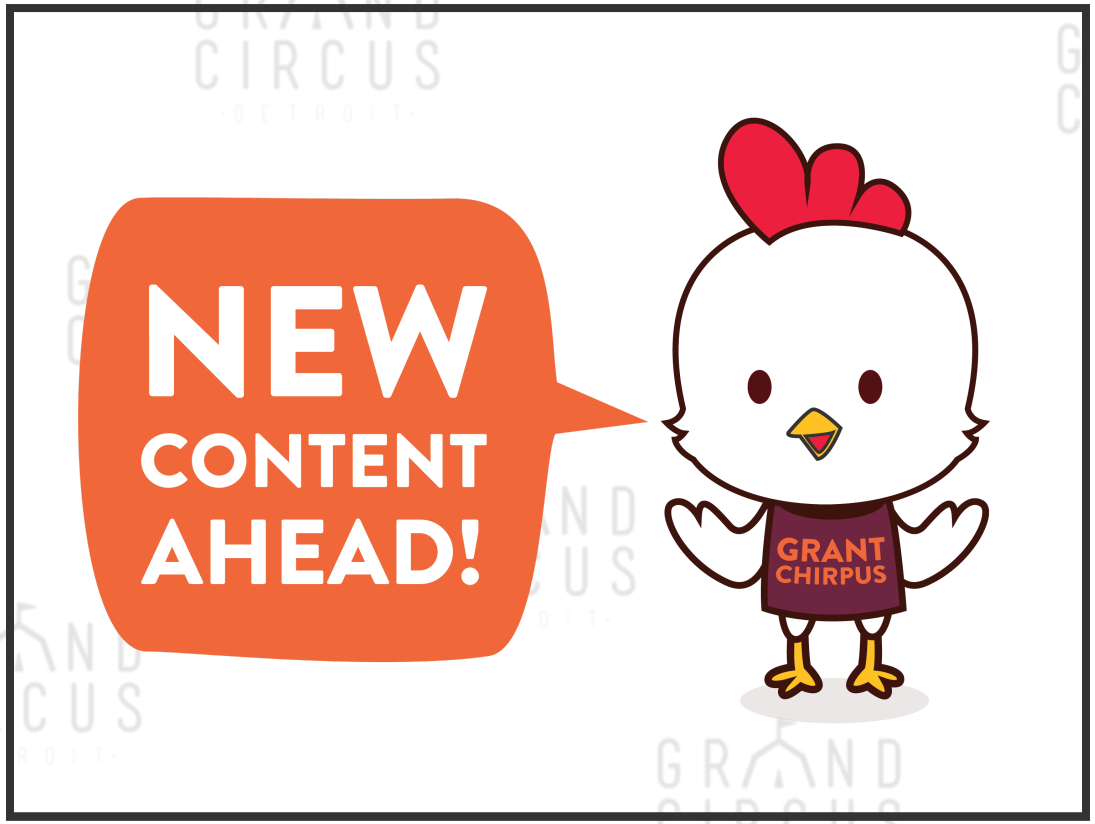

NEW
CONTENT
AHEAD!
GRANT
CHIRPUS

FORM VALIDATION

FORM VALIDATION

We've talked about forms before, but let's talk about how to validate them the Angular way.

Angular provides us with a handful of properties for our form. These properties give us control and provide information regarding the form itself and the inputs associated with it.

VALIDATION DEMO

Demo

Property	Class	Description
$valid	ng-valid	Boolean - Is the item valid based on our rules
$invalid	ng-invalid	Boolean - Is the item invalid based on our rules
$pristine	ng-pristine	Boolean - Has the form or input NOT been used yet
$dirty	ng-dirty	Boolean - Has the form or input been used yet
$touched	ng-touch	Boolean - Has the input been blurred yet

FORM VALIDATION

Let's take a look at our form. Notice any properties?

`novalidate` - prevents HTML5 validations since AngularJS will handle this

`min-length`/`max-length` - min/max length of characters for inputs

`required` - required to be submit form

FORM VALIDATION

`ng-disabled` - button is disabled when form is invalid

`ng-show` - shows or hides an element based on an expression

`ng-class` - adds class based on an expression

DIRECTIVES

DIRECTIVES

Directives are the most important and most powerful part of Angular. Directives are the part of angular that allows us to extend native HTML with custom elements and attributes. The result is that this makes our markup a much more expressive and easier to follow.

COMPARE

```
<div class="container">
  <div id="header">
    <div class="nav">
      <div class="nav-item active"><a href="/home">Home</a></div>
      <div class="nav-item"><a href="/about">About</a></div>
      <div class="nav-item"><a href="/register">Register</a></div>
      <div class="nav-item"><a href="/settings">Settings</a></div>
    </div>
  </div>
</div>
```

VS.

```
<app-container>
  <header>
    <nav>
      <nav-item href="/home">Home</nav-item>
      <nav-item href="/about">About</nav-item>
      <nav-item href="/register">Register</nav-item>
      <nav-item href="/settings">Settings</nav-item>
    </nav>
  </header>
</app-container>
```

DIRECTIVES

We've already made extensive use of directives in Angular. But we've only been using the ones that come pre-packaged with the framework. It is also possible to create our own directives to achieve more specific and custom functionality. At their heart, directives are the ability in angular to teach HTML new tricks.

