []);
…
```

The `module` method supports the three arguments described in Table 9-2, although it is common to use only the first two.

Table 9-2: The Arguments Accepted by the angular.module Method

Name	Description
`name`	The name of the new module
`requires`	The set of modules that this module depends on
`config`	The configuration for the module, equivalent to calling the `Module.config` method—see the "Working with the Module Life Cycle" section

When creating a module that will be associated with an HTML document (as opposed to organizing code, which I describe shortly), the convention is to give the module a name with the suffix `App`. In the example, I used the name `exampleApp` for my module, and the benefit of this convention is that it makes it clear which module represents the top-level AngularJS application in your code structure—something that can be useful in complex apps that can contain multiple modules.

Defining a module in JavaScript is only part of the process; the module must also be applied to the HTML content using the `ng-app` attribute. When AngularJS is the only web framework being used, the convention is to apply the `ng-app` attribute to the `html` element, as illustrated by Listing 9-3, which shows the element from the `example.html` file to which the `ng-app` element has been applied.

Listing 9-3: Applying the `ng-app` Attribute in the example.html File

```
…
<html ng-app="exampleApp">
…
```

The `ng-app` attribute is used during the *bootstrap* phase of the AngularJS life cycle, which I describe later in this chapter (and which is not to be confused with the Bootstrap CSS framework that I described in Chapter 4).

Avoiding the Module Create/Lookup Trap

When creating a module, you must specify the `name` and `requires` arguments, even if your module has no dependencies. I explain how dependencies work later in this chapter, but a common mistake is to omit the `requires` argument, like this:

```
…
var myApp = angular.module("exampleApp");
…
```

This has the effect of trying to locate a previously created module called `exampleApp`, rather than creating one, and will usually result in an error (unless there is already a module by that name, in which case you will usually get some unexpected behavior).

Using Modules to Define AngularJS Components

The `angular.module` method returns a `Module` object that provides access to the most important features that AngularJS provides via the properties and methods I have described in Table 9-3. As I explained at the start of this chapter, the features that the `Module` object provides access to are the features that I spend a lot of this book describing. I provide a brief demonstration and explanation of the most important features in the section and include references to the chapters in which you can find more details.

Table 9-3: The Members of the Module Object

Name	Description
`animation(name, factory)`	Supports the animation feature, which I describe in Chapter 23.
`config(callback)`	Registers a function that can be used to configure a module when it is loaded. See the "Working with the Module Life Cycle" section for details.
`constant(key, value)`	Defines a service that returns a constant value. See the "Working with the Module Life Cycle" section later in this chapter.
`controller(name, constructor)`	Creates a controller. See Chapter 13 for details.
`directive(name, factory)`	Creates a directive, which extends the standard HTML vocabulary. See Chapters 15-17.
`factory(name, provider)`	Creates a service. See Chapter 18 for details and an explanation of how this method differs from the `provider` and `service` methods.
`filter(name, factory)`	Creates a filter that formats data for display to the user. See Chapter 14 for details.
`provider(name, type)`	Creates a service. See Chapter 18 for details and an explanation of how this method differs from the `service` and `factory` methods.
`name`	Returns the name of the module.
`run(callback)`	Registers a function that is invoked after AngularJS has loaded and configured all of the modules. See the "Working with the Module Life Cycle" section for details.
`service(name, constructor)`	Creates a service. See Chapter 18 for details and an explanation of how this method differs from the `provider` and `factory` methods.
`value(name, value)`	Defines a service that returns a constant value; see the "Defining Values" section later in this chapter.

The methods defined by the `Module` object fall into three broad categories: those that define components for an AngularJS application, those that make it easier to create those building blocks, and those that help manage the AngularJS life cycle. I'll start by introducing the building blocks and then talk about the other features that are available.

Defining Controllers

Controllers are one of the big building blocks of an AngularJS application, and they act as a conduit between the model and the views. Most AngularJS projects will have multiple controllers, each of which delivers the data and logic required for one aspect of the application. I describe controllers in depth in Chapter 13.

Controllers are defined using the `Module.controller` method, which takes two arguments: the name of the controller and a *factory* function, which is used to set up the controller and get it ready for use (see the "Factory and Worker Functions" sidebar later in the chapter for more details). Listing 9-4 shows the statements from the `example.html` file that create the controller.

Listing 9-4: Creating a Controller in the example.html File

```
…
myApp.controller("dayCtrl", function ($scope) {
    // controller statements will go here
});
…
```

The convention for controller names is to use the suffix `Ctrl`. The statement in the listing creates a new controller called `dayCtrl`. The function passed to the `Module.controller` method is used to declare the controller's *dependencies*, which are the AngularJS components that the controller requires. AngularJS provides some built-in services and features

that are specified using argument names that start with the `$` symbol. In the listing you can see that I have specified the `$scope`, which asks AngularJS to provide the scope for the controller. To declare a dependency on `$scope`, I just have to use the name as an argument to the factory function, like this:

