Creating Custom Directives

Note: this guide is targeted towards developers who are already familiar with AngularJS basics. If you're just getting started, we recommend the tutorial (tutorial/) first. If you're looking for the **directives API**, we recently moved it to `$compile` (api/ng/service/$compile).

This document explains when you'd want to create your own directives in your AngularJS app, and how to implement them.

What are Directives?

At a high level, directives are markers on a DOM element (such as an attribute, element name, comment or CSS class) that tell AngularJS's **HTML compiler** (`$compile` (api/ng/service/$compile)) to attach a specified behavior to that DOM element (e.g. via event listeners), or even to transform the DOM element and its children.

Angular comes with a set of these directives built-in, like `ngBind` , `ngModel` , and `ngClass` . Much like you create controllers and services, you can create your own directives for Angular to use. When Angular bootstraps (guide/bootstrap) your application, the HTML compiler (guide/compiler) traverses the DOM matching directives against the DOM elements.

What does it mean to "compile" an HTML template? For AngularJS, "compilation" means attaching directives to the HTML to make it interactive. The reason we use the term "compile" is that the recursive process of attaching directives mirrors the process of compiling

source code in compiled programming languages (http://en.wikipedia.org/wiki/Compiled_languages).

Matching Directives

Before we can write a directive, we need to know how Angular's HTML compiler (guide/compiler) determines when to use a given directive.

Similar to the terminology used when an element **matches** a selector (https://developer.mozilla.org/en-US/docs/Web/API/Element.matches), we say an element **matches** a directive when the directive is part of its declaration.

In the following example, we say that the `<input>` element **matches** the `ngModel` directive

```
<input ng-model="foo">
```

The following `<input>` element also **matches** `ngModel` :

```
<input data-ng-model="foo">
```

And the following element **matches** the `person` directive:

```
<person>{{name}}</person>
```

Normalization

Angular **normalizes** an element's tag and attribute name to determine which elements match which directives. We typically refer to directives by their case-sensitive camelCase (http://en.wikipedia.org/wiki/CamelCase) **normalized** name (e.g. `ngModel`). However, since HTML is case-insensitive, we refer to directives in the DOM by lower-case forms, typically using dash-delimited (http://en.wikipedia.org/wiki/Letter_case#Computers) attributes on DOM elements (e.g. `ng-model`).

The **normalization** process is as follows:

1. Strip `x-` and `data-` from the front of the element/attributes.
2. Convert the `:` , `-` , or `_` -delimited name to `camelCase` .

For example, the following forms are all equivalent and match the `ngBind` (api/ng/directive/ngBind) directive:

index.html () script.js () protractor.js () Edit in Plunker

```
<div ng-controller="Controller">
  Hello <input ng-model='name'> <hr/>
  <span ng-bind="name"></span> <br/>
  <span ng:bind="name"></span> <br/>
  <span ng_bind="name"></span> <br/>
  <span data-ng-bind="name"></span> <br/>
  <span x-ng-bind="name"></span> <br/>
</div>
```

Hello Max Karl Ernst Ludwig Planc

Max Karl Ernst Ludwig Planck (April 23, 1858 – October 4, 1947)
Max Karl Ernst Ludwig Planck (April 23, 1858 – October 4, 1947)
Max Karl Ernst Ludwig Planck (April 23, 1858 – October 4, 1947)
Max Karl Ernst Ludwig Planck (April 23, 1858 – October 4, 1947)
Max Karl Ernst Ludwig Planck (April 23, 1858 – October 4, 1947)

Best Practice: Prefer using the dash-delimited format (e.g. `ng-bind` for `ngBind`). If you want to use an HTML validating tool, you can instead use the `data` -prefixed version (e.g. `data-ng-bind` for `ngBind`). The other forms shown above are accepted for legacy reasons but we advise you to avoid them.

Directive types

`$compile` can match directives based on element names, attributes, class names, as well as comments.

All of the Angular-provided directives match attribute name, tag name, comments, or class name. The following demonstrates the various ways a directive (`myDir` in this case) can be referenced from within a template:

```
<my-dir></my-dir>
<span my-dir="exp"></span>
<!-- directive: my-dir exp -->
<span class="my-dir: exp;"></span>
```

Best Practice: Prefer using directives via tag name and attributes over comment and class names. Doing so generally makes it easier to determine what directives a given element matches.

