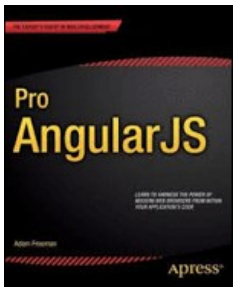

Pro AngularJS

by Adam Freeman

Chapter 10: Using Binding and Template Directives

Overview

In the previous chapter, I briefly described the range of components that can be used to create an AngularJS application. You may have found the variety of these components overwhelming and, even with the examples, struggled to understand what they are all for. Don't worry—as I explained at the start of that chapter, those descriptions and examples are intended to provide context for the detailed information that follows, starting with this chapter, in which I describe *directives*.

Directives are the most powerful AngularJS feature; they allow you to extend HTML to create the foundation for rich and complex web applications in a way that is naturally expressive. AngularJS includes a wide range of built-in directives, and you can get a surprising amount done by ignoring every other aspect of AngularJS and just relying on them. There are a lot of built-in directives to describe, which I start to do in this chapter and continue in Chapter 11 and Chapter 12. You can also create your own custom directives, and I describe the process for doing this in Chapters 15-17, after describing some of the features that are required for writing custom directives.

The directives I describe in this chapter are the ones you will use most often at the start of a new AngularJS project, but they are also the most complex and can be used in several ways. The directives I describe in the chapters that follow are simpler, so, once again, don't worry if you don't take in all of the details in one go. Table 10-1 summarizes this chapter.

Table 10-1: Chapter Summary

Problem	Solution	Listing
Create a one-way binding.	Define properties on the controller `$scope` and use the `ng-bind` or `ng-bind-template` directive or inline expressions (denoted by the `{{` and `}}` characters).	1-2
Prevent AngularJS from processing inline binding expressions.	Use the `ng-non-bindable` directive.	2
Create two-way data bindings.	Use the `ng-model` directive.	3
Generate repeated elements.	Use the `ng-repeat` directive.	4-6
Get context information about the current object in an `ng-repeat` directive.	Use the built-in variables provided by the `ng-repeat` directive, such as `$first` or `$last`.	7-9
Repeat multiple top-level attributes.	Use the `ng-repeat-start` and `ng-repeat-end` directives.	10
Load a partial view.	Use the `ng-include` directive.	11-16
Conditionally display elements.	Use the `ng-switch` directive.	17
Hide inline template expressions while AngularJS is processing content.	Use the `ng-cloak` directive.	18

Why and When to Use Directives

Directives are the signature feature of AngularJS, setting the overall style of AngularJS development and the shape of an AngularJS application. Other JavaScript libraries—including the much-loved jQuery—treat the elements in an HTML document as a problem to be overcome, requiring manipulation and correction before they can be used to create a web application.

The AngularJS approach is different: You create AngularJS web apps by embracing and enhancing HTML and treating it not as a problem but a *foundation* on which to build application features. It can take a little while to get used to the way that directives work—especially when you start to create your own custom HTML elements, a process I describe in Chapter 16—but it becomes second nature, and the result is a pleasing mix of standard HTML mixed with custom elements and attributes.

AngularJS comes with more than 50 built-in directives that provide access to core features that are useful in almost every web application including data binding, form validation, template generation, event handling, and manipulating HTML elements. And, as I already mentioned, you use custom directives to apply your application's capabilities. Table 10-2 summarizes why and when to use directives in an AngularJS application.

Table 10-2: Why and When to Use Directives

Why	When
Directives expose core AngularJS functionality such as event handling, form validation, and templates. You use custom directives to apply your application features to views.	Directives are used throughout an AngularJS application.

Preparing the Example Project

To prepare for this chapter, I deleted the contents of the `angularjs` web server folder and installed the `angular.js`, `bootstrap.css`, and `bootstrap-theme.css` files, as described in Chapter 1. I then created a file called `directives.html`, which you can see in Listing 10-1.

Listing 10-1: The Contents of the directives.html File

```
<!DOCTYPE html>
<html ng-app="exampleApp">
<head>
    <title>Directives</title>
    <script src="angular.js"></script>
    <link href="bootstrap.css" rel="stylesheet" />
    <link href="bootstrap-theme.css" rel="stylesheet" />
    <script>
        angular.module("exampleApp", [])
            .controller("defaultCtrl", function ($scope) {
                $scope.todos = [
                    { action: "Get groceries", complete: false },
                    { action: "Call plumber", complete: false },
                    { action: "Buy running shoes", complete: true },
                    { action: "Buy flowers", complete: false },
                    { action: "Call family", complete: false }];
            });
    </script>
</head>
<body>
    <div id="todoPanel" class="panel" ng-controller="defaultCtrl">
        <h3 class="panel-header">To Do List</h3>
        Data items will go here...
    </div>
</body>
</html>
```

This is a skeletal outline for the classic to-do list application (one of the reasons that so many web app examples are based on to-do lists is because lists of data objects are perfect for demonstrating template techniques).

You will recognize some of the AngularJS components that I described in Chapter 9. I created a module called `exampleApp` using the `angular.module` method and then used the fluent API to define a controller called `defaultCtrl`. The controller uses the `$scope` service to add some data items to the data model, and the module and the controller are applied to HTML elements with the `ng-app` and `ng-controller` directives. You can see how the initial content in the `directives.html` file is displayed by the browser in Figure 10-1.

Figure 10-1: The initial contents of the directives.html file

Tip You can treat the content of Listing 10-1 as a black box for the moment, get a brief description about each component from Chapter 9, or consult the chapters later in the book where I describe these building blocks in detail.

Using the Data Binding Directives

The first category of built-in directives is responsible for performing *data binding*, which is one of the features that elevates AngularJS from a template package into a full-fledged application development framework. Data binding uses values from the model and inserts them into the HTML document. Table 10-3 describes the directives in this category, and I demonstrate their use in the sections that follow.

Table 10-3: The Data Binding Directives

Directive	Applied As	Description
`ng-bind`	Attribute, class	Binds the `innerText` property of an HTML element.
`ng-bind-html`	Attribute, class	Creates data bindings using the `innerHTML` property of an HTML element. This is potentially dangerous because it means that the browser will interpret the content as HTML, rather than content. See Chapter 19 for details of how to use this directive and the service that supports it.
`ng-bind-template`	Attribute, class	Similar to the `ng-bind` directive but allows for multiple template expressions to be specified in the attribute value.
`ng-model`	Attribute, class	Creates a two-way data binding.
`ng-non-bindable`	Attribute, class	Declares a region of content for which data binding will not be performed.

Data binding is incredibly important in AngularJS development, and you will rarely encounter any substantial fragment of HTML in an AngularJS application that doesn't have some kind of data binding applied to it, and as you'll learn in the next section, the functionality provided by the `ng-bind` directive is so central to AngularJS that it has an alternative notation so you can create data bindings more easily.

Applying Directives

Table 10-3 contains an Applied As column that tells you how you can use each directive. All of the data binding directives can be applied as an attribute or as a class. Later in this chsapter, I describe a directive that can be applied as a custom HTML element.

The way you apply a directive is generally a matter of style preferences combined with a consideration for your development toolset. I generally prefer to apply directives as attributes, as follows:

```
...
There are <span ng-bind="todos.length" ></span> items
...
```

The directive is specified as the attribute name, `ng-bind` in this case, and the configuration for the directive is set as the attribute value. This fragment comes from Listing 10-2 and sets up a one-way data binding on the `todos.length` property, which I explain in the following section.

Some developers don't like the attribute approach, and—surprisingly often—attributes cause problems in development tool chains. Some JavaScript libraries make assumptions about attribute names, and some restrictive revision control systems won't let HTML content be committed with nonstandard attributes. (I encounter this most often in large corporations where the revision control system is managed by a central group that lags far behind the needs of the development teams it supports.) If you can't or won't use custom attributes, then you can configure directives using the standard `class` attribute, as follows:

```
...
There are <span class="ng-bind: todos.length" ></span> items
...
```

The value of the `class` attribute is the name of the directive, followed by a colon, followed by the configuration for the directive. This statement has the same effect as the last one: It creates a one-way data binding on the `todos.length`

property. Some directives can be applied as custom AngularJS elements. You can see an example of this in the "Working with Partial Views" section when I demonstrate the `ng-include` directive.