```
…
myApp.controller("dayCtrl", function ($scope) {
…
```

This is an example of *dependency injection* (DI), where AngularJS inspects the arguments that are specified for a function and locates the components they correspond to; see the "Understanding Dependency Injection" sidebar for details. The function I passed to the `controller` method has an argument called `x`, and AngularJS will automatically pass in the scope object when the function is called. I explain how services work in Chapter 18 and show you how scopes work in Chapter 13.

Understanding Dependency Injection

One of the AngularJS features that causes the most confusion is dependency injection (DI). It can be hard to figure out what DI is, how it works, and why it is useful. Even if you have encountered dependency injection in other frameworks, AngularJS takes an unusual approach and mixes in some features that are distinct from other languages.

As you will learn as you read this chapter, an AngularJS application consists of different components: controllers, directives, filters, and so on. I describe each of them and give a little example.

The place to start is to understand the problem that DI sets out to solve. Some of the components in an AngularJS application will depend on others. In Listing 9-4, my controller needs to use the `$scope` component, which allows it to pass data to the view. This is an example of a *dependency—my* controller *depends* on the `$scope` component to perform its work.

Dependency injection simplifies the process of dealing with dependencies—known as *resolving a dependency*—between components. Without DI, I would have to locate `$scope` myself somehow, probably using global variables. It would work, but it wouldn't be as simple as the AngularJS technique.

A component in an AngularJS application *declares its dependencies* by defining arguments on its factory function whose names match the components it depends on. In this example, AngularJS inspects the arguments of my controller function, determines that it depends on the `$scope` component, locates `$scope` for me, and passes it as an argument to the factory function when it is invoked.

To put it another way, DI changes the purpose of function arguments. Without DI, arguments are used to *receive* whatever objects the caller wants to pass, but with DI, the function uses arguments to make *demands*, telling AngularJS what building blocks it needs.

One of the interesting side effects of the way that DI works in AngularJS is that the order of the arguments always matches the order in which the dependencies are declared. Consider this function:

```
…
myApp.controller("dayCtrl", function ($scope, $filter) {
…
```

The first argument passed to the function will be the `$scope` component, and the second will be the `$filter` service object. Don't worry about what the `$filter` object does for the moment. I introduce it later in this chapter. What's important is that the order in which you declare dependencies is honored by AngularJS. If I change the order of my dependencies, like this:

```
…
myApp.controller("dayCtrl", function ($filter, $scope) {
…
```

then AngularJS will pass me the `$filter` object as the first argument and the `$scope` object as the second. In short, it doesn't matter what order you define dependency-injected arguments. This may seem obvious, but it isn't the way that JavaScript usually works, and it can take some time to get used to. You may already have seen a similar technique used in other programming languages—this is known as *named parameters* in C#, for example.

The main benefit of using dependency injection during development is that AngularJS takes care of managing component and feeding them to your functions when they are needed. DI also provides benefits when testing your code,

because it allows you to easily replace real building blocks with *fake* or *mock objects* that let you focus on specific parts of your code; I explain how this works in Chapter 25.

Applying Controllers to Views

Defining controllers is only part of the process—they must also be applied to HTML elements so that AngularJS knows which part of an HTML document forms the view for a given controller. This is done through the `ng-controller` attribute, and Listing 9-5 shows the HTML elements from the `example.html` file that apply the `dayCtrl` controller to the HTML document.

Listing 9-5: Defining Views in the example.html File

```
…
<body>
    <div class="panel" ng-controller="dayCtrl">
        <div class="page-header">
            <h3>AngularJS App</h3>
        </div>
        <h4>Today is {{day || "(unknown)"}}</h4>
    </div>
</body>
…
```

The view in this example is the `div` element and its contents—in other words, the element to which the `ng-controller` attribute has been applied and the elements it contains.

The `$scope` component that I specified as an argument when I created the controller is used to provide the view with data, and only the data configured via `$scope` can be used in expressions and data bindings. At the moment, when you navigate to the `example.html` file with the browser, the data binding generates the string `(unknown)` because I have used the `||` operator to coalesce `null` values, like this:

```
…
<h4>Today is {{ day || "(unknown)" }}</h4>
…
```

A nice feature of AngularJS data bindings is that you can use them to evaluate JavaScript expressions. This binding will display the value of the `day` property provided by the `$scope` component unless it is `null`, in which case `(unknown)` will be displayed instead. To provide a value for the `day` property, I must assign it to the `$scope` in the controller setup function, as shown in Listing 9-6.

Listing 9-6: Defining a Model Data Value in the example.html File

```
…
<script>

    var myApp = angular.module("exampleApp", []);

    myApp.controller("dayCtrl", function ($scope) {
        var dayNames = ["Sunday", "Monday", "Tuesday", "Wednesday",
            "Thursday", "Friday", "Saturday"];
        $scope.day = dayNames[new Date().getDay()];
    });

</script>
…
```

I create a new `Date`, call the `getDay` method to get the numeric day of the week, and look up the day name from an array of string values. As soon as I make this addition to the `script` element, the value I have specified is available to the view and is used in the HTML output, as shown in Figure 9-2.

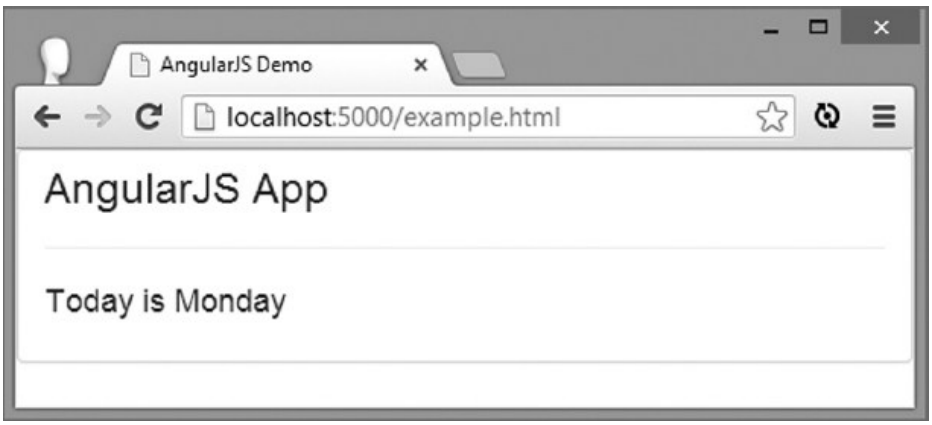

Figure 9-2: The effect of defining a variable using the $scope service

Creating Multiple Views

Each controller can support multiple views, which allows the same data to be presented in different ways or for closely related data to be created and managed efficiently. In Listing 9-7, you can see how I have added a `data` property to `$scope` and created a second view that takes advantage of it.

Listing 9-7: Adding a Second View to the example.html File