Best Practice: Comment directives were commonly used in places where the DOM API limits the ability to create directives that spanned multiple elements (e.g. inside `<table>` elements). AngularJS 1.2 introduces `ng-repeat-start` and `ng-repeat-end` (api/ng/directive/ngRepeat) as a better solution to this problem. Developers are encouraged to use this over custom comment directives when possible.

Creating Directives

First let's talk about the API for registering directives (api/ng/provider/$compileProvider#directive). Much like controllers, directives are registered on modules. To register a directive, you use the `module.directive` API. `module.directive` takes the normalized (guide/directive#matching-directives) directive name followed by a **factory function.** This factory function should return an object with the different options to tell `$compile` how the directive should behave when matched.

The factory function is invoked only once when the compiler (api/ng/service/$compile) matches the directive for the first time. You can perform any initialization work here. The function is invoked using $injector.invoke (api/auto/service/$injector#invoke) which makes it injectable just like a controller.

Best Practice: Prefer using the definition object over returning a function.

We'll go over a few common examples of directives, then dive deep into the different options and compilation process.

Best Practice: In order to avoid collisions with some future standard, it's best to prefix your own directive names. For instance, if you created a `<carousel>` directive, it would be problematic if HTML7 introduced the same element. A two or three letter prefix (e.g. `btfCarousel`) works well. Similarly, do not prefix your own directives with `ng` or they might conflict with directives included in a future version of Angular.

For the following examples, we'll use the prefix `my` (e.g. `myCustomer`).

Template-expanding directive

Let's say you have a chunk of your template that represents a customer's information. This template is repeated many times in your code. When you change it in one place, you have to change it in several others. This is a good opportunity to use a directive to simplify your template.

Let's create a directive that simply replaces its contents with a static template:

script.js () index.html () Edit in Plunker

```
angular.module('docsSimpleDirective', [])
.controller('Controller', ['$scope', function($scope) {
  $scope.customer = {
    name: 'Naomi',
    address: '1600 Amphitheatre'
  };
}])
.directive('myCustomer', function() {
  return {
    template: 'Name: {{customer.name}} Address: {{customer.address}}'
  };
});
```

Name: Naomi Address: 1600 Amphitheatre

Notice that we have bindings in this directive. After `$compile` compiles and links `<div my-customer></div>`, it will try to match directives on the element's children. This means you can compose directives of other directives. We'll see how to do that in an example (guide/directive#creating-directives-that-communicate) below.

In the example above we in-lined the value of the **`template`** option, but this will become annoying as the size of your template grows.

Best Practice: Unless your template is very small, it's typically better to break it apart into its own HTML file and load it with the `templateUrl` option.

If you are familiar with `ngInclude`, `templateUrl` works just like it. Here's the same example using `templateUrl` instead:

script.js () | index.html () | my-customer.html () | Edit in Plunker

```
angular.module('docsTemplateUrlDirective', [])
.controller('Controller', ['$scope', function($scope) {
  $scope.customer = {
    name: 'Naomi',
    address: '1600 Amphitheatre'
  };
}])
.directive('myCustomer', function() {
  return {
    templateUrl: 'my-customer.html'
  };
});
```

Name: Naomi Address: 1600 Amphitheatre

`templateUrl` can also be a function which returns the URL of an HTML template to be loaded and used for the directive. Angular will call the `templateUrl` function with two parameters: the element that the directive was called on, and an `attr` object associated with that element.

Note: You do not currently have the ability to access scope variables from the `templateUrl` function, since the template is requested before the scope is initialized.

script.js () index.html () customer-name.html () customer-address.html ()

Edit in Plunker

```
angular.module('docsTemplateUrlDirective', [])
.controller('Controller', ['$scope', function($scope) {
  $scope.customer = {
    name: 'Naomi',
    address: '1600 Amphitheatre'
  };
}])
.directive('myCustomer', function() {
  return {
    templateUrl: function(elem, attr){
      return 'customer-'+attr.type+'.html';
    }
  };
});
```

Name: Naomi
Address: 1600 Amphitheatre

Note: When you create a directive, it is restricted to attribute and elements only by default. In order to create directives that are triggered by class name, you need to use the `restrict` option.