Not all directives can be applied in every way; most of them can be applied as attributes or classes, but only some can be applied as custom elements. The information in the tables I provide for each category of directive explains how each can be used. I explain how to create custom directives in Chapter 16, and you can specify how you want new directives to be applied as part of this process.

Note that older versions of Internet Explorer don't support custom HTML elements by default. See http://docs.angularjs.org/guide/ie for information and workarounds.

Performing One-Way Bindings (and Preventing Them)

AngularJS supports two kinds of data binding. The first, *one-way* binding, means a value is taken from the data model and inserted into an HTML element. AngularJS bindings are *live*, which means that when the value associated with the binding is changed in the data model, the HTML element will be updated to display the new value.

The `ng-bind` directive is responsible for creating one-way data bindings, but it is rarely used directly because AngularJS will also create this kind of binding whenever it encounters the `{{` and `}}` characters in the HTML document. Listing 10-2 shows the different ways you can create one-way data bindings.

Listing 10-2: Creating One-Way Data Bindings in the directives.html File

```
<!DOCTYPE html>
<html ng-app="exampleApp">
<head>
    <title>Directives</title>
    <script src="angular.js"></script>
    <link href="bootstrap.css" rel="stylesheet" />
    <link href="bootstrap-theme.css" rel="stylesheet" />
    <script>
        angular.module("exampleApp", [])
            .controller("defaultCtrl", function ($scope) {
                $scope.todos = [
                    { action: "Get groceries", complete: false },
                    { action: "Call plumber", complete: false },
                    { action: "Buy running shoes", complete: true },
                    { action: "Buy flowers", complete: false },
                    { action: "Call family", complete: false }];
            });
    </script>
</head>
<body>
    <div id="todoPanel" class="panel" ng-controller="defaultCtrl">
        <h3 class="panel-header">To Do List</h3>

        <div>There are {{todos.length}} items</div>

        <div>
            There are <span ng-bind="todos.length"></span> items
        </div>
        <div ng-bind-template=
             "First: {{todos[0].action}}. Second: {{todos[1].action}}">
        </div>

        <div ng-non-bindable>
            AngularJS uses {{ and }} characters for templates
        </div>
    </div>
</body>
</html>
```

You can see the result of navigating to the `directives.html` file with the browser in Figure 10-2. The effect isn't the most visually striking, but the directives in the example are doing some interesting things.

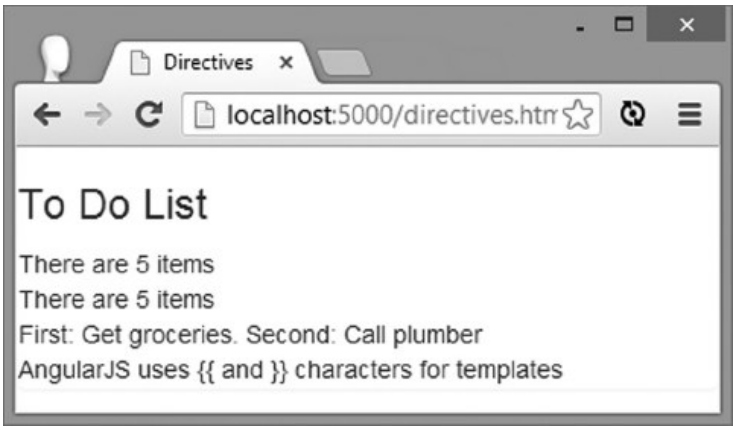

Figure 10-2: Creating one-way data bindings

Tip AngularJS isn't the only JavaScript package that uses the `{{` and `}}` characters, which can be a problem if you are trying to make multiple libraries work together. AngularJS allows you to change the characters used for inline bindings; I explain the process in Chapter 19.

The first two data bindings in this example are equivalent. I have used `{{` and `}}` to denote a one-way binding for the number of items in the `$scope.todos` collection:

```
...
<div>There are {{todos.length}} items</div>
...
```

This is the most natural and expressive way of creating data bindings: The bindings are easy to read and fit naturally into the content of HTML elements. The second data binding uses the `ng-bind` directive, which has the same effect but requires an additional element:

```
...
There are <span ng-bind="todos.length"></span> items
...
```

The `ng-bind` directive replaces the content of the element that it is applied to, which means I have to add a `span` element to create the effect I want. I don't use the `ng-bind` directive in my own projects; I prefer the inline bindings.

The `ng-bind` directive *does* allow you to hide your template markup when the HTML content is shown to the user before it is processed by AngularJS (because browsers don't display attribute values to users), but this is rarely a problem and is addressed by the `ng-cloak` directive, which I describe later in this chapter.

Caution You can create bindings only for data values that are added to the `$scope` object by the controller. I explain how `$scope` works in Chapter 13.

Aside from being a little awkward to use, the `ng-bind` directive is limited to being able to process a single data binding expression. If you need to create multiple data bindings, then you should use the `ng-bind-template` directive, which is more flexible, as follows:

```
...
<div ng-bind-template="First: {{todos[0].action}}. Second: {{todos[1].action}}" ></div>
...
```

The value I specified for the directive contains two data bindings, which the `ng-bind` directive would not be able to process. I have never used this directive in a real project, and I suspect I never will. I include this directive here for completeness only.

Preventing Inline Data Binding

The drawback of the inline bindings is that AngularJS will find and process every set of `{{` and `}}` characters in your content. This can be a problem, especially if you are mixing and matching JavaScript toolkits and want to use some other template system on a region of HTML (or if you just want to use double-brace characters in your text). The solution is to use the `ng-non-bindable` directive, which prevents AngularJS from processing inline bindings:

```
...
```

```
<div ng-non-bindable >
    AngularJS uses {{ and }} characters for templates
</div>
...
```

If I had not applied the directive, AngularJS would have processed the contents of the `div` element and then tried to bind to a model property called `and`. AngularJS doesn't complain when it is asked to bind to a nonexisting model property because it assumes it will be created later (as I explain when I describe the `ng-model` directive later in this chapter). Instead, it inserts no content at all, which means that instead of the output I wanted:

```
AngularJS uses {{ and }} characters for templates
```

I would instead produce this:

```
AngularJS uses characters for templates
```

Creating Two-Way Data Bindings

Two-way data bindings track changes in both directions, allowing elements that gather data from the user to modify the state of the application. Two-way bindings are created with the `ng-model` directive, and as Listing 10-3 demonstrates, a single data model property can be used for both one- and two-way bindings.

Listing 10-3: Creating Two-Way Bindings in the directives.html File

```
...
<body>
    <div id="todoPanel" class="panel" ng-controller="defaultCtrl">
        <h3 class="panel-header">To Do List</h3>
        <div class="well">
            <div>The first item is: {{todos[0].action}}</div>
        </div>

        <div class="form-group well">
            <label for="firstItem">Set First Item:</label>
            <input name="firstItem" class="form-control" ng-model="todos[0].action" />
        </div>
    </div>
</body>
...
```

There are two data bindings in this listing, both of which are applied to the `action` property of the first object in the `todos` data array (which I set up using the `$scope` object in the controller and reference in bindings as `todos[0].action)`. The first binding is an inline one-way binding that simply displays the value of the data property, just as I did in the previous example. The second binding is applied via the `input` element and is a two-way binding:

```
...
<input name="firstItem" class="form-control" ng-model="todos[0].action" />
...
```

Two-way bindings can be applied only to elements that allow the user to provide a data value, which means the `input`, `textarea`, and `select` elements. The `ng-model` directive sets the content of the element it is applied to and then responds to changes that the user makes by updating the data model.