```
<!DOCTYPE html>
<html ng-app="exampleApp">
<head>
    <title>AngularJS Demo</title>
    <link href="bootstrap.css" rel="stylesheet" />
    <link href="bootstrap-theme.css" rel="stylesheet" />
    <script src="angular.js"></script>
    <script>

        var myApp = angular.module("exampleApp", []);

        myApp.controller("dayCtrl", function ($scope) {
            var dayNames = ["Sunday", "Monday", "Tuesday", "Wednesday",
                "Thursday", "Friday", "Saturday"];
            $scope.day = dayNames[new Date().getDay()];
            $scope.tomorrow = dayNames[(new Date().getDay() + 1) % 7];
        });

    </script>
</head>
<body>
    <div class="panel">
        <div class="page-header">
            <h3>AngularJS App</h3>
        </div>
        <h4 ng-controller="dayCtrl">Today is {{day || "(unknown)"}}</h4>
        <h4 ng-controller="dayCtrl">Tomorrow is {{tomorrow || "(unknown)"}}</h4>
    </div>
</body>
</html>
```

I have moved the `ng-controller` attribute so that I can create two simple views side-by-side in the HTML document; you can see the effect in Figure 9-3.

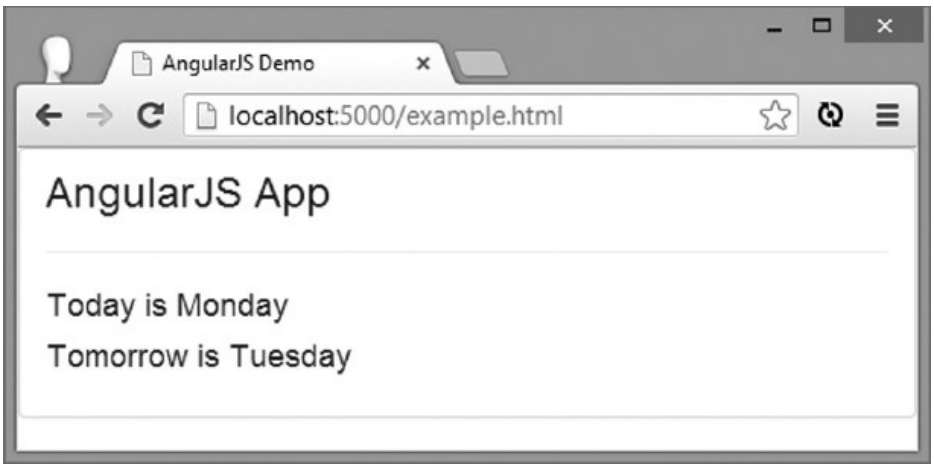

Figure 9-3: Adding a controller

I could have achieved the same result within a single view, of course, but I want to demonstrate different ways in which controllers and views can be used.

Creating Multiple Controllers

All but the simplest applications will contain multiple controllers, each of which will be responsible for a different aspect of the application functionality. Listing 9-8 shows how I have added a second controller to the `example.html` file.

Listing 9-8: Adding a Second Controller to the example.html File

```
<!DOCTYPE html>
<html ng-app="exampleApp">
<head>
    <title>AngularJS Demo</title>
    <link href="bootstrap.css" rel="stylesheet" />
    <link href="bootstrap-theme.css" rel="stylesheet" />
    <script src="angular.js"></script>
    <script>

        var myApp = angular.module("exampleApp", []);

        myApp.controller("dayCtrl", function ($scope) {
            var dayNames = ["Sunday", "Monday", "Tuesday", "Wednesday",
                "Thursday", "Friday", "Saturday"];
            $scope.day = dayNames[new Date().getDay()];
        });

        myApp.controller("tomorrowCtrl", function ($scope) {
            var dayNames = ["Sunday", "Monday", "Tuesday", "Wednesday",
                "Thursday", "Friday", "Saturday"];
            $scope.day = dayNames[(new Date().getDay() + 1) % 7];
        });

