AngularJS Web Application Development Cookbook

Over 90 hands-on recipes to architect performant applications and implement best practices in AngularJS

Matt Frisbie
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 "Maximizing AngularJS Directives"
- A synopsis of the book’s content
- More information on AngularJS Web Application Development Cookbook

About the Author

Matt Frisbie is currently a full stack developer at DoorDash (YC S13), where he joined as the first engineer. He led their adoption of AngularJS, and he also focuses on the infrastructural, predictive, and data projects within the company.

Matt has a degree in Computer Engineering from the University of Illinois at Urbana-Champaign. He is the author of the video series Learning AngularJS, available through O'Reilly Media. Previously, he worked as an engineer at several educational technology start-ups.
AngularJS Web Application Development Cookbook

"Make it work. Make it right. Make it fast."

Back when the world was young, Kent Beck forged this prophetic sentiment. Even today, in the ultra-modern realm of performant single-page application JavaScript frameworks, his idea still holds sway. This nine-word expression describes the general progression through which a pragmatic developer creates high-quality software.

In the process of discovering how to optimally wield a technology, a developer will execute this progression many times, and each time will be a learning experience regarding some new understanding of the technology.

This cookbook is intended to act as a companion guide through this process. The recipes in this book will intimately examine every major aspect of the framework in order to maximize your comprehension. Every time you open this book, you should gain an expanded understanding of the brilliance of the AngularJS framework.

What This Book Covers

Chapter 1, Maximizing AngularJS Directives, dissects the various components of directives and demonstrates how to wield them in your applications. Directives are the bread and butter of AngularJS, and the tools presented in this chapter will maximize your ability to take advantage of their extensibility.

Chapter 2, Expanding Your Toolkit with Filters and Service Types, covers two major tools for code abstraction in your application. Filters are an important pipeline between the model and its appearance in the view, and are essential tools for managing data presentation. Services act as broadly applicable houses for dependency-injectable modules and resource access.

Chapter 3, AngularJS Animations, offers a collection of recipes that demonstrate various ways to effectively incorporate animations into your application. Additionally, it will dive deep down into the internals of animations in order to give you a complete perspective on how everything really works under the hood.

Chapter 4, Sculpting and Organizing Your Application, gives you strategies for controlling the application initialization, organizing your files and modules, and managing your template delivery.

Chapter 5, Working with the Scope and Model, breaks open the various components involving ngModel and provides details of the ways in which they can integrate into your application flow.
Chapter 6, *Testing in AngularJS*, gives you all the pieces you need to jump into writing test-driven applications. It demonstrates how to configure a fully operational testing environment, how to organize your test files and modules, and everything involved in creating a suite of unit and E2E tests.

Chapter 7, *Screaming Fast AngularJS*, is a response to anyone who has ever complained about AngularJS being slow. The recipes in this chapter give you all the tools you need to tune all aspects of your application's performance and take it from a steam engine to a bullet train.

Chapter 8, *Promises*, breaks apart the asynchronous program flow construct, exposes its internals, then builds it all the way back up to discuss strategies for your application's integration. This chapter also demonstrates how promises can and should integrate into your application's routing and resource access utilities.

Chapter 9, *What's New in AngularJS 1.3*, goes through how your application can integrate the slew of new features and changes that were introduced in the AngularJS 1.3 and the later AngularJS 1.2.x releases.

Chapter 10, *AngularJS Hacks*, is a collection of clever and interesting strategies that you can use to stretch the boundaries of AngularJS's organization and performance.
Maximizing AngularJS Directives

In this chapter, we will cover the following recipes:

- Building a simple element directive
- Working through the directive spectrum
- Manipulating the DOM
- Linking directives
- Interfacing with a directive using isolate scope
- Interaction between nested directives
- Optional nested directive controllers
- Directive scope inheritance
- Directive templating
- Isolate scope
- Directive transclusion
- Recursive directives

Introduction

In this chapter, you will learn how to shape AngularJS directives in order to perform meaningful work in your applications. Directives are perhaps the most flexible and powerful tool available to you in this framework and utilizing them effectively is integral to architecting clean and scalable applications. By the same token, it is very easy to fall prey to directive antipatterns, and in this chapter, you will learn how to use the features of directives appropriately.
Building a simple element directive

One of the most common use cases of directives is to create custom HTML elements that are able to encapsulate their own template and behavior. Directive complexity increases very quickly, so ensuring your understanding of its foundation is essential. This recipe will demonstrate some of the most basic features of directives.

How to do it...

Creating directives in AngularJS is accomplished with a directive definition object. This object, which is returned from the definition function, contains various properties that serve to shape how a directive will act in your application.

You can build a simple custom element directive easily with the following code:

(app.js)

// application module definition
angular.module('myApp', [])
  .directive('myDirective', function() {
    // return the directive definition object
    return {
      // only match this directive to element tags
      restrict: 'E',
      // insert the template matching 'my-template.html'
      templateUrl: 'my-template.html'
    };
  });

As you might have guessed, it's bad practice to define your directive template with the template property unless it is very small, so this example will skip right to what you will be using in production: templateUrl and $templateCache. For this recipe, you'll use a relatively simple template, which can be added to $templateCache using ng-template. An example application will appear as follows:

(index.html)

<!-- specify root element of application -->
<div ng-app="myApp">
  <!-- register 'my-template.html' with $templateCache -->
  <script type="text/ng-template" id="my-template.html">
    <div ng-repeat="num in [1,2,3,4,5]">{{ num }}</div>
  </script>
  <!-- your custom element -->
  <my-directive></my-directive>
</div>
When AngularJS encounters an instance of a custom directive in the index.html template, it will compile the directive into HTML that makes sense to the browser, which will look as follows:

```html
<div>1</div>
<div>2</div>
<div>3</div>
<div>4</div>
<div>5</div>
```

[JSFiddle: http://jsfiddle.net/msfrisbie/uwpdptLn/]

**How it works...**

The restrict: 'E' statement indicates that your directive will appear as an element. It simply instructs AngularJS to search for an element in the DOM that has the my-directive tag.