HOW TO DIRECTIVE

To create a custom directive, we start by declaring it in much the same way we do a service or controller and providing a callback

```
var app = angular.module('myModule', []);
app.directive('helloWorld', function(){
});
```

HOW TO DIRECTIVE

Inside the callback function, return an object literal (directive definition) that has a set of properties that will configure the directive.

```
app.directive('helloWorld', function(){
  return {
    restrict: "E",
    template: "<h1>Hello World</h1>",
    replace: false
  };
});
```

DIRECTIVE PROPERTIES

Property	Description
`restrict`	Defines how the directive can be used (A for Attribute, E for element, etc.)
`template`	Defines the HTML that will be used when this directive is compiled and inserted into the DOM
`templateUrl`	Provides a path to an html file instead of writing the template in place
`replace`	Determines whether the directive will be replaced with the template.

TEMPLATE URL

`templateUrl` allows us to specify a path to a file for our template instead of hard coding a template right in the directive itself. This is typically a better practice, especially as the complexity of templates increases.

```
app.directive('awesomeDir', function() {
  return {
    restrict: 'E',
    templateUrl: 'partials/awesome-dir.html',
    replace: false
  };
});
```

HOW TO DIRECTIVE

Once you have defined your directive. You can use it in your HTML just like a native element or attribute. So our previous directive example is used thusly.

```
<hello-world></hello-world>
<!-- or -->
<hello:world></hello:world>
```

DEMO

CODE ALONG

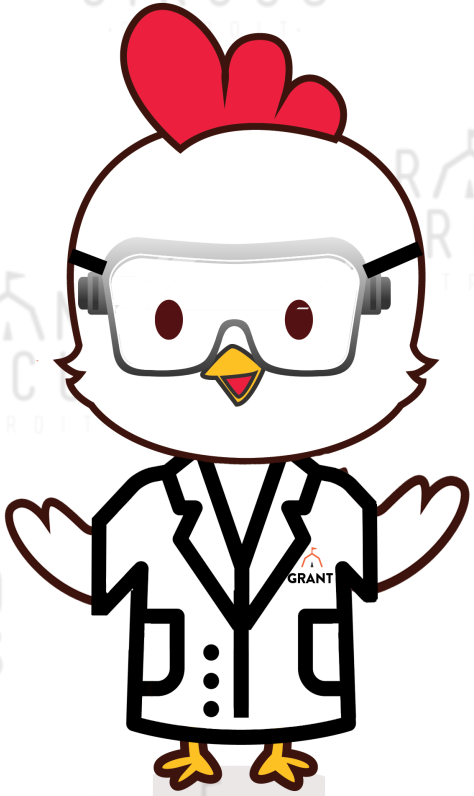
1. Create new folder with basic index.html and script.js.
2. Include Angular in index.html via download or CDN.
3. Create a module in script.js and use it in index.html.
4. Create a `helloWorld` directive in script.js that creates an `h2` tag with the text "Hello World".
5. Use your new custom directive in index.html.

CODE ALONG

1. Create a `loremIpsum` directive in script.js that uses a template URL for its content.
2. Create a partial HTML file for the content of the loremIpsum directive. It should be two `p` tags with some Lorem Ipsum text.
3. Use your new custom directive in index.html.

LAB 16

CUSTOM DIRECTIVES



CUSTOM DIRECTIVES

Refactor an HTML site using custom directives.

- Download a basic website template from OSWD.
- Replace the sections of the site (main content, header, sidebar, etc.) with custom directives.
- You must use at least three custom directives.

HINTS

Make sure the name for your directive matches the name you're using in your markup.

Double check the path to your templateUrl.

BONUS

Make a second page of the site that also uses some or all of the same custom directives but differs in some way.

Use Angular expressions `{{ }}` within you custom directives, for example to specify the page title or part of the content.

PRE-GAME

MORE ON DIRECTIVES

DIRECTIVES, PART 2

If this were all they did, Directives would still be pretty rad. But they actually are capable of a lot more. One of the most powerful aspects of directives is their access to angular's scope object.

LINK FUNCTION

Another of the properties that can be defined for a directive is the `link` property. The `link` property is mainly used for attaching event listeners and doing DOM manipulation. To achieve this, the value of the link property is an anonymous function that runs when the directive is compiled.

SIMPLE LINK FUNCTION

By default, directives **share** the scope object with their parent controller.

```
app.directive('linkEx', function(){
  return {
    restrict: "E",
    template: "<h1>Hello, {{name}}</h1>",
    link: function(scope, elem, attrs){
      scope.name = 'James!'
    }
  };
});
```

DEMO

MORE COMPLEX LINK EXAMPLE

The `link` function's main job is to create event handlers and DOM Manipulation (like in jQuery)

In the next example, we bind a `color` property to the scope and use it to change the color of the heading.