    </script>
</head>
<body>
    <div class="panel">
        <div class="page-header">
            <h3>AngularJS App</h3>
        </div>
        <h4 ng-controller="dayCtrl">Today is {{day || "(unknown)"}}</h4>
         <h4 ng-controller="tomorrowCtrl">Tomorrow is {{day || "(unknown)"}}</h4>
    </div>
</body>
</html>
```

I have added a controller called `tomorrowCtrl` that works out tomorrow's name. I have also edited the HTML markup so that each controller has its own view. The result of these changes is the same as shown in Figure 9-3—only the way the

content is generated differs.

Tip Notice how I am able to use the `day` property in both views without the values interfering with each other. Each controller has its own part of the overall application scope, and the `day` property of the `dayCtrl` controller is isolated from the one defined by the `tomorrowCtrl` controller. I describe scopes in Chapter 13.

You would not create two controllers and two views for such a simple app in the real world, but I want to demonstrate the different features of modules, and this is a good foundation for doing so.

Using the Fluent API

The result of the methods defined by the `Module` object is the `Module` object itself. This is an odd-sounding but neat trick that allows for a *fluent API*, where multiple calls to methods are chained together. As a simple example, I can rewrite the script element from Listing 9-8 without needing to define the `myApp` variable, as follows:

```
…
<script>

    angular.module("exampleApp", [])
        .controller("dayCtrl", function ($scope) {
            var dayNames = ["Sunday", "Monday", "Tuesday", "Wednesday",
                "Thursday", "Friday", "Saturday"];
            $scope.day = dayNames[new Date().getDay()];
        })
        .controller("tomorrowCtrl", function ($scope) {
            var dayNames = ["Sunday", "Monday", "Tuesday", "Wednesday",
                "Thursday", "Friday", "Saturday"];
            $scope.day = dayNames[(new Date().getDay() + 1) % 7];
        });

</script>
…
```

I call the `angular.module` method and get a `Module` object as the result, on which I immediately call the `controller` method to set up the `dayCtrl` controller. The result from the `controller` method is the same `Module` object that I got when I called the `angular.module` method, so I can use it again to call the `controller` method to set up `tomorrowCtrl`.

Defining Directives

Directives are the most powerful AngularJS feature because they extend and enhance HTML to create rich web applications. There are lots of features to like in AngularJS, but directives are the most enjoyable and flexible to create. I describe the built-in directives that come with AngularJS in Chapters 10-12, but you can also create your own custom directives when the built-in ones don't meet your needs. I explain this process in detail in Chapters 15-17, but the short version is that custom directives are created via the `Module.directive` method. You can see a simple example of a custom directive in Listing 9-9.

Listing 9-9: Creating a Custom Directive in the example.html File

```
<!DOCTYPE html>
<html ng-app="exampleApp">
<head>
    <title>AngularJS Demo</title>
    <link href="bootstrap.css" rel="stylesheet" />
    <link href="bootstrap-theme.css" rel="stylesheet" />
    <script src="angular.js"></script>
    <script>

        var myApp = angular.module("exampleApp", []);

        myApp.controller("dayCtrl", function ($scope) {
            var dayNames = ["Sunday", "Monday", "Tuesday", "Wednesday",
                "Thursday", "Friday", "Saturday"];
            $scope.day = dayNames[new Date().getDay()];
```

```
        });

        myApp.controller("tomorrowCtrl", function ($scope) {
            var dayNames = ["Sunday", "Monday", "Tuesday", "Wednesday",
                "Thursday", "Friday", "Saturday"];
            $scope.day = dayNames[(new Date().getDay() + 1) % 7];
        });

        myApp.directive("highlight", function () {
            return function (scope, element, attrs) {
                if (scope.day == attrs["highlight"]) {
                    element.css("color", "red");
                }
            }
        });

    </script>
</head>
<body>
    <div class="panel">
        <div class="page-header">
            <h3>AngularJS App</h3>
        </div>
        <h4 ng-controller="dayCtrl" highlight="Monday">
            Today is {{day || "(unknown)"}}
        </h4>
        <h4 ng-controller="tomorrowCtrl">Tomorrow is {{day || "(unknown)"}}</h4>
    </div>
</body>
</html>
```

There are different ways to create a custom directive, and the listing shows one of the simplest. I have called the `Module.directive` method, providing the name of the directive I want to create and a factory function that creates the directive.

Factory and Worker Functions

All of the `Module` methods that create AngularJS building blocks accept functions as arguments. These are often *factory functions*, so called because they are responsible for creating the object that AngularJS will employ to perform the work itself. Often, factory functions will return a *worker function*, which is to say that the object that AngularJS will use to perform some work is a function, too. You can see an example of this when I call the `directive` method in Listing 9-9. The second argument to the directive method is a factory function, as follows:

```
…
myApp.directive("highlight", function () {
    return function (scope, element, attrs) {
        if (scope.day == attrs["highlight"]) {
            element.css("color", "red");
        }
}
});
…
```

The `return` statement in the factory function returns another function, which AngularJS will invoke each time it needs to apply the directive, and this is the *worker function:*

```
…
myApp.directive("highlight", function () {
    return function (scope, element, attrs) {
        if (scope.day == attrs["highlight"]) {
            element.css("color", "red");
        }
    }
});
…
```

It is important to understand that you can't rely on either the factory or worker function being called at a specific time. You call the `Module` method—`directive` in this case—when you want to register a building block. AngularJS will call the factory function when it wants to set up the building block and then calls the worker function when it needs to apply the building block, and these three events won't occur in an immediate sequence (in other words, other `Module` methods will be called before your factory function is invoked, and other factory functions will be invoked before your worker function is called).

Applying Directives to HTML Elements

The factory function in this example is responsible for creating a directive, which is a worker function that AngularJS calls when it encounters the directive in the HTML. To understand how a custom directive works, it helps to start with the way it is applied to an HTML element, as follows:

```
…
<h4 ng-controller="dayCtrl" highlight="Monday">
…
```

My custom directive is called `highlight`, and it is applied as an attribute (although there are other options—such as custom HTML elements—as I describe in Chapter 16). I have set the value of the `highlight` attribute to be `Monday`. The purpose of my custom directive is to highlight the contents of the element that it is applied to if the `day` model property corresponds to the attribute value.

The factory function I passed to the `directive` method is called when AngularJS encounters the `highlight` attribute in the HTML. The directive function that the factory function creates is invoked by AngularJS and is passed three arguments: the scope for the view, the element to which the directive has been applied, and the attributes of that element.

Tip Notice that the argument for the directive function is `scope` and not `$scope`. I explain why there is no `$` sign and what the difference is in Chapter 15.

The `scope` argument lets me inspect the data that is available in the view; in this case, it allows me to get the value of the `day` property. The `attrs` argument provides me with a complete set of the attributes that have been applied to the element, including the attribute that applies the directive: I use this to get the value of the `highlight` attribute. If the value of the `highlight` attribute and the `day` value from the scope match, then I use the `element` argument to configure the HTML content.

The `element` argument is a jqLite object, which is the cut-down version of jQuery that is included with AngularJS. The method I used in this example—`css`—sets the value of a CSS property. By setting the `color` property, I change the color of the text for the element. I explain the complete set of jqLite methods in Chapter 15. You can see the effect of the directive in Figure 9-4 (although you will have to change the value of the `highlight` attribute if you are not running the example on a Monday).

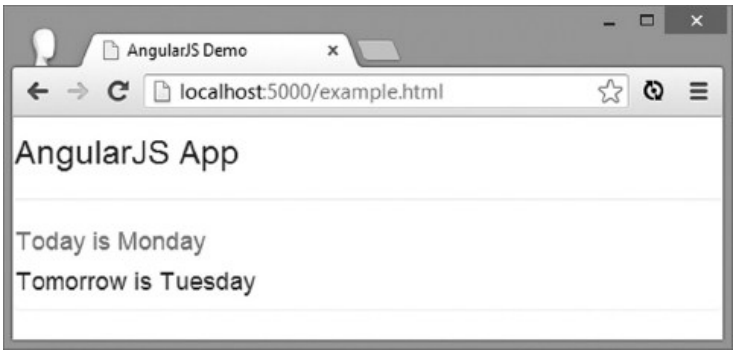

Figure 9-4: The effect of a custom directive

Defining Filters

Filters are used in views to format the data displayed to the user. Once defined, filters can be used throughout a module, which means you can use them to ensure consistency in data presentation across multiple controllers and views. In Listing 9-10, you can see how I have updated the `example.html` file to include a filter, and in Chapter 14, I explain the different ways that filters can be used, including using the built-in filters that come with AngularJS.

Listing 9-10: Adding a Filter to the example.html File

```
<!DOCTYPE html>
<html ng-app="exampleApp">
<head>
    <title>AngularJS Demo</title>
    <link href="bootstrap.css" rel="stylesheet" />
    <link href="bootstrap-theme.css" rel="stylesheet" />
    <script src="angular.js"></script>
    <script>

        var myApp = angular.module("exampleApp", []);

        myApp.controller("dayCtrl", function ($scope) {
            $scope.day = new Date().getDay();
        });

        myApp.controller("tomorrowCtrl", function ($scope) {
            $scope.day = new Date().getDay() + 1;
        });

        myApp.directive("highlight", function () {
            return function (scope, element, attrs) {
                if (scope.day == attrs["highlight"]) {
                    element.css("color", "red");
                }
            }
        });

        myApp.filter("dayName", function () {
            var dayNames = ["Sunday", "Monday", "Tuesday", "Wednesday",
                            "Thursday", "Friday", "Saturday"];
            return function (input) {
                return angular.isNumber(input) ? dayNames[input] : input;
            };
        });

    </script>
</head>
<body>
    <div class="panel">
        <div class="page-header">
            <h3>AngularJS App</h3>
        </div>
        <h4 ng-controller="dayCtrl" highlight="Monday">
            Today is {{day || "(unknown)" | dayName}}
        </h4>
        <h4 ng-controller="tomorrowCtrl">
            Tomorrow is {{day || "(unknown)" | dayName}}
        </h4>
    </div>
</body>
</html>
```

The `filter` method is used to define a filter, and the arguments are the name of the new filter and a factory function that will create the filter when invoked. Filters are themselves functions, which receive a data value and format it so it can be displayed.

My filter is called `dayName`, and I have used it to consolidate the code that transforms the numeric day of the week that I get from the `Date` objects into a name. My factory function defines the array of weekday names and returns a function that uses that array to transform numeric values:

```
…
return function (input) {
    return angular.isNumber(input) ? dayNames[input] : input;
};
…
```

I use the `angular.isNumber` method that I described in Chapter 5 to check that I am dealing with a numeric value and return the day name if I am. (To keep the example simple, I am not checking for values that are out of bounds.)

Applying Filters

Filters are applied in template expressions contained in views. The data binding or expression is followed by a bar (the `|` character) and then the name of the filter, as follows:

```
…
<h4 ng-controller="dayCtrl" highlight="Monday">
    Today is {{day || "(unknown)" | dayName}}
</h4>
…
```

Filters are applied after JavaScript expressions are evaluated, which allows me to use the `||` operator to check for `null` values and then the `|` operator to apply the filter. The result of this is that the value of the `day` property will be passed to the filter function if it is not `null`, and if it is, then `(unknown)` will be passed instead, which is why I used the `isNumber` method.

Fixing the Directive

If you have sharp eyes, you may have noticed when I added the filter, I managed to break the directive I created earlier. This is because my controllers now add a numeric representation of the current day to their scopes, rather than the formatted name. My directive checks for the value `Monday`, but it will only ever find 1, 2, and so on, and so will never highlight the date.

AngularJS development is full of little challenges like this because you will often refactor your code to move functionality from one component to another, just as I removed the name formatting from the controllers to a filter. There are several ways to solve this problem—including updating the directive to use numeric values as well—but the solution I want to demonstrate is a little more complex. In Listing 9-11, you can see the modifications I made to the definition of the directive.

Listing 9-11: Changing the Directive in the example.html File

```
…
myApp.directive("highlight", function ($filter) {

    var dayFilter = $filter("dayName");

    return function (scope, element, attrs) {
        if (dayFilter(scope.day) == attrs["highlight"]) {
            element.css("color", "red");
        }
    }
});
…
```

What I want to demonstrate with this example is that the building blocks that you create in an AngularJS application are not just limited to use on HTML elements; you can also use them in your JavaScript code.

In this case, I have added a `$filter` argument to my directive factory function, which tells AngularJS that I want to receive the filter service object when my function is called. The `$filter` service gives me access to all of the filters that have been defined, including my custom addition from the previous example. I obtain my filter by name, like this:

```
…
var dayFilter = $filter("dayName");
…
```

I receive the filter function that my factory creates, and I can then call that function to transform my numeric value to a name:

```
…
if (dayFilter(scope.day) == attrs["highlight"]) {
…
```

With this change, my directive works again. There are two important points to note in this example. The first is that refactoring code is a natural part of the AngularJS development process, and the second is that AngularJS makes

refactoring easier by providing both declarative (via HTML) and imperative (via JavaScript) access to the building blocks you create.

Defining Services

Services are *singleton* objects that provide any functionality that you want to use throughout an application. There are some useful built-in services that come with AngularJS for common tasks such as making HTTP requests. Some key AngularJS are delivered as services, including the `$scope` and `$filter` objects that I used in the earlier example. Since this is AngularJS, you can create your own services, a process that I demonstrate briefly here and describe in depth in Chapter 18.

> **Tip** *Singleton* means that only one instance of the object will be created by AngularJS and shared between the parts of the application that need the service.

Three of the methods defined by the `Module` object are used to create services in different ways: `service`, `factory`, and `provider`. All three are closely related, and I'll explain the differences between them in Chapter 18. For this chapter, I am going to use the `service` method to create a basic service to consolidate some of the logic in my example, as shown in Listing 9-12.

Listing 9-12: Creating a Simple Service in the example.html File

```
<!DOCTYPE html>
<html ng-app="exampleApp">
<head>
    <title>AngularJS Demo</title>
    <link href="bootstrap.css" rel="stylesheet" />
    <link href="bootstrap-theme.css" rel="stylesheet" />
    <script src="angular.js"></script>
    <script>

        var myApp = angular.module("exampleApp", []);

        myApp.controller("dayCtrl", function ($scope, days) {
            $scope.day = days.today;
        });

        myApp.controller("tomorrowCtrl", function ($scope, days) {
            $scope.day = days.tomorrow;
        });

        myApp.directive("highlight", function ($filter) {

            var dayFilter = $filter("dayName");

            return function (scope, element, attrs) {
                if (dayFilter(scope.day) == attrs["highlight"]) {
                    element.css("color", "red");
                }
            }
        });

        myApp.filter("dayName", function () {
            var dayNames = ["Sunday", "Monday", "Tuesday", "Wednesday",
                            "Thursday", "Friday", "Saturday"];
            return function (input) {
                return angular.isNumber(input) ? dayNames[input] : input;
            };
        });
        myApp.service("days", function () {
            this.today = new Date().getDay();
            this.tomorrow = this.today + 1;
        });

    </script>
</head>
<body>
```

```
    <div class="panel">
        <div class="page-header">
            <h3>AngularJS App</h3>
        </div>
        <h4 ng-controller="dayCtrl" highlight="Monday">
            Today is {{day || "(unknown)" | dayName}}
        </h4>
        <h4 ng-controller="tomorrowCtrl">
            Tomorrow is {{day || "(unknown)" | dayName}}
        </h4>
     </div>
</body>
</html>
```

The `service` method takes two arguments: the name of the service and a factory function that is called to create the service object. When AngularJS calls the factory function, it assigns a new object that is accessible via the `this` keyword, and I use this object to define `today` and `tomorrow` properties. This is a simple service, but it means I can access the `today` and `tomorrow` values via my service anywhere in my AngularJS code—something that helps simplify the development process when creating more complex applications.

Tip Notice that I am able to use the server from within the controllers, even though I call the `service` method after I call the `controller` method. You can create your component in any order, and AngularJS will ensure that everything is set up correctly before it starts calling factory functions and performing dependency injection. See the "Working with the AngularJS Life Cycle" section later in this chapter for more details.

I access my service by declaring a dependency for my `days` service, like this:

```
…
myApp.controller("tomorrowCtrl", function ($scope, days) {
…
```

AngularJS uses dependency injection to locate the `days` service and pass it as an argument to the factory function, which means I can then get the value of the `today` and `tomorrow` properties and use the `$scope` service to pass them to the view:

```
…
myApp.controller("tomorrowCtrl", function ($scope, days) {
    $scope.day = days.tomorrow;
});
…
```

I show you the other ways of creating services—including how to use the `service` method to take advantage JavaScript prototypes—in Chapter 18.

Defining Values

The `Module.value` method lets you create services that return fixed values and objects. This may seem like an odd thing to do, but it means you can use dependency injection for any value or object, not just the ones you create using module methods like `service` and `filter`. It makes for a more consistent development experience, simplifies unit testing, and allows you to use some advanced features, like *decoration*, which I describe in Chapter 24. In Listing 9-13, you can see how I have modified the `example.html` file to use a value.

Listing 9-13: Defining a Value in the example.html File

```
…
<script>

    var myApp = angular.module("exampleApp", []);

    // … statements omitted for brevity …

    var now = new Date();
    myApp.value("nowValue", now);

    myApp.service("days", function (nowValue) {
        this.today = nowValue.getDay();
```

```
        this.tomorrow = this.today + 1;
    });

</script>
...
```

In this listing I have defined a variable called `now`. I have assigned a new `Date` to the variable and then called the `Module.value` method to create the value service, which I called `nowValue`. I then declared a dependency on the `nowValue` service when I created my `days` service.

Using Objects Without Values

Using values may seem like an unnecessary complication, and you may not be persuaded by the argument for unit testing. Even so, you will find that creating AngularJS values is just simpler than *not* using them because AngularJS assumes any argument to a factory function declares a dependency that it needs to resolve. Developers who are new to AngularJS will often try to write code like this, which doesn't use a value:

```
...
var now = new Date();

myApp.service("days", function (now) {
    this.today = now.getDay();
    this.tomorrow = this.today + 1;
});
...
```

If you try to run this code, you will see an error like this one in the browser JavaScript console:

Error: Unknown provider: nowProvider <- now <- days

The problem here is that AngularJS won't use the local variable as the value for the `now` parameter when it calls the factory function, and the `now` variable will no longer be in scope when it is required.

If you are determined that you don't want to create AngularJS values—and most developers go through a phase of feeling this way—then you can rely on the JavaScript *closure* feature, which will let you reference variables from within functions when they are defined, like this:

```
...
var now = new Date();

myApp.service("days", function () {
    this.today = now.getDay();
    this.tomorrow = this.today + 1;
});
...
```

I removed the argument from the factory function, which means that AngularJS won't find a dependency to resolve. This code works, but it makes the `days` service harder to test, and my advice is to follow the AngularJS approach of creating value services.

Using Modules to Organize Code

In previous examples, I showed you how AngularJS uses dependency injection with factory functions when you create components such as controllers, filters, and services. Right at the start of the chapter, I explained that the second argument to the `angular.module` method, used to create modules, was an array of the module's dependencies:

```
...
var myApp = angular.module("exampleApp", []);
...
```

Any AngularJS module can rely on components defined in other modules, and this is a feature that makes it easier to organize the code in a complex application. To demonstrate how this works, I have added a JavaScript file called `controllers.js` to the `angularjs` folder. You can see the contents of the new file in Listing 9-14.

Listing 9-14: The Contents of the controller.js File

```
var controllersModule = angular.module("exampleApp.Controllers", [])

controllersModule.controller("dayCtrl", function ($scope, days) {
    $scope.day = days.today;
});

controllersModule.controller("tomorrowCtrl", function ($scope, days) {
    $scope.day = days.tomorrow;
});
```

In the `controllers.js file` I created a new module called `exampleApp.Controllers` and used it to define the two controllers from earlier examples. One common convention is to organize your application into modules that have the same type of component and to make it clear which building block a module contains by using the main module's name and appending the block type, which is why it's called `exampleApp.Controllers`. Similarly, I have created a second JavaScript file called `filters.js`, the contents of which are shown in Listing 9-15.

Listing 9-15: The Contents of the filters.js File

```
angular.module("exampleApp.Filters", []).filter("dayName", function () {
    var dayNames = ["Sunday", "Monday", "Tuesday", "Wednesday",
                    "Thursday", "Friday", "Saturday"];
    return function (input) {
        return angular.isNumber(input) ? dayNames[input] : input;
    };
});
```

I have created a module called `exampleApp.Filters` and used it to create the filter that was part of the main module in earlier examples. For variety, I have used the fluent API to call the `filter` method directly on the result of the module method (see the "Using the Fluent API" sidebar earlier in the chapter for details).

> **Tip** There is no requirement to put modules in their own files or to organize modules based on the building blocks they contain, but it is the approach I generally prefer and is a good place to start while you are figuring out your own AngularJS development processes and preferences.

In Listing 9-16, you can see how I have added `script` elements to import the `controllers.js` and `filters.js` files and added the modules they contain as dependencies to the main `exampleApp` module. I have also created two further modules within the `example.html` file to emphasize that modules don't have to be defined in their own files.

Listing 9-16: Using Module Dependencies in the example.html File

```
<!DOCTYPE html>
<html ng-app="exampleApp">
<head>
    <title>AngularJS Demo</title>
    <link href="bootstrap.css" rel="stylesheet" />
    <link href="bootstrap-theme.css" rel="stylesheet" />
    <script src="angular.js"></script>
    <script src="controllers.js"></script>
    <script src="filters.js"></script>
    <script>

        var myApp = angular.module("exampleApp",
            ["exampleApp.Controllers", "exampleApp.Filters",
             "exampleApp.Services", "exampleApp.Directives"]);

        angular.module("exampleApp.Directives", [])
            .directive("highlight", function ($filter) {

                var dayFilter = $filter("dayName");
                return function (scope, element, attrs) {
                    if (dayFilter(scope.day) == attrs["highlight"]) {
                        element.css("color", "red");
```

