CHAPTER 11
Directives

Having explored all the other parts of AngularJS, like controllers, services, and filters, we dive deep into directives in this chapter. Directives are the AngularJS way of dealing with DOM manipulation and rendering reusable UI widgets. They can be used for simple things like reusing HTML snippets, to more complex things like modifying the behavior of existing elements (think `ng-show`, `ng-class`, or making elements draggable) or integrating with third-party components like charts and other fancy doodads.

In this chapter, we start with developing a very basic directive, and explore some of the more common options like `template`, `templateUrl`, `link`, and `scopes`. By the end of the chapter, we will have created multiple versions of our simple reusable widgets, each one building on top of the previous, along the way explaining how each directive definition object changes the functioning of our directive.

What Are Directives?

When we hear the word "directives," the very first association that should come to our minds is dealing directly with the UI or the HTML that the user sees. Directives are of two major types in AngularJS (though they can be subclassified further and further):

Behavior modifiers
: These types of directives work on existing UI and HTML snippets, and just add or modify the existing behavior of what the UI does. Examples of such directives would be `ng-show` (which hides or shows an existing element based on a condition), or `ng-model` (which adds the AngularJS data-binding hooks to any input to which it is attached).

Reusable components
: These types of directives are the more common variety, in which the directive creates a whole new HTML structure. These directives have some *rendering logic* (how and what should it display) and some *business logic* (where should it get the data,

what happens when the user interacts with it) attached to it. Some examples of these directives could be a tab widget, a carousel/accordian directive (though HTML5/6/7 might introduce these as a control), and a pie chart directive. These are also the best way to integrate third-party UI components into AngularJS (think jQuery UI or Google Charts).

While directives are the most powerful and complex part of AngularJS, the concepts they are built upon are quite simple. To ensure that we understand how they work, we'll introduce the options one at a time. This chapter focuses on some of the more straightforward and commonly used options for defining a directive, and the next chapter introduces some of the more complex and less used hooks of a directive.

Alternatives to Custom Directives

Before we jump into directives, let's quickly walk through some other options that might serve us well in the case that we want reusable HTML, or business logic in our HTML. We have two directives, `ng-include` and `ng-switch`, which can help us in extracting HTML into smaller chunks and deciding when to show and hide them in our HTML. In many cases, we can use these two directives instead of writing our custom directives.

ng-include

The `ng-include` directive takes an AngularJS expression (similar to `ng-show` and `ng-click`) and treats its value as the path to an HTML file. It then fetches that HTML file from the server and includes its content as the child (and the only child, replacing all other existing content) of the element that `ng-include` is placed on. Imagine that we extracted a *stock.html* file with the following content:

```
<!-- File: chapter11/ng-include/stock.html -->
<div class="stock-dash">
  Name:
  <span class="stock-name"
        ng-bind="stock.name">
  </span>
  Price:
  <span class="stock-price"
        ng-bind="stock.price | currency">
  </span>
  Percentage Change:
  <span class="stock-change"
        ng-bind="mainCtrl.getChange(stock) + '%'">
  </span>
</div>
```

This HTML is very simple in that it takes a variable called `stock` and displays its `name` and `price` in separate spans. In the last span, it calls a function on `mainCtrl` to calculate

and display the percentage change for the current stock. Let's now take a look at *index.html* for this application:

```
<!-- File: chapter11/ng-include/index.html -->
<html>
<head>
  <title>Stock Market App</title>
</head>
<body ng-app="stockMarketApp">

  <div ng-controller="MainCtrl as mainCtrl">
    <h3>List of Stocks</h3>
    <div ng-repeat="stock in mainCtrl.stocks">
      <div ng-include="mainCtrl.stockTemplate">
      </div>
    </div>
  </div>

  <script src="http://code.angularjs.org/1.2.16/angular.js"></script>
  <script src="app.js"></script>
</body>
</html>
```