> **Tip** The `ng-model` directive provides additional features for working with HTML forms and even for creating custom form directives. See Chapters 12 and 17 for details.

Changes to data model properties are disseminated to all of the relevant bindings, ensuring that the application is kept in sync. For my example, this means that changes to the `input` element update the data model, which then causes the update to be shown in the inline one-way binding.

To see the effect, use the browser to navigate to the `directives.html` document and edit the text in the `input` element; you will see that the one-way binding is kept in sync with the contents of the `input` element, all through the magic of the two-way binding. The best way to experience this effect is by re-creating the example and experiencing it first hand, but

you can get a sense of what happens in Figure 10-3.

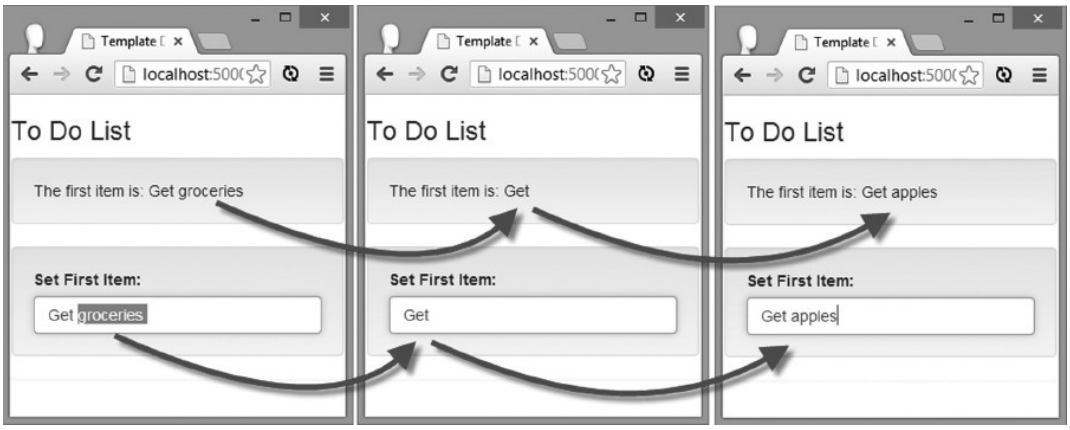

Figure 10-3: Using two-way data bindings

Note OK, two-way bindings are not really magic. AngularJS uses standard JavaScript events to receive notifications from the `input` element when its content changes and propagates these changes via the `$scope` service. You can see the event handler that AngularJS sets up through the F12 developer tools, and I explain how the `$scope` service detects and disseminates changes in Chapter 13.

Tip In this example, I used properties that had been explicitly added to the data model through the `$scope` service in the controller factory method. One nice feature of data binding is that AngularJS will dynamically create model properties as they are needed, which means you don't have to laboriously define all of the properties you use to glue views together. You can see further examples of this technique in Chapter 12 when I describe the AngularJS support for working with form elements.

Using the Template Directives

Data bindings are the core feature of AngularJS views, but on their own they are pretty limited. Web applications—or *any* kind of application for that matter—tend to operate on collections of similar data objects and vary the view they present to the user based on different data values.

Fortunately, AngularJS includes a set of directives that can be used to generate HTML elements using templates, making it easy to work with data collections and to add basic logic to a template that responds to the state of the data. I have summarized the template directives in Table 10-4.

Table 10-4: The Template Directives

Directive	Applied As	Description
`ng-cloak`	Attribute, class	Applies a CSS style that hides inline binding expressions, which can be briefly visible when the document first loads
`ng-include`	Element, attribute, class	Loads, processes, and inserts a fragment of HTML into the Document Object Model
`ng-repeat`	Attribute, class	Generates new copies of a single element and its contents for each object in an array or property on an object
`ng-repeat-start`	Attribute, class	Denotes the start of a repeating section with multiple top-level elements
`ng-repeat-end`	Attribute, class	Denotes the end of a repeating section with multiple top-level elements
`ng-switch`	Element, attribute	Changes the elements in the Document Object Model based on the value of data bindings

These directives help you put simple logic into views without having to write any JavaScript code. As I explained in Chapter 3, the logic in views should be restricted to generating the content required to display data, and these directives all fit into that definition.

Generating Elements Repeatedly

One of the most common tasks in any view is to generate the same content for each item in a collection of data. In AngularJS this is done with the `ng-repeat` directive, which is applied to the element that should be duplicated. Listing 10-4 contains a simple example of using the `ng-repeat` directive.

Listing 10-4: Using the ng-repeat Directive in the directives.html File

```
...
<body>
    <div id="todoPanel" class="panel" ng-controller="defaultCtrl">
        <h3 class="panel-header">To Do List</h3>

        <table class="table">
            <thead>
                <tr>
                    <th>Action</th>
                    <th>Done</th>
                </tr>
            </thead>
            <tbody>
                <tr ng-repeat="item in todos">
                    <td>{{item.action}}</td>
                    <td>{{item.complete}}</td>
                </tr>
            </tbody>
        </table>
    </div>
</body>
...
```

This is the simplest and most common way of using the `ng-repeat` directive: to generate rows for a `table` element using a collection of objects. There are two parts to using the `ng-repeat` directive. The first is to specify the source of the data objects and the name by which you want to refer to the object that is being processed from within the template:

```
...
<tr ng-repeat="item in todos" >
...
```

The basic format of the value for the `ng-repeat` directive attribute is `<variable> in <source>`, where `source` is an object or array defined by the controller `$scope`, in this example the `todos` array. The directive iterates through the objects in the array, creates a new instance of the element and its content, and then processes the templates it contains. The `<variable>` name assigned in the directive attribute value can be used to refer to the current data object. In my example, I used the variable name `item`:

```
...
<tr ng-repeat="item in todos">
    <td>{{item.action}} </td>
    <td>{{item.complete}} </td>
</tr>
...
```

In my example, I generate a `tr` element that contains `td` elements that, in turn, contain inline data bindings that refer to the `action` and `complete` properties of the current object. If you navigate to the `directives.html` file in the browser, AngularJS will process the directive and generate the following HTML elements:

```
...
<tbody>
    <!-- ngRepeat: item in todos -->
    <tr ng-repeat="item in todos" class="ng-scope">
        <td class="ng-binding">Get groceries</td>
        <td class="ng-binding">false</td>
    </tr>
    <tr ng-repeat="item in todos" class="ng-scope">
        <td class="ng-binding">Call plumber</td>
        <td class="ng-binding">false</td>
    </tr>
    <tr ng-repeat="item in todos" class="ng-scope">
        <td class="ng-binding">Buy running shoes</td>
```

```
        <td class="ng-binding">true</td>
    </tr>
    <tr ng-repeat="item in todos" class="ng-scope">
        <td class="ng-binding">Buy flowers</td>
        <td class="ng-binding">false</td>
    </tr>
    <tr ng-repeat="item in todos" class="ng-scope">
        <td class="ng-binding">Call family</td>
        <td class="ng-binding">false</td>
    </tr>
</tbody>
...
```

You will see that AngularJS has generated a comment to make it easier to see which directive generated the elements and has added the generated elements to some classes (these are used internally by AngularJS). Figure 10-4 illustrates the effect of this HTML in the browser window.