Especially in the context of directives, you should always think of AngularJS as an HTML compiler. AngularJS traverses the DOM tree of the page to look for directives (among many other things) that it needs to perform an action for. Here, AngularJS looks at the `<my-directive>` element, locates the relevant template in $templateCache, and inserts it into the page for the browser to handle. The provided template will be compiled in the same way, so the use of `ng-repeat` and other AngularJS directives is fair game, as demonstrated here.

**There's more...**

A directive in this fashion, though useful, isn't really what directives are for. It provides a nice jumping-off point and gives you a feel of how it can be used. However, the purpose that your custom directive is serving can be better implemented with the built-in `ng-include` directive, which inserts a template into the designated part of HTML. This is not to say that directives shouldn't ever be used this way, but it's always good practice not to reinvent the wheel. Directives can do much more than template insertion (which you will soon see), and it's best to leave the simple tasks to the tools that AngularJS already provides to you.

**Working through the directive spectrum**

Directives can be incorporated into HTML in several different ways. Depending on how this incorporation is done, the way the directive will interact with the DOM will change.
Maximizing AngularJS Directives

How to do it...

All directives are able to define a link function, which defines how that particular directive instance will interact with the part of the DOM it is attached to. The link functions have three parameters by default: the directive scope (which you will learn more about later), the relevant DOM element, and the element’s attributes as key-value pairs.

A directive can exist in a template in four different ways: as an HTML pseudo-element, as an HTML element attribute, as a class, and as a comment.

The element directive

The element directive takes the form of an HTML tag. As with any HTML tag, it can wrap content, have attributes, and live inside other HTML elements.

The directive can be used in a template in the following fashion:

(index.html)

```html
<div ng-app="myApp">
  <element-directive some-attr="myvalue">
    <!-- directive's HTML contents -->
  </element-directive>
</div>
```

This will result in the directive template replacing the wrapped contents of the `element-directive` tag with the template. This element directive can be defined as follows:

(app.js)

```javascript
angular.module('myApp', [])
    .directive('elementDirective', function ($log) {
    return {
      restrict: 'E',
      template: '<p>Ze template!</p>',
      link: function(scope, el, attrs) {
        $log.log(el.html());
        // <p>Ze template!</p>
        $log.log(attrs.someAttr);
        // myvalue
      }
    };
    });
```
Note that for both the tag string and the attribute string, AngularJS will match the CamelCase for `elementDirective` and `someAttr` to their hyphenated `element-directive` and `some-attr` counterparts in the markup.

If you want to replace the directive tag entirely with the content instead, the directive will be defined as follows:

```
(index.html)

angular.module('myApp', [])
.directive('elementDirective', function ($log) {
  return {
    restrict: 'E',
    replace: true,
    template: '<p>Ze template!</p>',
    link: function(scope, el, attrs) {
      $log.log(el.html());
      // Ze template!
      $log.log(attrs.someAttr);
      // myvalue
    }
  };
});
```

This approach will operate in an identical fashion, but the directive's inner HTML will not be wrapped with `<element-directive>` tags in the compiled HTML. Also, note that the logged template is missing its `<p></p>` tags that have become the root directive element as they are the top-level tags inside the template.

**The attribute directive**

Attribute directives are the most commonly used form of directives, and for good reason. They have the following advantages:

- They can be added to existing HTML as standalone attributes, which is especially convenient if the directive's purpose doesn't require you to break up an existing template into fragments
It is possible to add an unlimited amount of attribute directives to an HTML element, which is obviously not possible with an element directive. Attribute directives attached to the same HTML element are able to communicate with each other (refer to the Interaction between nested directives recipe).

This directive can be used in a template in the following fashion:

(index.html)

```html
<div ng-app="myApp">
  <div attribute-directive="aval"
       some-attr="myvalue">
  </div>
</div>
```

A nonstandard element's attributes need the data- prefix to be compliant with the HTML5 specification. That being said, pretty much every modern browser will have no problem if you leave it out.

The attribute directive can be defined as follows:

(app.js)

```javascript
angular.module('myApp', [])
  .directive('attributeDirective', function ($log) {
    return {
      restrict: 'A',
      template: '<p>An attribute directive</p>',
      link: function(scope, el, attrs) {
        $log.log(el.html());
        // <p>An attribute directive</p>
        $log.log(attrs.attributeDirective);
        // aval
        $log.log(attrs.someAttr);
        // myvalue
      }
    };
  });
```

[JSFiddle: http://jsfiddle.net/msfrisbie/y2tsgxjt/]
Other than its form in the HTML template, the attribute directive functions in pretty much the same way as an element directive. It assumes its attribute values from the container element's attributes, including the attribute directive and other directives (whether or not they are assigned a value).

**The class directive**

Class directives are not altogether that different from attribute directives. They provide the ability to have multiple directive assignments, unrestricted local attribute value access, and local directive communication.