The main *index.html* file loads AngularJS and our application code, and instantiates `ng-app` (`stockMarketApp`). It then loads a controller on the main `div`, and displays a list of stocks inside of it. We're extracting the content of `ng-repeat` into *stock.html*, instead of having it inline. We then tell the `ng-include` to load whatever `mainCtrl.stockTemplate` points to. Let's now see what the controller is doing:

```
// File: chapter11/ng-include/app.js

angular.module('stockMarketApp', [])
  .controller('MainCtrl', [function() {
    var self = this;
    self.stocks = [
      {name: 'First Stock', price: 100, previous: 220},
      {name: 'Second Stock', price: 140, previous: 120},
      {name: 'Third Stock', price: 110, previous: 110},
      {name: 'Fourth Stock', price: 400, previous: 420}
    ];

    self.stockTemplate = 'stock.html';

    self.getChange = function(stock) {
      return Math.ceil((
        (stock.price - stock.previous) / stock.previous) * 100);
    };
  }]);
```

`MainCtrl` defines a list of stocks in its controller, each with a name, price, and previous price. It also defines a `getChange` function, which is used to figure out the percentage

change for a particular stock that is passed to it (it also multiplies it by 100, and rounds it off for easier display).

Note that in the previous example, we included the following in *index.html*:

```
<div ng-include="mainCtrl.stockTemplate"></div>
```

This required us to define a variable in the controller and refer to it. Another option is to inline the string directly in the HTML for the `ng-include`. When we inline it, we have to ensure that AngularJS understands that we have not passed it a variable on our controller, but the actual value itself. We could do that as follows:

```
<div ng-include="'views/stock.html'"></div>
```

Notice the single quotes inside the double quotes. The single quotes tell AngularJS that the value passed to it is a string literal, not a variable. If we don't add the single quotes, AngularJS will look for a variable called `views/stock.html` (which obviously is an illegal variable name, and doesn't exist), and throw an error saying that it is unable to parse the expression. So don't forget to include the single quotes if you're directly using the filename in the HTML.

Any time we have to serve HTML partials, we need an HTTP server because the browser does not allow serving or requesting files on the *file://* protocol. So to make this application work on our local machine, we have a few options:

- Install Node's `http-server` by running **`sudo npm install -g http-server`** (drop `sudo` if you're on Windows). Then run `http-server` from the directory that contains *index.html*.
- Python addicts can run **`python -m SimpleHTTPServer`** from the folder where the *index.html* file is as well.
- Finally, WebStorm can start a built-in server when you ask it to open the *index.html* file in a browser.

At this point if we run this application, we will see an HTML page that displays four stocks, each with its own name, price, and percentage change. `ng-include` helped us by allowing us to extract the HTML that would otherwise have been in *index.html* (technically, this could be any file that we are working on), and extract it to a smaller, reusable, and easier-to-maintain file.

This is the best feature of `ng-include`. If we have large HTML files, we can easily extract them out into smaller, easier-to-manage HTML files and make our HTML modular as well.

Limitations of ng-include

While `ng-include` is great at extracting snippets of HTML into smaller files, it is not without limitations. The *stock.html* file we created has two major limitations if we include it using the `ng-include` directive in AngularJS:

- The *stock.html* file currently looks for a variable called `stock` and displays its name, price, and percentage change information. Now, in *index.html*, if we ended up changing the repeater from `stock` in `mainCtrl.stocks` to `each` in `mainCtrl.stocks`, we would end up showing four empty blocks without the name, price, or change information. This is because although we changed the name of the variable in the main *index.html* file, the *stock.html* file still expects a variable called `stock` for it to display. Thus, if we are using `ng-include`, each file that includes *stock.html* must name the variable containing the information `stock`.
- The *stock.html* file is also currently dependent on being used along with `mainCtrl`. This is because it expects there to be a `mainCtrl.getChange` function that it uses to calculate the percentage change of the stock. If *stock.html* is used in some other HTML that either does not have the controller named `mainCtrl`, or if the controller does not have the function `getChange()`, then the HTML will not be able to display the percentage change. Thus, the behavior that the extracted HTML depends on will have to be manually included with the right name every time this HTML is used.