```
                    }
                }
            });

        var now = new Date();
        myApp.value("nowValue", now);

        angular.module ("exampleApp.Services", [])
            .service("days", function (nowValue) {
                this.today = nowValue.getDay();
                this.tomorrow = this.today + 1;
            });

    </script>
</head>
<body>
    <div class="panel">
        <div class="page-header">
            <h3>AngularJS App</h3>
        </div>
        <h4 ng-controller="dayCtrl" highlight="Monday">
            Today is {{day || "(unknown)" | dayName}}
        </h4>
        <h4 ng-controller="tomorrowCtrl">
            Tomorrow is {{day || "(unknown)" | dayName}}
        </h4>
    </div>
</body>
</html>
```

To declare the dependencies for the main module, I add each module's name to the array I pass as the second argument to the module, like this:

```
…
var myApp = angular.module("exampleApp", ["exampleApp.Controllers", "exampleApp.Filters",
    "exampleApp.Services", "exampleApp.Directives"]);
…
```

The dependencies don't have to be defined in any particular order, and you can define modules in any order as well. For example, I define the `exampleApp.Services` module in the listing after I have defined the `exampleApp` module that depends on it.

AngularJS loads all the modules that are defined in an application and then resolves the dependencies, merging the building blocks that each contains. This merging is important because it allows for the seamless use of functionality from other modules. As an example, the `days` service in the `exampleApp.Services` module depends on the `nowValue` value in the `exampleApp` module, and the directive in the `exampleApp.Directives` module relies on the filter from the `exampleApp.Filters` module.

You can put as much or as little functionality in other modules as you like. I have defined four modules in this example but left the value in the main module. I could have created a module just for values, a module that combined services and values, or a module with any other combination that suited my development style.

Working with the Module Life Cycle

The `Module.config` and `Module.run` methods register functions that are invoked at key moments in the life cycle of an AngularJS app. A function passed to the `config` method is invoked when the current module has been loaded, and a function passed to the `run` method is invoked when *all* modules have been loaded. You can see an example of both methods in use in Listing 9-17.

Listing 9-17: Using the config and run Methods in the example.html File

```
…
<script>

    var myApp = angular.module("exampleApp",
```

```
        ["exampleApp.Controllers", "exampleApp.Filters",
            "exampleApp.Services", "exampleApp.Directives"]);

    myApp.constant("startTime", new Date().toLocaleTimeString());
    myApp.config(function (startTime) {
        console.log("Main module config: " + startTime);
    });
    myApp.run(function (startTime) {
        console.log("Main module run: " + startTime);
    });

    angular.module("exampleApp.Directives", [])
        .directive("highlight", function ($filter) {

            var dayFilter = $filter("dayName");

            return function (scope, element, attrs) {
                if (dayFilter(scope.day) == attrs["highlight"]) {
                    element.css("color", "red");
                }
            }
        });

    var now = new Date();
    myApp.value("nowValue", now);

    angular.module("exampleApp.Services", [])
        .service("days", function (nowValue) {
            this.today = nowValue.getDay();
            this.tomorrow = this.today + 1;
        })
        .config(function() {
            console.log("Services module config: (no time)");
        })
        .run(function (startTime) {
            console.log("Services module run: " + startTime);
        });

</script>
...
```

My first change in this listing is to use the `constant` method, which is similar to the `value` method but creates a service that can be declared as a dependency by the `config` method (which you can't do when you create values).

The `config` method accepts a function that is invoked after the module on which the method is called is loaded. The `config` method is used to configure a module, usually by injecting values that have been obtained from the server, such as connection details or user credentials.

The `run` method also accepts a function, but it will be invoked only when all of the modules have been loaded and their dependencies have been resolved. Here is the sequence in which the callback functions are invoked:

1. The `config` callback on the `exampleApp.Services` module
2. The `config` callback on the `exampleApp` module
3. The `run` callback on the `exampleApp.Services` module
4. The `run` callback on the `exampleApp` module

AngularJS does something clever, which is to ensure that modules on which there are dependencies have their callbacks invoked first. You can see this in the way that the callbacks for the `exampleApp.Services` module are made before those for the main `exampleApp` module. This allows modules to configure themselves before they are used to resolve module dependencies. If you run the example, you will see JavaScript console output like the following:

```
Services module config: (no time)
Main module config: 16:57:28
```

```
Services module run: 16:57:28
Main module run: 16:57:28
```

I am able to use the `startTime` constant in three of the four callbacks, but I can't use in the `config` callback for the `exampleApp.Services` module because the module dependencies have yet to be resolved. At the moment the `config` callback is invoked, the `startTime` constant is unavailable.

Summary

In this chapter, I explained the basic structure of an AngularJS application from the perspective of the module. I demonstrated how to create modules; how to use them to create key building blocks like controllers, services, and filters; and how these building blocks can be used to organize the code in an application and respond to two key moments in the application life cycle. As I explained at the start of the chapter, the information I have presented here is intended to give you somewhere to refer to in order to put individual features described in the following chapters in a broader context and to point you to where you can find more details. In the next chapter, I start digging into the details, beginning with the built-in directives.