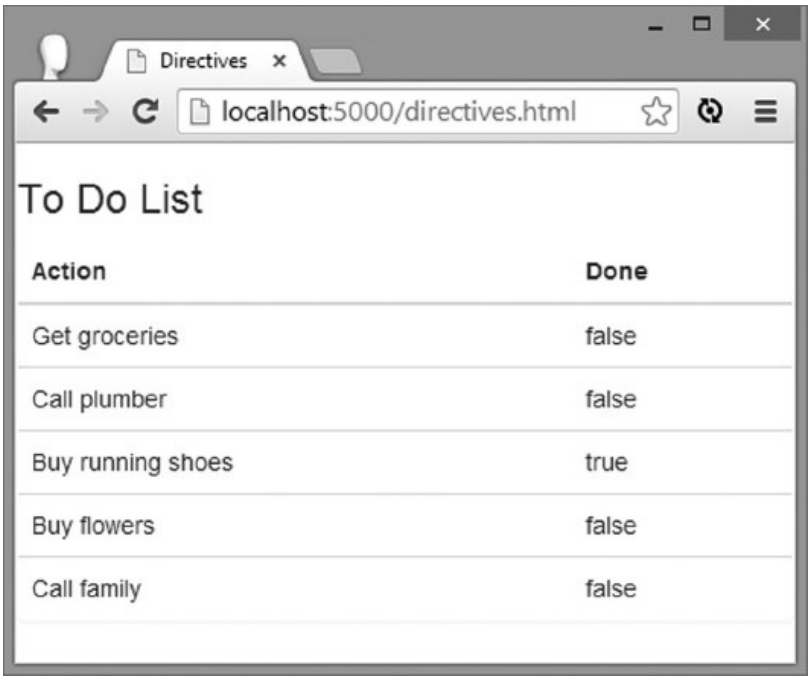

Figure 10-4: Generating HTML elements with the ng-repeat directive

Tip You will need to use your browser's F12 developer tools to see these elements, rather than the View HTML or View Page Source menu. Most browsers will display the HTML they receive from the server only through the View Page Source menu, which won't contain the elements that AngularJS generates from templates. The developer tools show you the live Document Object Model, which reflects the changes that AngularJS makes.

Repeating for Object Properties

The previous example used the `ng-repeat` directive to enumerate the objects in an array, but you can also enumerate the properties of an object. The `ng-repeat` directive can also be nested, and you can see how I have combined these features to simplify my template in Listing 10-5.

Listing 10-5: Repeating Object Properties and Nesting the ng-repeat Directive in the directives.html File

```
...
<table class="table">
    <thead>
        <tr>
            <th>Action</th>
            <th>Done</th>
        </tr>
    </thead>
    <tbody>
```

```
        <tr ng-repeat="item in todos">
            <td ng-repeat="prop in item">{{prop}}</td>
        </tr>
    </tbody>
</table>
...
```

The outer `ng-repeat` directive generates a `tr` element for each object in the `todos` array, and each object is assigned to the `item` variable. The inner `ng-repeat` directive generates a `td` element for each property of the `item` object and assigns the value of the property to the `prop` variable. Finally, the `prop` variable is used for a one-way data binding as the contents of the `td` element. This produces the same result as the previous example but will adapt fluidly to generate `td` elements for any new properties that are defined on the data objects. This is a simple example, but it gives a sense of the flexibility available when working with AngularJS templates.

Working with Data Object Keys

There is an alternative syntax for the `ng-repeat` directive configuration that allows you to receive a key with each property or data object that is processed. You can see an example of this syntax in Listing 10-6.

Listing 10-6: Receiving a Key Along with a Data Value in the directives.html File

```
...
<tr ng-repeat="item in todos">
    <td ng-repeat="(key, value) in item">
        {{key}}={{value}}
    </td>
</tr>
...
```

Instead of a single variable name, I have specified two names separated by a comma within parentheses. For each object or property that the `ng-repeat` directive enumerates, the second variable will be assigned the data object or property value. The way the first variable is used depends on the source of the data. For objects, the key is the current property name, and for collections the key is the position of the current object. I am enumerating the properties of an object in the listing, so the value of `key` will be the property name, and `value` will be assigned the property value. Here is an example of the HTML element that this `ng-repeat` directive will generate, with the values inserted by the data bindings to the `key` and `value` variables emphasized:

```
...
<tr ng-repeat="item in todos" class="ng-scope">
    <!-- ngRepeat: (key, value) in item -->
    <td ng-repeat="(key, value) in item" class="ng-scope ng-binding">
        action=Get groceries
    </td>
    <td ng-repeat="(key, value) in item" class="ng-scope ng-binding">
        complete=false
    </td>
</tr>
...
```

Working with the Built-in Variables

The `ng-repeat` directive assigns the current object or property to the variable you specify, but there is also a set of built-in variables that provide context for the data being processed. You can see an example of one of them applied in Listing 10-7.

Listing 10-7: Using a Built-in ng-repeat Variable in the directives.html File

```
...
<table class="table">
    <thead>
        <tr>
            <th>#</th>
            <th>Action</th>
            <th>Done</th>
```

```
        </tr>
    </thead>
    <tr ng-repeat="item in todos">
        <td>{{$index + 1}}</td>
        <td ng-repeat="prop in item">
            {{prop}}
        </td>
    </tr>
</table>
...
```

I added a new column to the table that contains the to-do items and used the `$index` variable, which is provided by the `ng-repeat` directive, to display the position of each item in the array. Since JavaScript collection indexes are zero-based, I simply add one to `$index`, relying on the fact that AngularJS will evaluate JavaScript expressions in data bindings. You can see the effect in Figure 10-5.

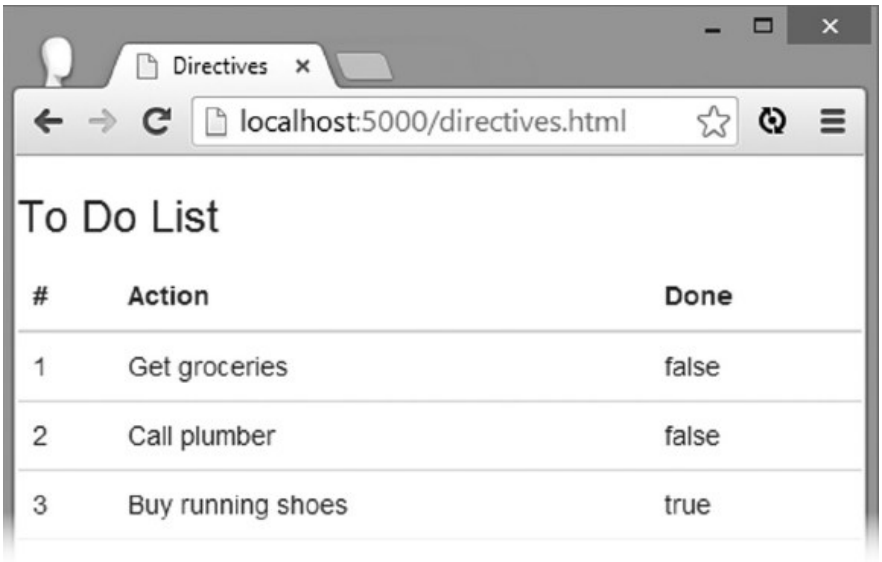

Figure 10-5: Using the built-in variables provided by the ng-repeat directive

The `$index` variable is the one that I find most useful, but I have described the complete set in Table 10-5.

Table 10-5: The Built-in ng-repeat Variables

Variable	Description
`$index`	Returns the position of the current object or property
`$first`	Returns `true` if the current object is the first in the collection
`$middle`	Returns `true` if the current object is neither the first nor last in the collection
`$last`	Returns `true` if the current object is the last in the collection
`$even`	Returns `true` for the even-numbered objects in a collection
`$odd`	Returns `true` for the odd-numbered objects in a collection

You can use these variables to control the elements you generate. A typical use of these variables is to create the classic striping effect for table elements, which I have shown in Listing 10-8.