The `restrict` option is typically set to:

- `'A'` - only matches attribute name
- `'E'` - only matches element name
- `'C'` - only matches class name
- `'M'` - only matches comment

These restrictions can all be combined as needed:

- `'AEC'` - matches either attribute or element or class name

Let's change our directive to use `restrict: 'E'`:

script.js () | index.html () | my-customer.html ()

Edit in Plunker

```
angular.module('docsRestrictDirective', [])
.controller('Controller', ['$scope', function($scope) {
  $scope.customer = {
    name: 'Naomi',
    address: '1600 Amphitheatre'
  };
}])
.directive('myCustomer', function() {
  return {
    restrict: 'E',
    templateUrl: 'my-customer.html'
  };
});
```

Name: Naomi Address: 1600 Amphitheatre

For more on the `restrict` (api/ng/service/$compile#directive-definition-object) property, see the API docs (api/ng/service/$compile#directive-definition-object).

When should I use an attribute versus an element? Use an element when you are creating a component that is in control of the template. The common case for this is when you are creating a Domain-Specific Language for parts of your template. Use an attribute when you are decorating an existing element with new functionality.

Using an element for the `myCustomer` directive is clearly the right choice because you're not decorating an element with some "customer" behavior; you're defining the core behavior of the element as a customer component.

Isolating the Scope of a Directive

Our `myCustomer` directive above is great, but it has a fatal flaw. We can only use it once within a given scope.

In its current implementation, we'd need to create a different controller each time in order to re-use such a directive:

script.js () index.html () my-customer.html () Edit in Plunker

```
angular.module('docsScopeProblemExample', [])
.controller('NaomiController', ['$scope', function($scope) {
  $scope.customer = {
    name: 'Naomi',
    address: '1600 Amphitheatre'
  };
}])
.controller('IgorController', ['$scope', function($scope) {
  $scope.customer = {
    name: 'Igor',
    address: '123 Somewhere'
  };
}])
.directive('myCustomer', function() {
  return {
    restrict: 'E',
    templateUrl: 'my-customer.html'
  };
});
```

Name: Naomi Address: 1600 Amphitheatre

Name: Igor Address: 123 Somewhere

This is clearly not a great solution.

What we want to be able to do is separate the scope inside a directive from the scope outside, and then map the outer scope to a directive's inner scope. We can do this by creating what we call an **isolate scope**. To do this, we can use a directive's `scope` (api/ng/service/$compile#-scope-) option:

script.js () | index.html () | my-customer-iso.html () | Edit in Plunker

```
angular.module('docsIsolateScopeDirective', [])
.controller('Controller', ['$scope', function($scope) {
  $scope.naomi = { name: 'Naomi', address: '1600 Amphitheatre' };
  $scope.igor = { name: 'Igor', address: '123 Somewhere' };
}])
.directive('myCustomer', function() {
  return {
    restrict: 'E',
    scope: {
      customerInfo: '=info'
    },
    templateUrl: 'my-customer-iso.html'
  };
});
```

Name: Naomi Address: 1600 Amphitheatre

Name: Igor Address: 123 Somewhere

Looking at `index.html`, the first `<my-customer>` element binds the `info` attribute to `naomi`, which we have exposed on our controller's scope. The second binds `info` to `igor`.

Let's take a closer look at the scope option:

```
//...
scope: {
  customerInfo: '=info'
},
//...
```

The **scope option** is an object that contains a property for each isolate scope binding. In this case it has just one property:

- Its name (`customerInfo`) corresponds to the directive's **isolate scope** property `customerInfo`.
- Its value (`=info`) tells `$compile` to bind to the `info` attribute.

Note: These `=attr` attributes in the `scope` option of directives are normalized just like directive names. To bind to the attribute in `<div bind-to-this="thing">`, you'd specify a binding of `=bindToThis`.