This directive can be used in a template in the following fashion:

```
(index.html)
<div ng-app="myApp">
  <div class="class-directive: cval; normal-class"
    some-attr="myvalue">
  </div>
</div>
```

This attribute directive can be defined as follows:

```
(app.js)
angular.module('myApp', [])
  .directive('classDirective', function ($log) {
    return {
      restrict: 'C',
      template: '<p>A class directive</p>',
      link: function(scope, el, attrs) {
        $log.log(el.html());
        // p>A class directive</p>
        $log.log(el.hasClass('normal-class'));
        // true
        $log.log(attrs.classDirective);
        // cval
        $log.log(attrs.someAttr);
        // myvalue
      }
    };
  });
```

[JSFiddle: http://jsfiddle.net/msfrisbie/rt1f4qxx/](http://jsfiddle.net/msfrisbie/rt1f4qxx/)
It's possible to reuse class directives and assign CSS styling to them, as AngularJS leaves them alone when compiling the directive. Additionally, a value can be directly applied to the directive class name attribute by passing it in the CSS string.

**The comment directive**

Comment directives are the runt of the group. You will very infrequently find their use necessary, but it's useful to know that they are available in your application.

This directive can be used in a template in the following fashion:

```html
<div ng-app="myApp">
  <!-- directive: comment-directive val1 val2 val3 -->
</div>
```

The comment directive can be defined as follows:

```javascript
angular.module('myApp', [])
  .directive('commentDirective', function ($log) {
    return {
      restrict: 'M',
      // without replace: true, the template cannot
      // be inserted into the DOM
      replace: true,
      template: '<p>A comment directive</p>',
      link: function(scope, el, attrs) {
        $log.log(el.html())
        // <p>A comment directive</p>
        $log.log(attrs.commentDirective)
        // 'val1 val2 val3'
      }
    };
  });
```

[JSFiddle: http://jsfiddle.net/msfrisbie/thfvx275/](http://jsfiddle.net/msfrisbie/thfvx275/)
Formerly, the primary use of comment directives was to handle scenarios where the DOM API made it difficult to create directives with multiple siblings. Since the release of AngularJS 1.2 and the inclusion of ng-repeat-start and ng-repeat-end, comment directives are considered an inferior solution to this problem, and therefore, they have largely been relegated to obscurity. Nevertheless, they can still be employed effectively.

**How it works...**

AngularJS actively compiles the template, searching for matches to defined directives. It's possible to chain directive forms together within the same definition. The mydir directive with restrict: 'EACM' can appear as follows:

```html
<mydir></mydir>
<div mydir></div>
<div class="mydir"></dir>
<!-- directive: mydir -->
```

**There's more...**

The $log.log() statements in this recipe should have given you some insight into the extraordinary use that directives can have in your application.

**See also**

- The *Interaction between nested directives* recipe demonstrates how to allow directives attached to the same element to communicate with each other

**Manipulating the DOM**

In the previous recipe, you built a directive that didn't care what it was attached to, what it was in, or what was around it. Directives exist for you to program the DOM, and the equivalent of the last recipe is to instantiate a variable. In this recipe, you will actually implement some logic.
How to do it...

The far more common use case of directives is to create them as an HTML element attribute (this is the default behavior for `restrict`). As you can imagine, this allows us to decorate existing material in the DOM, as follows:

```javascript
angular.module('myApp', [])
 .directive('counter', function () {
   return {
     restrict: 'A',
     link: function (scope, el, attrs) {
       // read element attribute if it exists
       var incr = parseInt(attrs.incr || 1), val = 0;
       // define callback for vanilla DOM click event
       el.bind('click', function () {
         el.html(val += incr);
       });
     }
   };
});
```

This directive can then be used on a `<button>` element as follows:

```html
<div ng-app="myApp">
  <button counter></button>
  <button counter incr="5"></button>
</div>
```

How it works...

AngularJS includes a subset of jQuery (dubbed jqLite) that lets you use a core toolset to modify the DOM. Here, your directive is attached to a singular element that the directive sees in its linking function as the element parameter. You are able to define your DOM modification logic here, which includes initial element modification and the setup of events.
In this recipe, you are consuming a static attribute value `incr` inside the `link` function as well as invoking several jqLite methods on the element. The element parameter provided to you is already packaged as a jqLite object, so you are free to inspect and modify it at your will. In this example, you are manually increasing the integer value of a counter, the result of which is inserted as text inside the button.

**There's more...**

Here, it's important to note that you will never need to modify the DOM in your controller, whether it is a directive controller or a general application controller. Because AngularJS and JavaScript are very flexible languages, it's possible to contort them to perform DOM manipulation. However, managing the DOM transformation out of place causes an undesirable dependency between the controller and the DOM (they should be totally decoupled) as well as makes testing more difficult. Thus, a well-formed AngularJS application will never modify the DOM in controllers. Directives are tailor-made to layer and group DOM modification tasks, and you should have no trouble using them as such.

Additionally, it's worth mentioning that the `attrs` object is read-only, and you cannot set attributes through this channel. It's still possible to modify attributes using the element attribute, but state variables for elements can be much more elegantly implemented, which will be discussed in a later recipe.

**See also**

- In this recipe, you saw the `link` function used for the first time in a fairly rudimentary fashion. The next recipe, *Linking directives*, goes into further detail.
- The *Isolate scope* recipe goes over the writable DOM element attributes that can be used as state variables.

**Linking directives**

For a large subset of the directives you will eventually build, the bulk of the heavy lifting will be done inside the directive's `link` function. This function is returned from the preceding compile function, and as seen in the previous recipe, it has the ability to manipulate the DOM in and around it.