Listing 10-8: Creating a Striped Table Using the ng-repeat Directive in the directives.html File

```
<!DOCTYPE html>
<html ng-app="exampleApp">
<head>
    <title>Directives</title>
    <script src="angular.js"></script>
```

```
    <link href="bootstrap.css" rel="stylesheet" />
    <link href="bootstrap-theme.css" rel="stylesheet" />
    <script>
        angular.module("exampleApp", [])
            .controller("defaultCtrl", function ($scope) {
                $scope.todos = [
                    { action: "Get groceries", complete: false },
                    { action: "Call plumber", complete: false },
                    { action: "Buy running shoes", complete: true },
                    { action: "Buy flowers", complete: false },
                    { action: "Call family", complete: false }];
            });
    </script>
    <style>
        .odd { background-color: lightcoral}
        .even { background-color: lavenderblush}
    </style>
</head>
<body>
    <div id="todoPanel" class="panel" ng-controller="defaultCtrl">
        <h3 class="panel-header">To Do List</h3>
        <table class="table">
            <thead>
                <tr>
                    <th>#</th>
                    <th>Action</th>
                    <th>Done</th>
                </tr>
            </thead>
            <tr ng-repeat="item in todos" ng-class="$odd ? 'odd' : 'even'">
                <td>{{$index + 1}}</td>
                <td ng-repeat="prop in item">{{prop}}</td>
            </tr>
        </table>
    </div>
</body>
</html>
```

I have used the `ng-class` directive, which sets the `class` attribute of an element using a data binding. I use a JavaScript ternary expression to assign elements to either the `odd` or `even` class based on the value of the `$odd` variable. You can see the result in Figure 10-6.

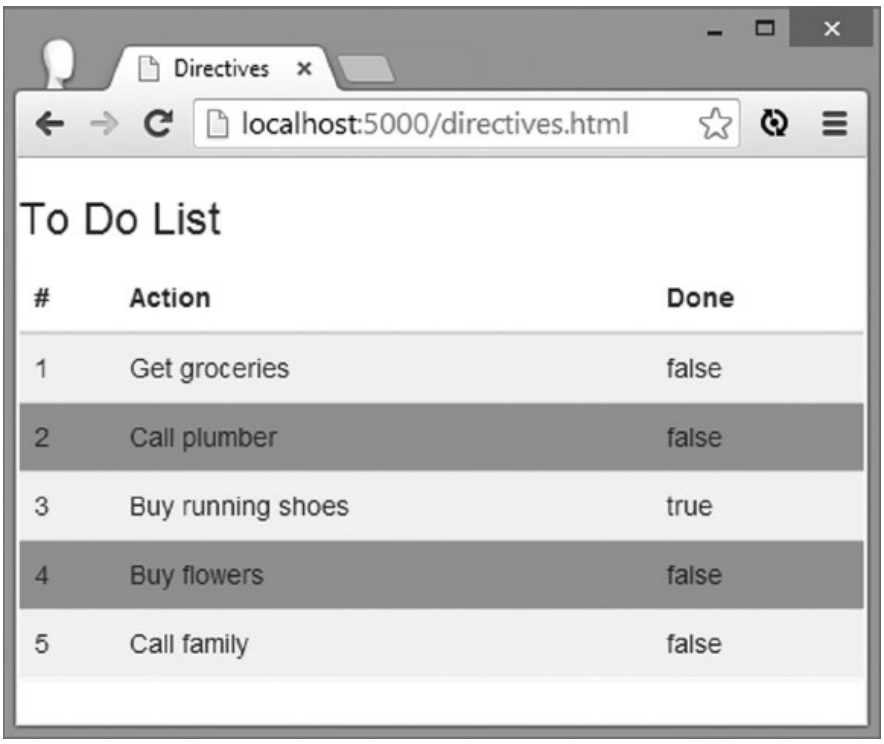

Figure 10-6: Varying content styling based on the ng-repeat variables

Tip I explain the `ng-class` directive in Chapter 11, along with two related directives that are often used with `ng-repeat`: `ng-class-even` and `ng-class-odd`. As their names suggest, these directives set the value of the `class` attribute based on the `$odd` and `$even` variables defined by the `ng-repeat` directive.

Although this is the standard demonstration for the `ng-repeat` variables, most CSS frameworks can stripe tables, and this includes Bootstrap, as I demonstrated in Chapter 4. The real power of these variables comes when they are used in conjunction with other, more complex directives. Listing 10-9 provides a demonstration.

Listing 10-9: A More Complex ng-repeat Variable Example in the directives.html File

```
...
<table class="table">
    <thead>
        <tr>
            <th>#</th>
            <th>Action</th>
            <th>Done</th>
        </tr>
    </thead>
    <tr ng-repeat="item in todos" ng-class="$odd ? 'odd' : 'even'">
        <td>{{$index + 1}}</td>
        <td>{{item.action}}</td>
        <td><span ng-if="$first || $last">{{item.complete}}</span></td>
    </tr>
</table>
...
```

I have used the `ng-if` directive in this example, which I describe properly in Chapter 11. For now, it is enough to know that the `ng-if` directive will remove the element it is applied to if the expression it evaluates is `false`. I used this directive to control the presence of a `span` element in the Done column of the table, ensuring that it is displayed for only the first and last items.

Repeating Multiple Top-Level Elements

The `ng-repeat` directive repeats a single top-level element and its contents for each object or property that it processes. There are times, however, when you need to repeat *multiple* top-level elements for each data object. I encounter this

problem most often when I need to generate multiple table rows for each data item that I am processing—something that is difficult to achieve with `ng-repeat` because no intermediate elements are allowed between `tr` elements and their parents. To address this problem, I can use the `ng-repeat-start` and `ng-repeat-end` directives, as shown in Listing 10-10.

Listing 10-10: Using the ng-repeat-start and ng-repeat-end Directives in the directives.html File

```
...
<table class="table">
    <tbody>
        <tr ng-repeat-start="item in todos">
            <td>This is item {{$index}}</td>
        </tr>
        <tr>
            <td>The action is: {{item.action}}</td>
        </tr>
        <tr ng-repeat-end>
            <td>Item {{$index}} is {{$item.complete? '' : "not "}} complete</td>
        </tr>
    </tbody>
</table>
...
```

The `ng-repeat-start` directive is configured just like `ng-repeat`, but it repeats all of the top-level elements (and their contents) until (but including) the element to which the `ng-repeat-end` attribute has been applied. In this example, it means I am able to generate three `tr` elements for each object in the `todos` array.

Working with Partial Views

The `ng-include` directive retrieves a fragment of HTML content from the server, compiles it to process any directives that it might contain, and adds it to the Document Object Model. These fragments are known as *partial views*. To demonstrate how this works, I have added an HTML file called `table.html` to the web server `angularjs` folder. You can see the contents of the new file in Listing 10-11.

Listing 10-11: The Contents of the table.html File

```
<table class="table">
    <thead>
        <tr>
            <th>#</th>
            <th>Action</th>
            <th>Done</th>
        </tr>
    </thead>
    <tr ng-repeat="item in todos" ng-class="$odd ? 'odd' : 'even'">
        <td>{{$index + 1}}</td>
        <td ng-repeat="prop in item">{{prop}}</td>
    </tr>
</table>
```

This file contains the fragment of HTML that defines the `table` element from earlier examples, complete with data bindings and directives—a simple partial view. In Listing 10-12, you can see how I can use the `ng-include` directive to load, process, and insert the `table.html` file into the main document.

Listing 10-12: Using the ng-include Directive in the directives.html File

```
<!DOCTYPE html>
<html ng-app="exampleApp">
<head>
    <title>Directives</title>
    <script src="angular.js"></script>
    <link href="bootstrap.css" rel="stylesheet" />
    <link href="bootstrap-theme.css" rel="stylesheet" />
    <script>
        angular.module("exampleApp", [])
```

```
            .controller("defaultCtrl", function ($scope) {
                $scope.todos = [
                    { action: "Get groceries", complete: false },
                    { action: "Call plumber", complete: false },
                    { action: "Buy running shoes", complete: true },
                    { action: "Buy flowers", complete: false },
                    { action: "Call family", complete: false }];
            });
    </script>
</head>
<body>
    <div id="todoPanel" class="panel" ng-controller="defaultCtrl">
        <h3 class="panel-header">To Do List</h3>
        <ng-include src="'table.html'"></ng-include>
    </div>
</body>
</html>
```

This is the first of the built-in directives that can be used as an HTML element as well as an attribute or class. As the listing illustrates, the name of the directive is used as the element tag name, like this:

```
...
<ng-include src="'table.html'"></ng-include>
...
```

The custom element is used just like any of the standard ones. The `ng-include` directive supports three configuration parameters, and when used like an element, they are applied as attributes.