For cases where the attribute name is the same as the value you want to bind to inside the directive's scope, you can use this shorthand syntax:

```
...
scope: {
  // same as '=customer'
  customer: '='
},
...
```

Besides making it possible to bind different data to the scope inside a directive, using an isolated scope has another effect.

We can show this by adding another property, `vojta`, to our scope and trying to access it from within our directive's template:

script.js () | index.html () | my-customer-plus-vojta.html () | Edit in Plunker

```
angular.module('docsIsolationExample', [])
.controller('Controller', ['$scope', function($scope) {
  $scope.naomi = { name: 'Naomi', address: '1600 Amphitheatre' };
  $scope.vojta = { name: 'Vojta', address: '3456 Somewhere Else' };
}])
.directive('myCustomer', function() {
  return {
    restrict: 'E',
    scope: {
      customerInfo: '=info'
    },
    templateUrl: 'my-customer-plus-vojta.html'
  };
});
```

Name: Naomi Address: 1600 Amphitheatre

Name: Address:

Notice that `{{vojta.name}}` and `{{vojta.address}}` are empty, meaning they are undefined. Although we defined `vojta` in the controller, it's not available within the directive.

As the name suggests, the **isolate scope** of the directive isolates everything except models that you've explicitly added to the `scope: {}` hash object. This is helpful when building reusable components because it prevents a component from changing your model state except for the models that you explicitly pass in.

Note: Normally, a scope prototypically inherits from its parent. An isolated scope does not. See the "Directive Definition Object - scope" (api/ng/service/$compile#directive-definition-object) section for more information about isolate scopes.

Best Practice: Use the `scope` option to create isolate scopes when making components that you want to reuse throughout your app.

Creating a Directive that Manipulates the DOM

In this example we will build a directive that displays the current time. Once a second, it updates the DOM to reflect the current time.

Directives that want to modify the DOM typically use the `link` option to register DOM listeners as well as update the DOM. It is executed after the template has been cloned and is where directive logic will be put.

`link` takes a function with the following signature, `function link(scope, element, attrs, controller, transcludeFn) { ... }`, where:

- `scope` is an Angular scope object.
- `element` is the jqLite-wrapped element that this directive matches.
- `attrs` is a hash object with key-value pairs of normalized attribute names and their corresponding attribute values.
- `controller` is the directive's required controller instance(s) or its own controller (if any). The exact value depends on the directive's require property.
- `transcludeFn` is a transclude linking function pre-bound to the correct transclusion scope.

> For more details on the `link` option refer to the `$compile` API (api/ng/service/$compile#-link-) page.

In our `link` function, we want to update the displayed time once a second, or whenever a user changes the time formatting string that our directive binds to. We will use the `$interval` service to call a handler on a regular basis. This is easier than using `$timeout` but also works better with end-to-end testing, where we want to ensure that all `$timeout`s have completed before completing the test. We also want to remove the `$interval` if the directive is deleted so we don't introduce a memory leak.

script.js () **index.html ()** **Edit in Plunker**

```
angular.module('docsTimeDirective', [])
.controller('Controller', ['$scope', function($scope) {
  $scope.format = 'M/d/yy h:mm:ss a';
}])
.directive('myCurrentTime', ['$interval', 'dateFilter', function($interval, dateFilter) {

  function link(scope, element, attrs) {
    var format,
        timeoutId;

    function updateTime() {
      element.text(dateFilter(new Date(), format));
    }

    scope.$watch(attrs.myCurrentTime, function(value) {
      format = value;
      updateTime();
    });

    element.on('$destroy', function() {
      $interval.cancel(timeoutId);
    });

    // start the UI update process; save the timeoutId for canceling
    timeoutId = $interval(function() {
      updateTime(); // update DOM
    }, 1000);
  }

  return {
    link: link
  };
}]);
```

Date format: M/d/yy h:mm:ss a

Current time is: 2/6/16 6:59:06 AM

There are a couple of things to note here. Just like the `module.controller` API, the function argument in `module.directive` is dependency injected. Because of this, we can use `$interval` and `dateFilter` inside our directive's `link` function.

We register an event `element.on('$destroy', ...)`. What fires this `$destroy` event?