**How to do it...**

The following directive will display NW, NE, SW, or SE depending on where the cursor is relative to it:

```javascript
angular.module('myApp', [])
.directive('vectorText', function ($document) {
```

```javascript
};

```
return {
    template: '<span>{{ heading }}</span>',
    link: function (scope, el, attrs) {

        // initialize the css
        el.css({
            'float': 'left',
            'padding': attrs.buffer + 'px'
        });

        // initialize the scope variable
        scope.heading = '';

        // set event listener and handler
        $document.on('mousemove', function (event) {
            // mousemove event does not start $digest,
            // scope.$apply does this manually
            scope.$apply(function () {
                if (event.pageY < 300) {
                    scope.heading = 'N';
                } else {
                    scope.heading = 'S';
                }
                if (event.pageX < 300) {
                    scope.heading += 'W';
                } else {
                    scope.heading += 'E';
                }
            });
        });
    }
};

This directive will appear in the template as follows:

(index.html)

    <div ng-app="myApp">
        <div buffer="300" vector-text>
            </div>
    </div>
How it works...

This directive has a lot more to wrap your head around. You can see that it has $document injected into it, as you need to define event listeners relevant to this directive all across $document. Here, a very simple template is defined, which would preferably be in its own file, but for the sake of simplicity, it is merely incorporated as a string.

This directive first initializes the element with some basic CSS in order to have the relevant anchor point somewhere you can move the cursor around fully. This value is taken from an element attribute in the same fashion it was used in the previous recipe.

Here, our directive is listening to a $document mousemove event, with a handler inside wrapped in the scope.$apply() wrapper. If you remove this scope.$apply() wrapper and test the directive, you will notice that while the handler code does execute, the DOM does not get updated. This is because the event that the application is listening for does not occur in the AngularJS context—it is merely a browser DOM event, which AngularJS does not listen for. In order to inform AngularJS that models might have been altered, you must utilize the scope.$apply() wrapper to trigger the update of the DOM.

With all of this, your cursor movement should constantly be invoking the event handler, and you should see a real-time description of your cursor's relative cardinal locality.

There's more...

In this directive, we have used the scope parameter for the first time. You might be wondering, "Which scope am I using? I haven't declared any specific scope anywhere else in the application." Recall that a directive will inherit a scope unless otherwise specified, and this recipe is no different. If you were to inject $rootScope to the directive and log to the $rootScope.heading console inside the event handler, you would see that this directive is writing to the heading attribute of the $rootScope of the entire application!

See also

- The Isolate scope recipe goes into further details on directive scope management
Interfacing with a directive using isolate scope

Scopes and their inheritance is something you will frequently be dealing with in AngularJS applications. This is especially true in the context of directives, as they are subject to the scopes they are inserted into and, therefore, require careful management in order to prevent unexpected functionalities. Fortunately, AngularJS directives afford several robust tools that help manage visibility of and interaction with the surrounding scopes.

If a directive is not instructed to provide a new scope for itself, it will inherit the parent scope. In the case that this is not desirable behavior, you will need to create an isolate scope for that directive, and inside that isolate scope, you can define a whitelist of parent scope elements that the directive will need.

Getting ready

For this recipe, assume your directive exists inside the following setup:

(index.html)

```html
<div ng-app="myApp">
  <div ng-controller="MainCtrl">
    <div iso></div>
  </div>
</div>
```

(app.js)

```javascript
angular.module('myApp', [])
  .controller('MainCtrl', function ($log, $scope) {
    $scope.outerval = 'mydata';
    $scope.func = function () {
      $log.log('invoked!');
    };
  })
  .directive('iso', function () {
    return {};
  });
```
To declare a directive with an isolate scope, simply pass an empty object literal as the `scope` property:

(app.js)
```
.directive('iso', function () {
  return {
    scope: {}
  };}
});
```

With this, there will be no inheritance from the parent scope in `MainCtrl`, and the directive will be unable to use methods or variables in the parent scope.

If you want to pass a read-only value to the directive, you will use `@` inside the isolate scope declaration to indicate that a named attribute of the relevant HTML element contains a value that should be incorporated into the directive’s isolate scope. This can be done as follows:

(index.html)
```
<div ng-app="myApp">
  <div ng-controller="MainCtrl">
    <div>Outer: {{ outerval }}</div>
    <div iso myattr="{{ outerval }}"></div>
  </div>
</div>
```

(app.js)
```
.directive('iso', function () {
  return {
    template: 'Inner: {{ innerval }}',
    scope: {
      innerval: '@myattr'
    }
  };}
});
```

With this, the scope inside the directive now contains an `innerval` attribute with the value of `outerval` in the parent scope. AngularJS evaluates the expression string, and the result is provided to the directive’s scope. Setting the value of the variable does nothing to the parent scope or the attribute in the HTML; it is merely copied into the scope of the directive.
While this approach is useful, it doesn't involve data binding, which you have come to love in AngularJS, and it isn't all that more convenient than passing in a static string value. What is far more likely to be useful to you is a true whitelist of the data binding from the parent scope. This can be accomplished with the \= definition, as follows:

(index.html)

```html
<div ng-app="myApp">
  <div ng-controller="MainCtrl">
    <div>Outer: {{ outerval }}</div>
    <div iso myattr="outerval"></div>
  </div>
</div>
```

(app.js)

```javascript
.directive('iso', function () {
  return {
    template: 'Inner: {{ innerval }}',
    scope: {
      innerval: '='myattr'
    }
  };
});
```

Here, you are instructing the child directive scope to examine the parent controller scope, and bind the parent outerval attribute inside the child scope, aliased as the innerval attribute. Full data binding between scopes is supported, and all unnamed attributes and methods in the parent scope are ignored.
Taking a step further, methods can also be pulled down from the parent scope for use in the directive. In the same way that a model variable can be bound to the child scope, you can alias methods that are defined in the parent scope to be invoked from the child scope but are still in the parent scope context. This is accomplished with the & definition, as follows:

```
(index.html)

<div ng-app="myApp">
  <div ng-controller="MainCtrl">
    <div iso myattr="func()"></div>
  </div>
</div>

(app.js)

.directive('iso', function () {
  return {
    scope: {
      innerVal: '&myattr'
    },
    link: function(scope) {
      scope.innerVal();  // invoked!
    }
  };
});
```

[JSFiddle: http://jsfiddle.net/msfrisbie/1u24c4o8/]

Here, you are instructing the child directive to evaluate the expression passed to the myattr attribute within the context of the parent controller. In this case, the expression will invoke the func() method, but any valid AngularJS expression will also work. You can invoke it as you would invoke any other scope method, including parameters as required.
Maximizing AngularJS Directives

How it works...