Caution Don't try to use apply the `ng-include` directive as a void element (in other words, `<ng-include src="'table.html'" />`). The content that follows the `ng-include` element will be removed from the DOM. You must always specify open and close tags, as I have shown in the example.

You can see the first of these parameters in the listing: The `src` attribute sets the location of the partial view file I want loaded, processed, and added to the document. For this example, I have specified the `table.html` file. When AngularJS processes the `directive.html` file, it encounters the `ng-include` directive and automatically makes an Ajax request for the `table.html` file, processes the file contents, and adds them to the document. I have described the three configuration parameters in Table 10-6, although it is `src` that interests us in this chapter.

Table 10-6: The Configuration Parameters of the ng-include Directive

Name	Description
`src`	Specifies the URL of the content to load
`onload`	Specifies an expression to be evaluated when the content is loaded
`autoscroll`	Specifies whether AngularJS should scroll the viewport when the content is loaded

The contents of files loaded by the `ng-include` directive are processed as though they were defined in situ, meaning you have access to the data model and behaviors defined by the controller and, if you use the `ng-include` directive within the `ng-repeat` directive, the special variables such as `$index` and `$first` that I described earlier in this chapter.

Selecting Partial Views Dynamically

My previous example demonstrated how the `ng-include` directive can be used to break a view into multiple partial view files. This is, in and of itself, a useful feature, and it allows you to create reusable partial views that can be applied throughout an application to avoid duplication and ensure consistent presentation of data.

That's all well and good, but you may have noticed something a little odd in the way that I specified which file the `ng-include` directive should request from the server:

```
...
<ng-include src=" 'table.html' "></ng-include>
...
```

I specified the `table.html` file as a string literal, denoted by the single-quote characters. I had to do this because the `src` attribute is evaluated as a JavaScript expression, and to statically define a file, I have to surround the file name with single

quotes.

The real power of the `ng-include` directive comes from the way that the `src` setting is evaluated. To demonstrate how this works, I have created a new partial view file called `list.html` in the web server `angularjs` folder. You can see the content of the new file in Listing 10-13.

Listing 10-13: The Contents of the list.html File

```
<ol>
    <li ng-repeat="item in todos">
        {{item.action}}
        <span ng-if="item.complete"> (Done)</span>
    </li>
</ol>
```

This file contains a fragment of new markup that I have not used in previous examples. I use an `ol` element to denote an ordered list and use the `ng-repeat` directive on an `li` element to generate list items for each to-do. I use the `ng-if` directive, which I applied in a previous example (and explain fully in Chapter 11) to control the inclusion of a `span` element for those to-do items that are complete. Now that I have two partial views that can display the to-do items, I can use the `ng-include` directive to switch between them, as shown in Listing 10-14.

Listing 10-14: Using the ng-include Directive to Process Fragments Dynamically in the directives.html File

```
<!DOCTYPE html>
<html ng-app="exampleApp">
<head>
    <title>Directives</title>
    <script src="angular.js"></script>
    <link href="bootstrap.css" rel="stylesheet" />
    <link href="bootstrap-theme.css" rel="stylesheet" />
    <script>
        angular.module("exampleApp", [])
            .controller("defaultCtrl", function ($scope) {
                $scope.todos = [
                    { action: "Get groceries", complete: false },
                    { action: "Call plumber", complete: false },
                    { action: "Buy running shoes", complete: true },
                    { action: "Buy flowers", complete: false },
                    { action: "Call family", complete: false }];
                    $scope.viewFile = function () {
                        return $scope.showList ? "list.html" : "table.html";
                    };
                });
        </script>
    </head>
    <body>
        <div id="todoPanel" class="panel" ng-controller="defaultCtrl">
            <h3 class="panel-header">To Do List</h3>

            <div class="well">
                <div class="checkbox">
                    <label>
                        <input type="checkbox" ng-model="showList">
                        Use the list view
                    </label>
                </div>
            </div>

            <ng-include src="viewFile()"></ng-include>

        </div>
    </body>
    </html>
```

I have defined a behavior called `viewFile` in the controller that returns the name of one of the two fragment files I created

based on the value of a variable called `showList`. If `showList` is `true`, then the `viewFile` behavior returns the name of the `list.html` file; if `showList` is `false` or `undefined`, then the behavior returns the name of the `table.html` file.

The `showList` variable is initially undefined, but I have added a check box `input` element that sets the variable when it is checked using the `ng-model` directive, which I described earlier in this chapter. The user can change the value of the `showList` variable by checking or unchecking the element.

The final link in this chain is to change the way I apply the `ng-include` directive so that the `src` attribute gets its value from the controller behavior, which I do as follows:

```
...
<ng-include src="viewFile()" ></ng-include>
...
```

The AngularJS data binding feature will keep the check box and the value of the `showList` variable synchronized, and the `ng-include` directive will change the content it loads and displays in concert with the `showList` value. You can see the effect of checking and unchecking the box in Figure 10-7.

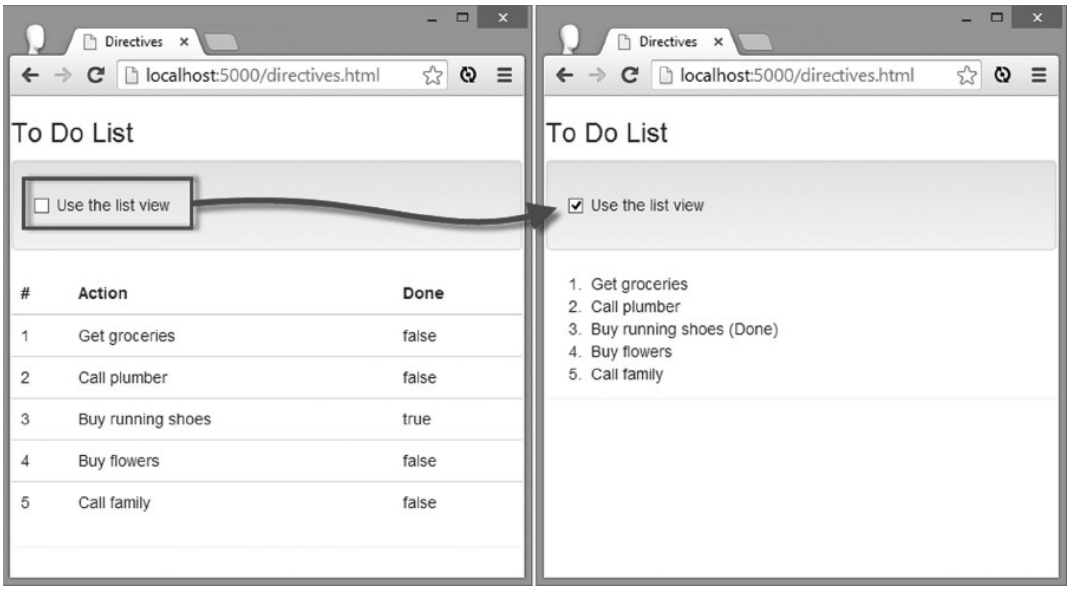

Figure 10-7: Using the ng-include directive to display content based on a model property

Using the ng-include Directive as an Attribute

Since this is the first directive I have described that can be expressed as an element, I am going to take a moment and show you how can achieve the same effect using an attribute. To start with, Listing 10-15 shows the `ng-include` directive applied as an element with the `src` and `onload` attributes set. You have seen `src` used in the previous examples. The `onload` attribute is used to specify an expression that will be evaluated when content is loaded; I have specified a call to the `reportChange` behavior that I added to the example and that writes a message to the JavaScript console reporting the name of the content file used. The `onload` attribute isn't especially interesting, but I want to use multiple configuration options to show you.