In "Creating a Directive" on page 175, we will see how to fix both of these problems using directives that we create.

ng-switch

The `ng-switch` is another directive that allows us to add some functionality to the UI for selectively displaying certain snippets of HTML. It gives us a way of conditionally including HTML snippets by behaving like a switch case directly in the HTML. Here is how a simple usage of `ng-switch` might operate:

```
<!-- File: chapter11/ng-switch/index.html -->
<html>
<head>
  <title>Switch App</title>
</head>
<body ng-app="switchApp">

  <div ng-controller="MainCtrl as mainCtrl">
    <h3>Conditional Elements in HTML</h3>
    <button ng-click="mainCtrl.currentTab = 'tab1'">
      Tab 1
    </button>
```

```
    <button ng-click="mainCtrl.currentTab = 'tab2'">
      Tab 2
    </button>
    <button ng-click="mainCtrl.currentTab = 'tab3'">
      Tab 3
    </button>
    <button ng-click="mainCtrl.currentTab = 'something'">
      Trigger Default
    </button>

    <div ng-switch="mainCtrl.currentTab">
      <div ng-switch-when="tab1">
        Tab 1 is selected
      </div>
      <div ng-switch-when="tab2">
        Tab 2 is selected
      </div>
      <div ng-switch-when="tab3">
        Tab 3 is selected
      </div>
      <div ng-switch-default>
        No known tab selected
      </div>
    </div>
  </div>

  <script src="http://code.angularjs.org/1.2.16/angular.js"></script>
  <script type="text/javascript">
    angular.module('switchApp', [])
        .controller('MainCtrl', [function() {
          this.currentTab = 'tab1';
        }]);
  </script>
</body>
</html>
```

This example is a simple application that shows five buttons. Clicking any of the first three buttons opens that tab. The last two buttons set a random value for the current tab. In such a case, the default case is triggered and the last `div` is shown.

Running this example requires the same steps as the previous example, so get a locally running server started.

It accomplishes this using the `ng-switch` directive. We add an `ng-switch` based on the value of `mainCtrl.currentTab`. Inside the `div`, we add multiple divs, which are shown selectively. We accomplish this by adding `ng-switch-when` attributes to the children elements. We add the conditions (like a `select` statement). Each `ng-switch-when` takes a string value (for example, "hello"). If this string value matches the value of the expression passed to `ng-switch` (in this case, the value of the variable `mainCtrl.current`

Tab), then the element is displayed. If none of the ng-switch-when values match the value of the original expression, then the ng-switch-default case is triggered.

There are a few things to note about ng-switch:

- ng-switch loads its content, and then based on the condition, comments out all the ng-switch-when conditions that are not satisfied. So even if we use ng-include inside ng-switch-when, they will not get loaded up front, and will be loaded only when the condition is met.
- ng-switch-when is treated as an attribute, and thus the value passed to it is expected to be direct, and not an AngularJS expression. That is, suppose we have ng-switch="mainCtrl.currentTab" and then ng-switch-when="mainCtrl.possibleValue". This would expect the value of currentTab in the controller to be equal to the string "mainCtrl.possibleValue", and not the value of mainCtrl.possibleValue. ng-switch-when does not understand AngularJS expressions.

Understanding the Basic Options

We now know how and when to use ng-include and ng-switch, and understand their shortcomings. Now let's see how we can create a directive to solve some of these problems.

The main intentions of a directive are:

- To make our intention declarative by specifying in the HTML what something is or tries to do.
- To make something reusable so the same functionality can be achieved easily without copying and pasting the code.
- To achieve abstraction in the sense that the user of the directive doesn't need to know or understand how something is performed, but only cares about the end result. The corollary of this is that the underlying implementation can be changed without having to change every single usage.