Isolate scope is entirely managed within the `scope` attribute in the directive's returned definition object. Using `@`, `=`, and `&`, you are instructing the directive to ignore the scopes it would normally inherit, and only utilize data, variables, and methods that you have provided interfaces for instead.

There's more...

If the directive is designed as a specific modifier for an aspect of your application, you might find that using isolate scope isn't necessary. On the other hand, if you're building a reusable, monolithic component that can be reused across multiple applications, it is unlikely that the directive will be using the parent scope in which it is used. Hence, isolate scope will be significantly more useful.

See also

- The Recursive directives recipe utilizes the isolate scope to maintain inheritance and separation in a recursive DOM tree

Interaction between nested directives

AngularJS provides a useful structure that allows you to build channels of communication between directive siblings (within the same HTML element) or parents in the same DOM ancestry without having to rely on AngularJS events.

Getting ready

For this recipe, suppose that your application template includes the following:

```html
(index.html)

```
How to do it...

Inter-directive communication is accomplished with the `require` attribute, as follows:

```javascript
return {
  require: ['^parentDirective', '^siblingDirective'],
  link: function (scope, el, attrs, ctrls) {
    $log.log(ctrls);
    // logs array of in-order required controller objects
  }
};
```

Using the stringified directive names passed through `require`, AngularJS will examine the current and parent HTML elements that match the directive names. The controller objects of these directives will be returned in an array as the `ctrls` parameter in the original directive's `link` function.

These directives can expose methods as follows:

```javascript
(app.js)
angular.module('myApp', [])
  .directive('parentDirective', function ($log) {
    return {
      controller: function () {
        this.identify = function () {
          $log.log('Parent!');
        };
      }
    };
  });

angular.module('myApp', [])
  .directive('siblingDirective', function ($log) {
    return {
      controller: function () {
        this.identify = function () {
          $log.log('Sibling!');
        };
      }
    };
  });

angular.module('myApp', [])
  .directive('childDirective', function ($log) {
    return {
      require: ['^parentDirective', '^siblingDirective'],
      link: function (scope, el, attrs, ctrls) {
        ctrls[0].identify();
        // Parent!
      }
    };
  });
```
Maximizing AngularJS Directives

```javascript
ctrls[1].identify();
// Sibling!
});
});

[ JSFiddle: http://jsfiddle.net/msfrisbie/Lnxeyj60/ ]

How it works...

The childDirective fetches the requested controllers and passes them to the link function, which can use them as regular JavaScript objects. The order in which directives are defined is not important, but the controller objects will be returned in the order in which they are requested.

See also

- The Optional nested directive controllers recipe demonstrates how to handle a scenario where parent or sibling controllers might not be present

Optional nested directive controllers

The AngularJS construct that allows you to build channels of communication between directive siblings or parents in the same DOM ancestry also allows you to optionally require a directive controller of a sibling or parent.

Getting ready

Suppose that your application includes the following:

(index.html)

```html
<div ng-app="myApp">
  <div parent-directive>
    <div child-directive
      sibling-directive>
    </div>
  </div>
</div>
```
angular.module('myApp', [])
.directive('parentDirective', function ($log) {
  return {
    controller: function () {
      this.identify = function () {
        $log.log('Parent!');
      };
    }
  };
}).directive('siblingDirective', function ($log) {
  return {
    controller: function () {
      this.identify = function () {
        $log.log('Sibling!');
      };
    }
  };
}).directive('childDirective', function ($log) {
  return {
    require: [
      '^parentDirective',
      '^siblingDirective',
      '?missingDirective'
    ],
    link: function (scope, el, attrs, ctrls) {
      ctrls[0].identify();
      // Parent!
      ctrls[1].identify();
    }
});

How to do it...

Note that in index.html, the missingDirective is not present. A `?` prefixed to the require array element denotes an optional controller directive. This is shown in the following code:

(app.js)
.directive('childDirective', function ($log) {
  return {
    require: [
      '^parentDirective',
      '^siblingDirective',
      '?missingDirective'
    ],
    link: function (scope, el, attrs, ctrls) {
      ctrls[0].identify();
      // Parent!
      ctrls[1].identify();
    }
});
Maximizing AngularJS Directives

```javascript
// Sibling!
$log.log(ctrls[2]);
// null
```

If the controller exists, it will be served in the same fashion as the others. If not, the returned array will be a null value at the corresponding index.

### How it works...

An AngularJS controller is merely a JavaScript constructor function, and when `parentDirective` and `siblingDirective` are required, each directive returns their controller object. As you are using the controller object and not the controller scope, you must define your public controller methods on `this` instead of `$scope`. The `$scope` doesn't make sense in the context of a foreign directive—recall that the directive is in the process of being linked when all of this happens.

### Directive scope inheritance

When a directive is not instructed to create its own isolate scope, it will inherit the scope of whatever scope it exists inside.

### Getting ready

Suppose that you begin with the following skeleton application:

```html
(index.html - uncompiled)