Listing 10-15: Using the ng-include Directive as an Element with Multiple Options in the directives.html File

```
<!DOCTYPE html>
<html ng-app="exampleApp">
<head>
    <title>Directives</title>
    <script src="angular.js"></script>
    <link href="bootstrap.css" rel="stylesheet" />
    <link href="bootstrap-theme.css" rel="stylesheet" />
    <script>
        angular.module("exampleApp", [])
            .controller("defaultCtrl", function ($scope) {
                $scope.todos = [
                    { action: "Get groceries", complete: false },
```

```
                    { action: "Call plumber", complete: false },
                    { action: "Buy running shoes", complete: true },
                    { action: "Buy flowers", complete: false },
                    { action: "Call family", complete: false }];
                $scope.viewFile = function () {
                    return $scope.showList ? "list.html" : "table.html";
                };

                $scope.reportChange = function () {
                    console.log("Displayed content: " + $scope.viewFile());
                }
            });
    </script>
</head>
<body>
    <div id="todoPanel" class="panel" ng-controller="defaultCtrl">
        <h3 class="panel-header">To Do List</h3>

        <div class="well">
            <div class="checkbox">
                <label>
                    <input type="checkbox" ng-model="showList">
                    Use the list view
                </label>
            </div>
        </div>

        <ng-include src="viewFile()" onload="reportChange()"></ng-include>
    </div>
</body>
</html>
```

Now, assuming I cannot use a custom element—or that I just prefer not to—I can rewrite this example to apply the `ng-include` directive as a custom attribute on a standard HTML element, as shown in Listing 10-16.

Listing 10-16: Applying the ng-include Directive as an Attribute in the directives.html File

```
...
<div ng-include="viewFile()" onload="reportChange()" ></div>
...
```

The `ng-include` attribute can be applied to any HTML element, and the value of the `src` parameter is taken from the attribute value, which is `viewFile()` in this case. The other directive configuration parameters are expressed as separate attributes, which you can see with the `onload` attribute. This application of the `ng-include` directive has *exactly* the same effect as using the custom element.

Conditionally Swapping Elements

The `ng-include` directive is excellent for managing more significant fragments of content in partial, but often you need to switch between smaller chucks of content that are already within the document—and for this, AngularJS provides the `ng-switch` directive. You can see how I have applied this directive in Listing 10-17.

Listing 10-17: Using the ng-switch Directive in the directives.html File

```
<!DOCTYPE html>
<html ng-app="exampleApp">
<head>
    <title>Directives</title>
    <script src="angular.js"></script>
    <link href="bootstrap.css" rel="stylesheet" />
    <link href="bootstrap-theme.css" rel="stylesheet" />
    <script>
        angular.module("exampleApp", [])
            .controller("defaultCtrl", function ($scope) {
```

```
                $scope.data = {};

                $scope.todos = [
                    { action: "Get groceries", complete: false },
                    { action: "Call plumber", complete: false },
                    { action: "Buy running shoes", complete: true },
                    { action: "Buy flowers", complete: false },
                    { action: "Call family", complete: false }];
            });
    </script>
</head>
<body>
    <div id="todoPanel" class="panel" ng-controller="defaultCtrl">

        <h3 class="panel-header">To Do List</h3>

        <div class="well">
            <div class="radio" ng-repeat="button in ['None', 'Table', 'List']">
                <label>
                    <input type="radio" ng-model="data.mode"
                           value="{{button}}" ng-checked="$first" />
                    {{button}}
                </label>
            </div>
        </div>

        <div ng-switch on="data.mode">
            <div ng-switch-when="Table">
                <table class="table">
                    <thead>
                        <tr><th>#</th><th>Action</th><th>Done</th></tr>
                    </thead>
                    <tr ng-repeat="item in todos" ng-class="$odd ? 'odd' : 'even'">
                        <td>{{$index + 1}}</td>
                        <td ng-repeat="prop in item">{{prop}}</td>
                    </tr>
                </table>
            </div>
            <div ng-switch-when="List">
                <ol>
                    <li ng-repeat="item in todos">
                        {{item.action}}<span ng-if="item.complete"> (Done)</span>
                    </li>
                </ol>
            </div>
            <div ng-switch-default>
                Select another option to display a layout
            </div>
        </div>

    </div>
</body>
</html>
```

I start this example by using the `ng-repeat` directive to generate a set of radio buttons that use two-way data bindings to set the value of a model property called `data.mode`. The three values defined by the radio buttons are `None, Table`, and `List` and use each to represent a layout to display the to-do items.

Tip Notice that I have defined the scope property `mode` as a property on an object called `data`. This is required because of the way that AngularJS scopes inherit from one another and how some directives—including `ng-model`— create their own scopes. I explain how this works in Chapter 13.

The rest of the example demonstrates the `ng-switch` directive, which lets me display a different set of elements for each value that the `data.mode` property will be set to. You can see the result in Figure 10-8, and I explain the different parts of the directive after the figure.

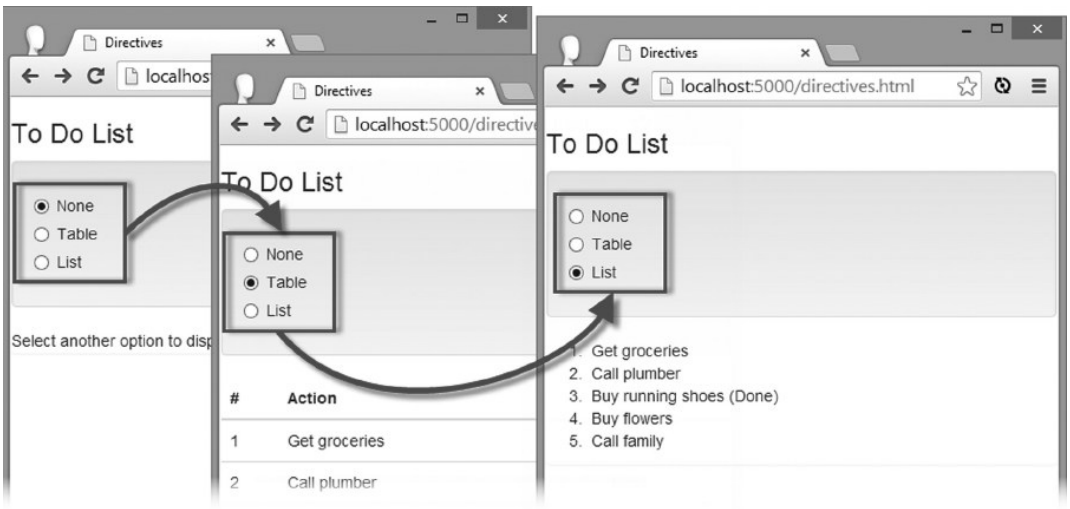

Figure 10-8: Using the ng-switch directive

Tip The `ng-switch` directive can be applied as an element, but the `ng-switch-when` and `ng-switch-default` sections have to be applied as attributes. Because of this, I tend to use `ng-switch` as an attribute as well for consistency.

The `ng-switch` directive is applied with an `on` attribute that specifies the expression that will be evaluated to decide which region of content will be displayed, as follows:

```
...
<div ng-switch on="data.mode" >
...
```

In this example, I have specified that the value of the `data.mode` model property—the one that the radio buttons manage—will be used. You then use the `ng-switch-when` directive to denote a region of content that will be associated with a specific value, like this:

```
...
<div ng-switch-when="Table" >
    <table class="table">
        <!-- elements omitted for brevity -->
    </table>
</div>
<div ng-switch-when="List" >
    <ol>
        <!-- elements omitted for brevity -->
    </ol>
</div>
...
```

AngularJS will show the element to which the `ng-switch-when` directive has been applied when the attribute value matches the expression defined by the `on` attribute. The other elements within the `ng-switch` directive block are removed. The `ng-switch-default` directive is used to specify content that should be displayed when none of the `ng-switch-when` sections matches, as follows:

```
...
<div ng-switch-default >
    Select another option to display a layout
</div>
...
```

The `ng-switch` directive responds to changes in the value of its data binding, which is why clicking a radio button in the example causes the layout to change.