There are a few special events that AngularJS emits. When a DOM node that has been compiled with Angular's compiler is destroyed, it emits a `$destroy` event. Similarly, when an AngularJS scope is destroyed, it broadcasts a `$destroy` event to listening scopes.

By listening to this event, you can remove event listeners that might cause memory leaks. Listeners registered to scopes and elements are automatically cleaned up when they are destroyed, but if you registered a listener on a service, or registered a listener on a DOM node that isn't being deleted, you'll have to clean it up yourself or you risk introducing a memory leak.

Best Practice: Directives should clean up after themselves. You can use `element.on('$destroy', ...)` or `scope.$on('$destroy', ...)` to run a clean-up function when the directive is removed.

Creating a Directive that Wraps Other Elements

We've seen that you can pass in models to a directive using the isolate scope, but sometimes it's desirable to be able to pass in an entire template rather than a string or an object. Let's say that we want to create a "dialog box" component. The dialog box should be able to wrap any arbitrary content.

To do this, we need to use the `transclude` option.

script.js () index.html () my-dialog.html () Edit in Plunker

```
angular.module('docsTransclusionDirective', [])
.controller('Controller', ['$scope', function($scope) {
  $scope.name = 'Tobias';
}])
.directive('myDialog', function() {
  return {
    restrict: 'E',
    transclude: true,
    scope: {},
    templateUrl: 'my-dialog.html'
  };
});
```

Check out the contents, Tobias!

What does this `transclude` option do, exactly? `transclude` makes the contents of a directive with this option have access to the scope **outside** of the directive rather than inside.

To illustrate this, see the example below. Notice that we've added a `link` function in `script.js` that redefines `name` as `Jeff`. What do you think the `{{name}}` binding will resolve to now?

script.js () index.html () my-dialog.html ()

Edit in Plunker

```
angular.module('docsTransclusionExample', [])
.controller('Controller', ['$scope', function($scope) {
  $scope.name = 'Tobias';
}])
.directive('myDialog', function() {
  return {
    restrict: 'E',
    transclude: true,
    scope: {},
    templateUrl: 'my-dialog.html',
    link: function (scope) {
      scope.name = 'Jeff';
    }
  };
});
```

Check out the contents, Tobias!

Ordinarily, we would expect that `{{name}}` would be **`Jeff`**. However, we see in this example that the `{{name}}` binding is still **`Tobias`**.

The `transclude` option changes the way scopes are nested. It makes it so that the **contents** of a transcluded directive have whatever scope is outside the directive, rather than whatever scope is on the inside. In doing so, it gives the contents access to the outside scope.

Note that if the directive did not create its own scope, then `scope` in `scope.name = 'Jeff'` would reference the outside scope and we would see `Jeff` in the output.

This behavior makes sense for a directive that wraps some content, because otherwise you'd have to pass in each model you wanted to use separately. If you have to pass in each model that you want to use, then you can't really have arbitrary contents, can you?