Let's jump into the options that are available when creating a directive using the example from the ng-include, and see how we might convert that into a proper, reusable directive.

Creating a Directive

Creating a directive is just like creating controllers, services, and filters in that the AngularJS module function allows us to create a directive by name. The first argument to

the function is the name of the directive, and the second argument is the standard Dependency Injection array syntax, with the last element in the array being our directive function.

Suppose we want to turn the *stock.html* file from the previous example into a reusable directive. Let's start with how we want to use it in our HTML:

```
<div stock-widget></div>
```

If we want to be able to declare that the current div is a stock widget (used as stated previously), we need to declare or create our directive as follows:

```
angular.module('stockMarketApp', [])
  .directive('stockWidget', [function() {
    return {
      // Directive definition will go here
    };
}]);
```

We define the `stockWidget` directive and provide it with a function. This function sets up our directive using what we call a *directive definition object* and returns this definition. AngularJS looks at this definition each time it encounters our directive in the HTML.

By default, when we create a directive this way, we can use it only as an attribute of existing elements in our HTML. That is, by default, we can only use `<div stock-widget>` and not `<stock-widget>`.

Directive and Attribute Naming

One thing to note is the naming of the directive. HTML is case-insensitive by default. To deal with this when translating names of attributes and directives from HTML to JavaScript, AngularJS converts dashes to camelCase. Thus, `stock-widget` (or `STOCK-WIDGET` or even `Stock-Widget`) in HTML becomes `stockWidget` in JavaScript.

Next, let's take a look at some commonly used options when creating directives.

Template/Template URL

The very first thing we can define as part of our directive is whether it has any content that needs to be inserted when the directive is encountered. We do this using the `template` and `templateUrl` keys of the directive definition object (similar to routing).

Let's now achieve the same functionality that we had with the `ng-include`, but using a directive so that it's more declarative. First, the *app.js* file, which remains unchanged:

```
// This is chapter11/directive-with-template/app.js

angular.module('stockMarketApp', [])
  .controller('MainCtrl', [function() {
    var self = this;
    self.stocks = [
      {name: 'First Stock', price: 100, previous: 220},
      {name: 'Second Stock', price: 140, previous: 120},
      {name: 'Third Stock', price: 110, previous: 110},
      {name: 'Fourth Stock', price: 400, previous: 420}
    ];

    self.getChange = function(stock) {
      return Math.ceil((
        (stock.price - stock.previous) / stock.previous) * 100);
    };
  }]);
```

Similarly, the *stock.html* file also remains unchanged from the previous example:

```
<!-- File: chapter11/directive-with-template/stock.html -->
<div class="stock-dash">
  Name:
  <span class="stock-name"
        ng-bind="stock.name">
  </span>
  Price:
  <span class="stock-price"
        ng-bind="stock.price | currency">
  </span>
  Percentage Change:
  <span class="stock-change"
        ng-bind="mainCtrl.getChange(stock) + '%'">
  </span>
</div>
```

The *index.html* file next changes slightly because it now uses our directive instead of `ng-include`:

```
<!-- File: chapter11/directive-with-template/index.html -->
<html>
<head>
  <title>Stock Market App</title>
</head>
<body ng-app="stockMarketApp">

  <div ng-controller="MainCtrl as mainCtrl">
    <h3>List of Stocks</h3>
    <div ng-repeat="stock in mainCtrl.stocks">
        <div stock-widget>
```

```
        </div>
      </div>
    </div>

    <script src="http://code.angularjs.org/1.2.16/angular.js"></script>
    <script src="app.js"></script>
    <script src="directive.js"></script>
  </body>
  </html>
```

Only the content of `ng-repeat` has changed; it now states `<div stock-widget></div>`. Finally, let's see how this directive is defined and created:

```
// File: chapter11/directive-with-template/directive.js
angular.module('stockMarketApp')
  .directive('stockWidget', [function() {
    return {
      templateUrl: 'stock.html'
    };
  }]);
```

In this file, we create a directive with a very simple definition. We tell AngularJS that any time we encounter a directive in our HTML called `stock-widget` (because we named it `stockWidget`, which translates to `stock-widget` in HTML), it should fetch the template *stock.html* from our server and insert it as a child of the element the directive is placed on.