Choosing Between the Ng-Include and Ng-Switch Directives

The `ng-include` and `ng-switch` directives can be used to create the same effects, and it can be difficult to figure out when to use each of them to best effect.

Use `ng-switch` when you need to alternate between smaller, simpler blocks of content and that there is a good chance the user will be shown most or all of those blocks in the normal execution of the web app. This is because you have to deliver all the content that the `ng-switch` directive needs as part of the HTML document, and that's a waste of bandwidth and loading time for content that is unlikely to be used.

The `ng-include` attribute is better suited for more complex content or content that you need to use repeatedly throughout an application. Partial views can help reduce the amount of duplication in a project when you need to include the same content in different places, but you must bear in mind that partial views are not requested until the first time they are required, and this can cause delays while the browser makes the Ajax request and receives the response from the server.

If in doubt, start with `ng-switch`. It is simpler and easier to work with, and you can always change to `ng-include` if your content gets too complex to easily manage or if you need to use the same content elsewhere in the same app.

Hiding Unprocessed Inline Template Binding Expressions

When working with complex content on slow devices, there can be a moment when the browser displays the HTML in the document while AngularJS is still parsing the HTML, processing the directives, and generally getting ready. In this interval, any inline template expressions you have defined will be visible to the user, as I have illustrated in Figure 10-9.

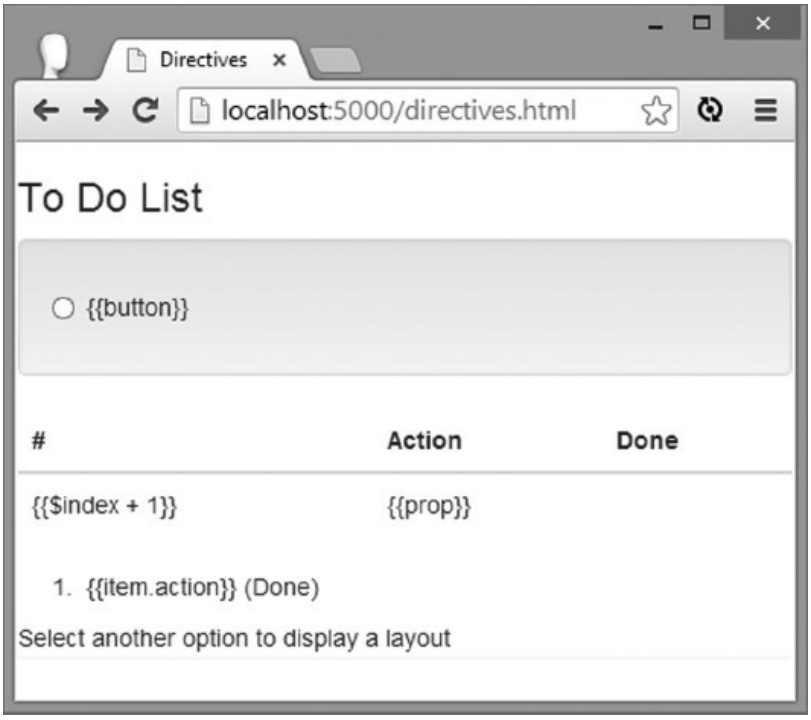

Figure 10-9: The template expressions displayed to the user while AngularJS is getting ready

Most devices have pretty good browsers these days with JavaScript implementations that are quick enough to prevent this from being a problem; in fact, I had to work pretty hard to capture the screenshot shown in the figure because desktop browsers are so fast that the situation doesn't arise.

But it does happen—especially if you are targeting older devices/browsers—and there are two ways to solve the problem. The first is to avoid using inline template expressions and stick with the `ng-bind` directive. I described this directive at the start of the chapter and made the point that it is ungainly when compared to inline expressions.

A better alternative is to use the `ng-cloak` directive, which has the effect of hiding content until AngularJS has finished processing it. The `ng-cloak` directive uses CSS to hide the elements to which it is applied, and the AngularJS library removes the CSS class when the content has been processed, ensuring that the user never sees the `{{` and `}}` characters of a template expression. You can apply the `ng-cloak` directive as broadly or selectively as you want. A common approach is to apply the directive to the `body` element, but that just means that the user sees an empty browser window while AngularJS processes the content. I prefer to be more selective and apply the directive only to the parts of the document

where there are inline expressions, as shown in Listing 10-18.

Listing 10-18: Selectively Applying the ng-cloak Directive to the directives.html File

```
...
<body>
    <div id="todoPanel" class="panel" ng-controller="defaultCtrl">
        <h3 class="panel-header">To Do List</h3>

        <div class="well">
            <div class="radio" ng-repeat="button in ['None', 'Table', 'List']">
                <label ng-cloak>
                    <input type="radio" ng-model="data.mode"
                        value="{{button}}" ng-checked="$first">
                    {{button}}
                </label>
            </div>
        </div>

        <div ng-switch on="data.mode" ng-cloak>
            <div ng-switch-when="Table">
                <table class="table">
                    <thead>
                        <tr><th>#</th><th>Action</th><th>Done</th></tr>
                    </thead>
                    <tr ng-repeat="item in todos" ng-class="$odd ? 'odd' : 'even'">
                        <td>{{$index + 1}}</td>
                        <td ng-repeat="prop in item">{{prop}}</td>
                    </tr>
                </table>
            </div>
            <div ng-switch-when="List">
                <ol>
                    <li ng-repeat="item in todos">
                        {{item.action}}<span ng-if="item.complete"> (Done)</span>
                    </li>
                </ol>
            </div>
            <div ng-switch-default>
                Select another option to display a layout
            </div>
        </div>
    </div>
</body>
...
```

Applying the directive to the sections of the document that contain template expressions leaves the user able to see the static structure of a page, which still isn't ideal but is a lot better than just an empty window. You can see the effect the directive creates in Figure 10-10 (and, of course, the full app layout is revealed to the user when AngularJS finishes processing the content).

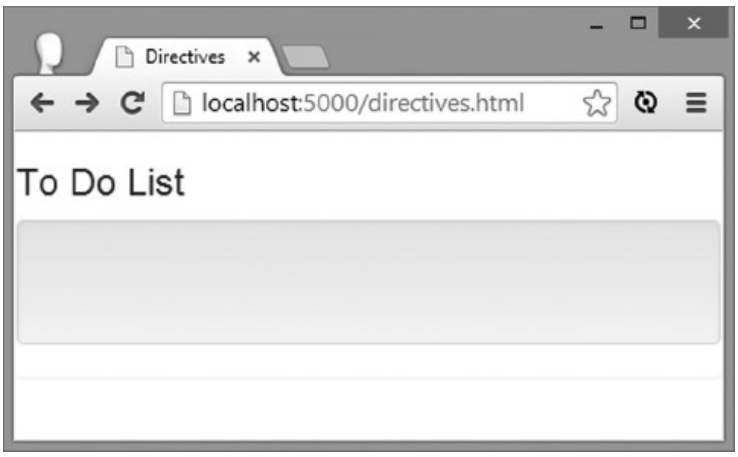

Figure 10-10: Showing static content without template expressions

Summary

In this chapter, I introduced you to AngularJS directives and described the directives that are used for data binding and managing templates. These are the most powerful and complex of the built-in templates, and they are the ones that underpin the early phases of development in an AngularJS project. In Chapter 11, I continue describing and demonstrating the built-in directives, focusing on the ones that manipulate elements and respond to events.