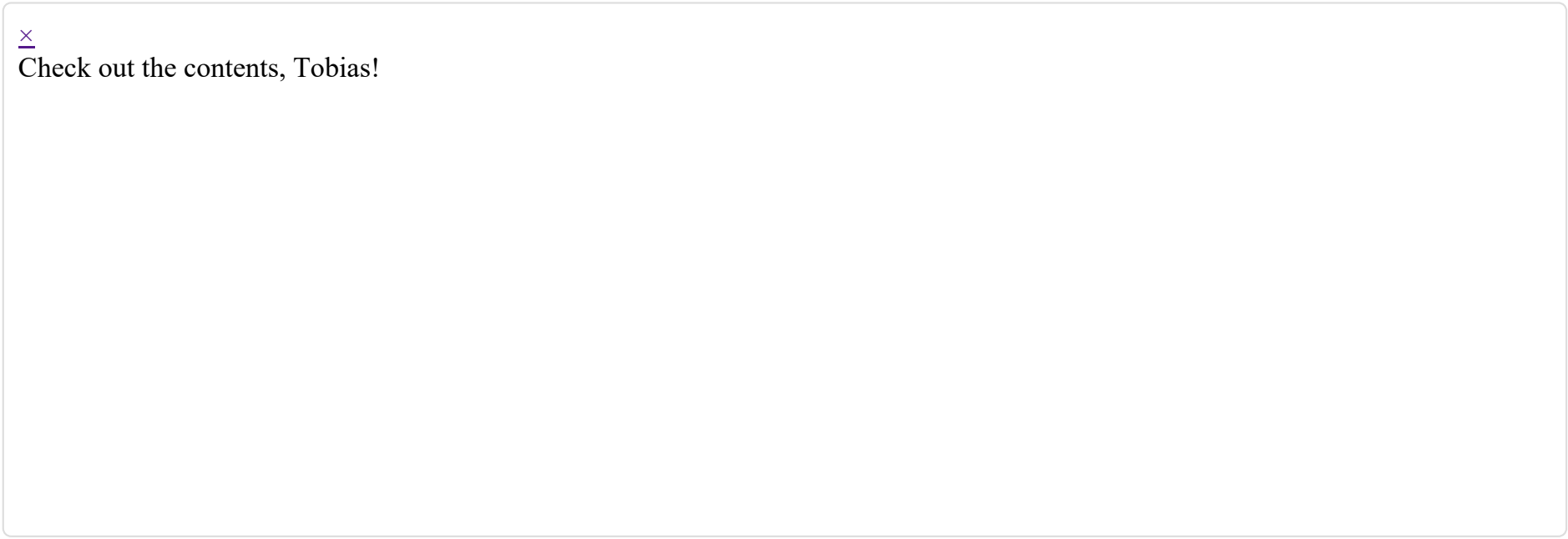
Best Practice: only use `transclude: true` when you want to create a directive that wraps arbitrary content.

Next, we want to add buttons to this dialog box, and allow someone using the directive to bind their own behavior to it.

script.js () | index.html () | my-dialog-close.html ()

Edit in Plunker

```
angular.module('docsIsoFnBindExample', [])
.controller('Controller', ['$scope', '$timeout', function($scope, $timeout) {
  $scope.name = 'Tobias';
  $scope.message = '';
  $scope.hideDialog = function (message) {
    $scope.message = message;
    $scope.dialogIsHidden = true;
    $timeout(function () {
      $scope.message = '';
      $scope.dialogIsHidden = false;
    }, 2000);
  };
}])
.directive('myDialog', function() {
  return {
    restrict: 'E',
    transclude: true,
    scope: {
      'close': '&onClose'
    },
    templateUrl: 'my-dialog-close.html'
  };
});
```



We want to run the function we pass by invoking it from the directive's scope, but have it run in the context of the scope where it's registered.

We saw earlier how to use `=attr` in the `scope` option, but in the above example, we're using `&attr` instead. The `&` binding allows a directive to trigger evaluation of an expression in the context of the original scope, at a specific time. Any legal expression is allowed, including an expression which contains a function call. Because of this, `&` bindings are ideal for binding callback functions to directive behaviors.

When the user clicks the `x` in the dialog, the directive's `close` function is called, thanks to `ng-click`. This call to `close` on the isolated scope actually evaluates the expression `hideDialog(message)` in the context of the original scope, thus running **`Controller`** 's `hideDialog` function.

Often it's desirable to pass data from the isolate scope via an expression to the parent scope, this can be done by passing a map of local variable names and values into the expression wrapper function. For example, the `hideDialog` function takes a message to display when the dialog is hidden. This is specified in the directive by calling `close({message: 'closing for now'})`. Then the local variable `message` will be available within the `on-close` expression.

Best Practice: use `&attr` in the `scope` option when you want your directive to expose an API for binding to behaviors.

Creating a Directive that Adds Event Listeners

Previously, we used the `link` function to create a directive that manipulated its DOM elements. Building upon that example, let's make a directive that reacts to events on its elements.