<div ng-app="myApp">
  <div ng-controller="MainCtrl">
    <my-directive>
      <p>HTML template</p>
      <p>Scope from {{origin}}</p>
      <p>Overwritten? {{overwrite}}</p>
    </my-directive>
  </div>
</div>
```
How to do it...

The most basic setup is to have the directive scope inherit from the parent scope that will be used by the directive within the link function. This allows the directive to manipulate the parent scope. This can be done as follows:

(app.js)

.directive('myDirective', function () {
    return {
        restrict: 'E',
        link: function (scope) {
            scope.overwrite = !!scope.origin;
            scope.origin = 'link function';
        }
    };
});

This will compile into the following:

(index.html – compiled)

<my-directive>
<p>HTML template</p>
<p>Scope from link function</p>
<p>Overwritten? true</p>
</my-directive>

[JSFiddle: http://jsfiddle.net/msfrisbie/c3b3a38t/]
Maximizing AngularJS Directives

How it works...

There's nothing tricky going on here. The directive has no template, and the HTML inside it is subject to the modifications that the link function makes to the scope. As this does not use isolate scope and there is no transclusion, the parent scope is provided as the scope parameter, and the link function writes to the parent scope's models. The HTML output tells us that the template was rendered from our index.html markup, the link function was the last to modify the scope, and the link function overwrote the original values set up in the parent controller.

See also

- The Directive templating recipe examines how a directive can apply an external scope to a transplanted template
- The Isolate scope recipe gives details on how a directive can be decoupled from its parent scope
- The Directive transclusion recipe demonstrates how a directive handles the application of a scope to the interpolated existing nested content

Directive templating

Directives will frequently load HTML templates from outside their definition. When using them in an application, you will need to understand how to properly manage them, how they interact (if at all) with the directive's parent scope, and how they interact with the content nested inside them.

Getting ready

Suppose that you begin with the following skeleton application:

(index.html - uncompiled)

```html
<div ng-app="myApp">
  <div ng-controller="MainCtrl">
    <my-directive>
      Stuff inside
    </my-directive>
  </div>
</div>
```
How to do it...

Introduce a template to the directive as follows:

(index.html - uncompiled)

```html
<div ng-app="myApp">
  <div ng-controller="MainCtrl">
    <my-directive>
      Stuff inside
    </my-directive>
  </div>
</div>
```

```javascript
angular.module('myApp', [])
  .controller('MainCtrl', function ($scope) {
    $scope.overwrite = false;
    $scope.origin = 'parent controller';
  });
```

```javascript
.directive('myDirective', function() {
```

(app.js)

```javascript
angular.module('myApp', [])
  .controller('MainCtrl', function ($scope) {
    $scope.overwrite = false;
    $scope.origin = 'parent controller';
  })
  .directive('myDirective', function() {
```
return {
  restrict: 'E',
  replace: true,
  templateUrl: 'my-directive.html',
  link: function (scope) {
    scope.overwrite = !!scope.origin;
    scope.origin = 'link function';
  }
};

This snippet will compile the directive element into the following:

(index.html – compiled)

```
<dir>
  <p>Directive template</p>
  <p>Scope from link function</p>
  <p>Overwritten? true</p>
</dir>
```

[JSFiddle: http://jsfiddle.net/msfrisbie/cojb59b1/]

**How it works...**

The parent scope from MainCtrl is inherited by the directive and is provided as the `scope` parameter inside the directive's `link` function. The directive template is inserted to replace the `<my-directive>` tag and its contents, but the supplanting template HTML is still subject to the inherited scope. The `link` function is able to modify the parent scope as though it were the directive's own. In other words, the link scope and the controller scope are the same object in this example.

**See also**

- The *Directive scope inheritance* recipe goes over the basics that involve carrying the parent scope through a directive
- The *Isolate scope* recipe gives details on how a directive can be decoupled from its parent scope
- The *Directive transclusion* recipe demonstrates how a directive handles the application of a scope to the interpolated existing nested content
Isolate scope

Often, you will find that the inheritance of a directive's parent scope is undesirable somewhere in your application. To prevent inheritance and to create a blank slate scope for the directive, isolate scope is utilized.

Getting ready

Suppose that you begin with the following skeleton application:

(index.html - uncompiled)

```
<div ng-app="myApp">
  <div ng-controller="MainCtrl">
    <my-directive>
      Stuff inside
    </my-directive>
  </div>

  <script type="text/ng-template" id="my-directive.html">
    <div>
      <p>Directive template</p>
      <p>Scope from {{origin}}</p>
      <p>Overwritten? {{overwrite}}</p>
    </div>
  </script>
</div>
```

(app.js)

```
angular.module('myApp', [])
  .controller('MainCtrl', function ($scope) {
    $scope.overwrite = false;
    $scope.origin = 'parent controller';
  });
```

How to do it...

Assign an isolate scope to the directive with an empty object literal, as follows:

(app.js)

```
.directive('myDirective', function() {
  return {
    templateUrl: 'my-directive.html',
    restrict: 'E',
    scope: { overwrite: '=' },
    replace: true,
    transclude: true,
    controller: function($scope, $element, $attrs) {
      $scope.origin = $attrs.myOrigin;
    }
  }
});
```
Maximizing AngularJS Directives

```javascript
templateUrl: 'my-directive.html',
replace: true,
scope: {},
link: function (scope) {
  scope.overwrite = !!scope.origin;
  scope.origin = 'link function';
}
});
```

This will compile into the following:

(index.html – compiled)