Note that AngularJS will be smart and fetch the HTML template that's at the `templateUrl` location only once when the directive is encountered the very first time. After that, it saves the template in its local cache and serves it from there.

If our HTML is small enough, we could possibly look at inlining it directly in our directive using the `template` key in the directive definition object. Here is how our directive might look if we did away with the *stock.html* file and inlined it in our directive:

```
angular.module('stockMarketApp')
  .directive('stockWidget', [function() {
    return {
      template: '<div class="stock-dash">' +
        'Name: ' +
        '<span class="stock-name"' +
              'ng-bind="stock.name">' +
        '</span>' +
        'Price: ' +
        '<span class="stock-price"' +
              'ng-bind="stock.price | currency">' +
        '</span>' +
        'Change: ' +
```

```
        '<span class="stock-change"' +
              'ng-bind="mainCtrl.getChange(stock) + '%'">' +
        '</span>' +
        '</div>'
    };
  }]);
```

Note that this is a replacement for the directive we created before. Both have the exact same effect in terms of the UI, in that they load an HTML template and place it as a child of the element that the directive is on.

The `template` key makes sense if our HTML snippet is small and easy to maintain. For larger, more complex templates, it almost always makes sense to load them via the `templateUrl` key.

Restrict

The `restrict` keyword defines how someone using the directive in their code might use it. As mentioned previously, the default way of using directives is via attributes of existing elements (we used `<div stock-widget>` for ours).

When we create our directive, we have control in deciding how it's used. The possible values for `restrict` (and thus the ways in which we can use our directive) are:

`A`
: The letter `A` in the value for `restrict` specifies that the directive can be used as an *attribute* on existing HTML elements (such as `<div stock-widget></div>`). This is the default value.

`E`
: The letter `E` in the value for `restrict` specifies that the directive can be used as a new HTML *element* (such as `<stock-widget></stock-widget>`).

`C`
: The letter `C` in the value for `restrict` specifies that the directive can be used as a class name in existing HTML elements (such as `<div class="stock-widget"></div>`).

`M`
: The letter `M` in the value for `restrict` specifies that the directive can be used as HTML comments (such as `<!-- directive: stock-widget -->`). This was previously necessary for directives that needed to encompass multiple elements, like multiple rows in tables, etc. The `ng-repeat-start` and `ng-repeat-end` directives were introduced for this sole purpose, so it's preferable to use them instead of comment directives.

Each of these can be used by itself as an argument to the `restrict` key, or could be used in combination with each other. Here's how we might update our `stock-widget` directive to be able to use it as either attributes or elements in our HTML:

```
// File: chapter11/directive-with-restrict/directive.js
angular.module('stockMarketApp')
  .directive('stockWidget', [function() {
    return {
      templateUrl: 'stock.html',
      restrict: 'AE'
    };
  }]);
```

We added a `restrict` key to our directive definition object, and gave it a value of `AE`. This tells AngularJS that we can use the widget in our HTML as either `<div stock-widget>` or directly as `<stock-widget>`. This also prevents us from using `<div class="stock-widget">`, because we have not allowed it to be used as class names.

Expressions in Class Directives

We have seen four possibilities for the `restrict` key. We will explore in the following sections how to pass values to our directive, but we might wonder how that is possible with the class-based directive. That is, how would `<div my-widget="someExp">` translate to a class-based directive?

Class-based directives translate to `<div class="my-widget: someExp;">` and AngularJS would treat this as similar to passing a value to an attribute in HTML.