For instance, what if we wanted to create a directive that lets a user drag an element?

script.js () index.html () Edit in Plunker

```
angular.module('dragModule', [])
.directive('myDraggable', ['$document', function($document) {
  return {
    link: function(scope, element, attr) {
      var startX = 0, startY = 0, x = 0, y = 0;

      element.css({
       position: 'relative',
       border: '1px solid red',
       backgroundColor: 'lightgrey',
       cursor: 'pointer'
      });

      element.on('mousedown', function(event) {
        // Prevent default dragging of selected content
        event.preventDefault();
        startX = event.pageX - x;
        startY = event.pageY - y;
        $document.on('mousemove', mousemove);
        $document.on('mouseup', mouseup);
      });

      function mousemove(event) {
        y = event.pageY - startY;
        x = event.pageX - startX;
        element.css({
          top: y + 'px',
          left:  x + 'px'
        });
      }

      function mouseup() {
        $document.off('mousemove', mousemove);
        $document.off('mouseup', mouseup);
      }
    }
  };
```

```
}]);
```

Creating Directives that Communicate

You can compose any directives by using them within templates.

Sometimes, you want a component that's built from a combination of directives.

Imagine you want to have a container with tabs in which the contents of the container correspond to which tab is active.

script.js () | index.html () | my-tabs.html () | my-pane.html () | Edit in Plunker

```
angular.module('docsTabsExample', [])
.directive('myTabs', function() {
  return {
    restrict: 'E',
    transclude: true,
    scope: {},
    controller: ['$scope', function($scope) {
      var panes = $scope.panes = [];

      $scope.select = function(pane) {
        angular.forEach(panes, function(pane) {
          pane.selected = false;
        });
        pane.selected = true;
      };

      this.addPane = function(pane) {
        if (panes.length === 0) {
          $scope.select(pane);
        }
        panes.push(pane);
      };
    }],
    templateUrl: 'my-tabs.html'
  };
})
.directive('myPane', function() {
  return {
    require: '^^myTabs',
    restrict: 'E',
    transclude: true,
    scope: {
      title: '@'
    },
    link: function(scope, element, attrs, tabsCtrl) {
      tabsCtrl.addPane(scope);
    },
```

```
    templateUrl: 'my-pane.html'
  };
});
```

- Hello
- World

Hello

Lorem ipsum dolor sit amet

The `myPane` directive has a **`require`** option with value `^^myTabs`. When a directive uses this option, `$compile` will throw an error unless the specified controller is found. The `^^` prefix means that this directive searches for the controller on its parents. (A `^` prefix would make the directive look for the controller on its own element or its parents; without any prefix, the directive would look on its own element only.)

So where does this `myTabs` controller come from? Directives can specify controllers using the unsurprisingly named `controller` option. As you can see, the `myTabs` directive uses this option. Just like `ngController`, this option attaches a controller to the template of the directive.

If it is necessary to reference the controller or any functions bound to the controller from the template, you can use the option `controllerAs` to specify the name of the controller as an alias. The directive needs to define a scope for this configuration to be used. This is particularly useful in the case when the directive is used as a component.

Looking back at `myPane`'s definition, notice the last argument in its `link` function: `tabsCtrl`. When a directive requires a controller, it receives that controller as the fourth argument of its `link` function. Taking advantage of this, `myPane` can call the `addPane` function of `myTabs`.

If multiple controllers are required, the **`require`** option of the directive can take an array argument. The corresponding parameter being sent to the `link` function will also be an array.

```
angular.module('docsTabsExample', [])
.directive('myPane', function() {
  return {
    require: ['^^myTabs', 'ngModel'],
    restrict: 'E',
    transclude: true,
    scope: {
      title: '@'
    },
    link: function(scope, element, attrs, controllers) {
      var tabsCtrl = controllers[0],
          modelCtrl = controllers[1];

      tabsCtrl.addPane(scope);
    },
    templateUrl: 'my-pane.html'
  };
});
```

Savvy readers may be wondering what the difference is between `link` and `controller`. The basic difference is that `controller` can expose an API, and `link` functions can interact with controllers using **`require`**.

Best Practice: use `controller` when you want to expose an API to other directives. Otherwise use `link`.

Summary

Here we've seen the main use cases for directives. Each of these samples acts as a good starting point for creating your own directives.

You might also be interested in an in-depth explanation of the compilation process that's available in the compiler guide (guide/compiler).

The $compile API (api/ng/service/$compile) page has a comprehensive list of directive options for reference.