```html
<div>
  <p>Directive template</p>
  <p>Scope from link function</p>
  <p>Overwritten? false</p>
</div>
```

[JSFiddle: http://jsfiddle.net/msfrisbie/a2vmuhd3/]

**How it works...**

The directive creates its own scope and performs the modifications on the scope instead of performing them inside the `link` function. The parent scope is unchanged and obscured from inside the directive's `link` function.

**See also**

- The *Directive scope inheritance* recipe goes over the basics that involve carrying the parent scope through a directive
- The *Directive templating* recipe examines how a directive can apply an external scope to an interpolated template
- The *Directive transclusion* recipe demonstrates how a directive handles the application of a scope to the interpolated existing nested content
Directive transclusion

Transclusion on its own is a relatively simple construct in AngularJS. This simplicity becomes muddied when mixed with the complexity of directives and scope inheritance. Directive transclusion is frequently used when the directive either needs to inherit from the parent scope, manage nested HTML, or both.

How to do it...

Assemble all the pieces required to use transclusion. This is shown here:

(index.html - uncompiled)

```html
<div ng-app="myApp">
  <div ng-controller="MainCtrl">
    <my-directive>
      <p>HTML template</p>
      <p>Scope from {{origin}}</p>
      <p>Overwritten? {{overwrite}}</p>
    </my-directive>
  </div>
</div>
```

(app.js)

```javascript
angular.module('myApp', [])
  .controller('MainCtrl', function ($scope) {
    $scope.overwrite = false;
    $scope.origin = 'parent controller';
  })
  .directive('myDirective', function() {
    return {
      restrict: 'E',
      templateUrl: 'my-directive.html',
      scope: {},
      transclude: true,
      link: function (scope) {
        scope.overwrite = !!scope.origin;
      }
    }
  });
```
scope.origin = 'link function';
}
);
});

This will compile into the following:

(index.html – compiled)

<p>HTML template</p>
<p>Scope from parent controller</p>
<p>Overwritten? false</p>

In the directive's template, the location of ng-transclude informs $compile that the directive's original HTML contents are to replace the contents of the specified element. Furthermore, using transclusion means that the parent scope will continue to be in the directive to be used for the interpolated HTML.

To see the main reason to use transclusion more clearly, modify the my-directive.html directive template slightly in order to see the results side by side. This can be done as follows:

(index.html - uncompiled)

<script type="text/ng-template" id="my-directive.html">
    <ng-transclude></ng-transclude>
    <hr />
    <p>Directive template</p>
    <p>Scope from {{origin}}</p>
    <p>Overwritten? {{overwrite}}</p>
</script>

This will compile into the following:

(index.html - compiled)

<p>HTML template</p>
<p>Scope from parent controller</p>
<p>Overwritten? false</p>
<hr />
<p>Directive template</p>
<p>Scope from link function</p>
<p>Overwritten? false</p>

[JSFiddle: http://jsfiddle.net/msfrisbie/1alld3mk/]
Chapter 1

How it works...

It should now be apparent exactly what is going on inside the directive that uses transclusion. The directive’s template is subject to the link function (which necessarily uses the isolate scope), and the original wrapped HTML template maintains its relationship with the parent scope without the directive interfering.

See also

- The Directive scope inheritance recipe goes over the basics that involve carrying the parent scope through a directive
- The Directive templating recipe examines how a directive can apply external scope to an interpolated template
- The Isolate scope recipe details how a directive can be decoupled from its parent scope

Recursive directives

The power of directives can also be effectively applied when consuming data in a more unwieldy format. Consider the case in which you have a JavaScript object that exists in some sort of recursive tree structure. The view that you will generate for this object will also reflect its recursive nature and will have nested HTML elements that match the underlying data structure.

Getting ready

Suppose you had a recursive data object in your controller as follows:

(app.js)

```javascript
angular.module('myApp', [])
.controller('MainCtrl', function($scope) {
  $scope.data = {
    text: 'Primates',
    items: [
      {
        text: 'Anthropoidea',
        items: [
          {
            text: 'New World Anthropoids'
          },
          {
            text: 'Old World Anthropoids",
          }
        ]
      }
    ]
  }
})
```
How to do it...

As you might imagine, iteratively constructing a view or only partially using directives to accomplish this will become extremely messy very quickly. Instead, it would be better if you were able to create a directive that would seamlessly break apart the data recursively, and define and render the sub-HTML fragments cleanly. By cleverly using directives and the $compile service, this exact directive functionality is possible.

The ideal directive in this scenario will be able to handle the recursive object without any additional parameters or outside assistance in parsing and rendering the object. So, in the main view, your directive will look something like this:

```html
<recursive value="nestedObject"></recursive>
```

The directive is accepting an isolate scope = binding to the parent scope object, which will remain structurally identical as the directive descends through the recursive object.
The $compile service

You will need to inject the $compile service in order to make the recursive directive work. The reason for this is that each level of the directive can instantiate directives inside it and convert them from an uncompiled template to real DOM material.

The angular.element() method

The angular.element() method can be thought of as the jQuery $() equivalent. It accepts a string template or DOM fragment and returns a jqLite object that can be modified, inserted, or compiled for your purposes. If the jQuery library is present when the application is initialized, AngularJS will use that instead of jqLite. If you use the AngularJS template cache, retrieved templates will already exist as if you had called the angular.element() method on the template text.

The $templateCache

Inside a directive, it’s possible to create a template using angular.element() and a string of HTML similar to an underscore.js template. However, it’s completely unnecessary and quite unwieldy to use compared to AngularJS templates. When you declare a template and register it with AngularJS, it can be accessed through the injected $templateCache, which acts as a key-value store for your templates.

The recursive template is as follows:

```html
<script type="text/ng-template" id="recursive.html">
  <span>{{ val.text }}</span>
  <button ng-click="delSubtree()">delete</button>
  <ul ng-if="isParent" style="margin-left:30px">
    <li ng-repeat="item in val.items">
      <tree val="item" parent-data="val.items"></tree>
    </li>
  </ul>
</script>
```

The `span` and `button` elements are present at each instance of a node, and they present the data at that node as well as an interface to the click event (which we will define in a moment) that will destroy it and all its children.

Following these, the conditional `<ul>` element renders only if the `isParent` flag is set in the scope, and it repeats through the `items` array, recursing the child data and creating new instances of the directive. Here, you can see the full template definition of the directive:

```html
<tree val="item" parent-data="val.items"></tree>
```
Maximizing AngularJS Directives

Not only does the directive take a `val` attribute for the local node data, but you can also see its `parent-data` attribute, which is the point of scope indirection that allows the tree structure. To make more sense of this, examine the following directive code:

```
(app.js)

.directive('tree', function($compile, $templateCache) {
  return {
    restrict: 'E',
    scope: {
      val: '=',
      parentData: '='
    },
    link: function(scope, el, attrs) {
      scope.isParent = angular.isArray(scope.val.items)
      scope.delSubtree = function() {
        if(scope.parentData) {
          scope.parentData.splice(
            scope.parentData.indexOf(scope.val),
            1
          );
        }
        scope.val={};
      }
      el.replaceWith(  
        $compile($templateCache.get('recursive.html'))(scope)
      );
    }
  };
});
```

With all of this, if you provide the recursive directive with the data object provided at the beginning of this recipe, it will result in the following (presented here without the auto-added AngularJS comments and directives):

```
(index.html – uncompiled)

<div ng-app="myApp">
  <div ng-controller="MainCtrl">
    <tree val="data"></tree>
  </div>
  <script type="text/ng-template" id="recursive.html">
  </script>
</div>
```

40
The recursive nature of the directive templates enables nesting, and when compiled using the recursive data object located in the wrapping controller, it will compile into the following HTML:

(index.html - compiled)

```html
<div ng-controller="MainController">  
  <span>Primates</span>
  <button ng-click="delSubtree()">delete</button>
  <ul ng-if="isParent" style="margin-left:30px">
    <li ng-repeat="item in val.items">
      <span>Anthropoidea</span>
      <button ng-click="delSubtree()">delete</button>
      <ul ng-if="isParent" style="margin-left:30px">
        <li ng-repeat="item in val.items">
          <span>New World Anthropoids</span>
          <button ng-click="delSubtree()">delete</button>
          <ul ng-if="isParent" style="margin-left:30px">
            <li ng-repeat="item in val.items">
              <span>Apes</span>
              <button ng-click="delSubtree()">delete</button>
              <ul ng-if="isParent" style="margin-left:30px">
                <li ng-repeat="item in val.items">
                  <span>Lesser Apes</span>
                  <button ng-click="delSubtree()">delete</button>
                </li>
                <li ng-repeat="item in val.items">
                  <span>Greater Apes</span>
                  <button ng-click="delSubtree()">delete</button>
                </li>
              </ul>
            </li>
            <li ng-repeat="item in val.items">
              <span>Old World Anthropoids</span>
              <button ng-click="delSubtree()">delete</button>
            </li>
          </ul>
        </li>
      </ul>
    </li>
  </ul>
</div>
```
Maximizing AngularJS Directives

```html
<ul>
  <li ng-repeat="item in val.items">
    <span>Monkeys</span>  
    <button ng-click="delSubtree()">delete</button>
  </li>
</ul>
<ul>
  <li ng-repeat="item in val.items">
    <span>Prosimii</span>  
    <button ng-click="delSubtree()">delete</button>
  </li>
</ul>
</div>

[JSFiddle: http://jsfiddle.net/msfrisbie/ka46yx4u/]

How it works...

The definition of the isolate scope through the nested directives described in the previous section allows all or part of the recursive objects to be bound through parentData to the appropriate directive instance, all the while maintaining the nested connectedness afforded by the directive hierarchy. When a parent node is deleted, the lower directives are still bound to the data object and the removal propagates through cleanly.

The meatiest and most important part of this directive is, of course, the link function. Here, the link function determines whether the node has any children (which simply checks for the existence of an array in the local data node) and declares the deleting method, which simply removes the relevant portion from the recursive object and cleans up the local node. Up until this point, there haven’t been any recursive calls, and there shouldn’t need to be. If your directive is constructed correctly, AngularJS data binding and inherent template management will take care of the template cleanup for you. This, of course, leads into the final line of the link function, which is broken up here for readability:

```javascript
el.replaceWith(
  $compile(
    $templateCache.get('recursive.html')
  )(scope)
);
```
Recall that in a **link** function, the second parameter is the jqLite-wrapped DOM object that the directive is linking—here, the `<tree>` element. This exposes to you a subset of jQuery object methods, including `replaceWith()`, which you will use here. The top-level instance of the directive will be replaced by the recursively-defined template, and this will carry down through the tree.

At this point, you should have an idea of how the recursive structure is coming together. The element parameter needs to be replaced with a recursively-compiled template, and for this, you will employ the `$compile` service. This service accepts a template as a parameter and returns a function that you will invoke with the current scope inside the directive’s **link** function. The template is retrieved from `$templateCache` by the `recursive.html` key, and then it’s compiled. When the compiler reaches the nested `<tree>` directive, the recursive directive is realized all the way down through the data in the recursive object.

**There’s more…**

This recipe demonstrates the power of constructing a directive to convert a complex data object into a large DOM object. Relevant portions can be broken into individual templates, handled with distributed directive logic, and combined together in an elegant fashion to maximize modularity and reusability.

**See also**

- The *Optional nested directive controllers* recipe covers vertical communication between directives through their controller objects
Where to buy this book
You can buy AngularJS Web Application Development Cookbook from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.