Now that we've explored the various ways in which directives can be used, let's cover some best practices around their usage:

- Internet Explorer 8 and below do not like custom HTML elements. If we plan on using them, we need to manually tell Internet Explorer that we have some new elements by calling `document.createElement('stock-widget')` and so on for each new element. AngularJS, with version 1.3 onwards, has dropped support for (or rather, testing on) Internet Explorer 8.
- Class-based directives are ideal for rendering-related work (like the `ng-cloak` directive that hides and shows elements, or image loading directives).
- Element directives are recommended if we are creating entirely new HTML content.
- Attribute directives are usually preferred for behavior modifiers (like `ng-show`, `ng-class`, and so on).

The link Function

The `link` keyword in the directive definition object is used to add what we call a "link function" for the directive. The `link` function does for a directive what a controller does for a view—it defines APIs and functions that are necessary for the directive, in addition to manipulating and working with the DOM.

AngularJS executes the `link` function for each instance of the directive, so each instance can get its own, fully contained business logic while not affecting any other instance of the directive. The `link` function gets a standard set of arguments passed to it that remain consistent across directives, which looks something like the following:

```
link: function($scope, $element, $attrs) {}
```

The `link` function gets passed the scope of the element the directive is working on, the HTML DOM element the directive is operating on, and all the attributes on the element as strings. If we need to add functionality to our instance of the directive, we can add it to the scope of the element we're working with.

Let's take our example from before and move the functionality from `mainCtrl` into our directive, so the directive doesn't have to depend on there being a `MainCtrl` with a function called `getChange`:

```
// File: chapter11/directive-with-link/directive.js
angular.module('stockMarketApp')
  .directive('stockWidget', [function() {
    return {
      templateUrl: 'stock.html',
      restrict: 'AE',
      link: function($scope, $element, $attrs) {
        $scope.getChange = function(stock) {
          return Math.ceil(((stock.price - stock.previous) /
              stock.previous) * 100);
        };
      }
    };
  }]);
```

In this code snippet, the directive defines a `link` function that adds a function called `getChange()` on its scope. This makes each instance of the directive have a `getChange()` function on its own scope. The function was moved from the controller to the directive, without any other changes. Now let's see how we might use it in *stock.html*:

```
<!-- File: chapter11/directive-with-link/stock.html -->
<div class="stock-dash">
  Name:
  <span class="stock-name"
        ng-bind="stock.name">
  </span>
  Price:
```

```
    <span class="stock-price"
          ng-bind="stock.price | currency">
    </span>
    Percentage Change:
    <span class="stock-change"
          ng-bind="getChange(stock) + '%'">
    </span>
  </div>
```

The *stock.html* file remains largely unchanged, except for the change in the last `ng-bind` directive. Instead of binding to `mainCtrl.getChange(stock)`, we bind to `getChange(stock)`. This implies that it's looking for a `getChange()` function on its own scope. Now regardless of which controller we use our directive in, or even if we rename our controller, the `stockWidget` directive has its own version of the `getChange` function. It becomes independent.

The `link` function is also where we can define our own listeners, work directly with the DOM element, and much more. We'll explore more of these in Chapter 13.

What's the Scope?

You might wonder what scope we are adding these functions to. You're right to be worried, and it is something we should always keep in mind when we add functions to the scope in the `link` function. In the example in this section, `ng-repeat` in the main *index.html* file creates a scope for each stock in our array, and it is to this scope that we're adding the functions. Because of this, we're not affecting the controller's scope directly, but that is an unintentional side effect of `ng-repeat`. If we had used our directive outside `ng-repeat`, we would have ended up modifying the controller's scope directly, which is bad practice.

The default scope given in the `link` function (unless specified otherwise) is the scope that the parent has. Adding functions to the parent scope should always be frowned upon, because the parent should ideally not be changed from within a child.

We are also still dependent on the variable name `stock`, which if renamed breaks our UI. We'll see in the next section how to remove this dependency as well.

Scope

By default, each directive inherits its parent's scope, which is passed to it in the `link` function. This can lead to the following problems:

- Adding variables/functions to the scope modifies the parent as well, which suddenly gets access to more variables and functions.