In addition, we bind a callback function to the `mouseover` event which changes the mouse cursor.

```
app.directive('colorText', function() {
  return {
    restrict: 'E',
    replace: true,
    template: '<h1 style="color:{{color}}">Hello World</h1>',
    link: function(scope, elem, attrs) {
      elem.on('click', function() {
        elem.css('color', 'black');
        scope.$apply(function() {
          scope.color = "black";
        });
      });
      elem.on('mouseover', function() {
        elem.css('cursor', 'crosshair');
      });
    }
  };
});
```

DEMO

ANATOMY OF THE LINK FUNCTION

The link function accepts three arguments.

Argument	Description
`scope`	The scope of the directive as defined by the directive definition object
`elem`	The jQLite (a subset of jQuery) wrapped element on which the directive is applied
`attrs`	An object of normalized attributes on which the directive is applied

DEEPER INTO DIRECTIVES

OTHER PROPERTIES

This is not an exhaustive list, but here are some more properties that are useful in the creation of custom directives.

Prop	Description
`transclude`	Allows a directive to include content from another template
`scope`	Gives the ability to modify the scope of the directive

TRANSCLUDE

Think *trans*fer and in*clude*. Whatever is inside the original directive gets transferred and included inside the resulting HTML as well.

The best metaphor I've ever heard for transclusion directives is to think of them as a picture frame with the directive making up the frame of the picture. The 'foreign' content is what shows up in the center of the picture and it can be included from an entirely different scope.

TRANSCLUDE EXAMPLE

```
app.directive('yesTransclude', function() {
  return{
    transclude: true,
    template: '<div>An example of more things <ng-transclude> </ng-transclude></div
    replace: true
  };
});
```

DEMO

DIRECTIVE SCOPE

By default, directives share their parent's scope. We don't always want that. If our directive needs to add properties or functions for its own use, we don't want those properties and functions polluting the parent scope. We have a couple of options to mitigate this:

- A child scope - A scope that inherits the parent scope.
- An isolated scope - A new scope that does not inherit from the parent and exists on its own.

DIRECTIVE SCOPE

In order to create these separate scopes, we use the `scope` property in our directive definition object.

```
app.directive('childScope', function() {
  return{
    scope: true,
    template: '<p>This directive will have an inherited scope.</p>'
  };
});
```

DIRECTIVE SCOPE

To create a scope that is completely isolated from its parent...

```
app.directive('childScope', function() {
  return{
    scope: {},
    template: '<p>This directive will have an isolated scope.</p>'
  };
});
```

ISOLATE SCOPE WHY?!

Isolating the directive's scope makes it easier to plug in multiple locations throughout your app without having to worry about what scope it will inherit and what will be accessible to it. This does not mean there's no way for an isolated scope to communicate with other components and scopes. But making that work is a pretty advanced subject. For now we'll stop here and cover the more advanced stuff on an as-needed basis.

CUSTOM ATTRIBUTES

```
<say-hello name="Dr. Jones"></say-hello>
```

```
app.directive("sayHello", function() {
  return {
    scope: {
      "name": "@"
    },
    template: "<h3>Hello {{name}}!</h3>"
  };
});
```

DEMO

JQUERY $.GET() VS. ANGULAR $HTTP

JQUERY

```
$.get('http://api.example.com/things', function(data) {
    console.log(data);
});
```

JQUERY

```
$.get('http://api.example.com/things', function(data) {
    console.log(data);
});
```

What is the same? What is different?

ANGULAR

```
$http.get('http://api.example.com/things').then(function(response) {
    console.log(response.data);
});
```

ANGULAR - POST

```
$http({
    method: 'POST',
    url: 'http://api.example.com/things',
    data: { name: 'Chioke', course: 'JS' }
}).then(function(response) {
    console.log(response.data);
});
```

`data` is the body of the HTTP request.

ANGULAR - GET WITH PARAMS

```
$http({
    method: 'GET',
    url: 'http://api.example.com/things',
    params: { name: 'Chioke', course: 'JS' }
}).then(function(response) {
    console.log(response.data);
});
```

`params` adds the parameters to the URL query:

http://api.example.com/things?name=Chioke&course=JS

PROMISES

PROMISE

Analogy: Restaurant Pager

...THEN

PROMISE

```
promise.then(function(data) {
   // do something with data here.
});
```

The `then` method of a promise takes a callback function that will be called when the promise is ready and actually has the data to give me. The function takes the data as a parameter.

PROMISE EXAMPLE

```
$http.get('http://api.example.com/things').then(function(response) {
    console.log(response.data);
});
```

Or we can store the promise in a variable. Both of these do the same thing.

```
var promise = $http.get('http://api.example.com/things');

promise.then(function(response) {
    console.log(response.data);
});
```

PROMISE QUIZ

What's wrong with this?

```
$scope.results = $http.get('http://api.example.com/things');
```

Should be...

```
$http.get('http://api.example.com/things').then(function(response) {
    $scope.results = response.data;
});
```

PROMISE QUIZ

What's wrong with this?

```
$scope.results = $http.get('http://api.example.com/things').then(function(response)
    console.log(response.data);
});
```

Should be...

```
$http.get('http://api.example.com/things').then(function(response) {
    $scope.results = response.data;
});
```

PROMISE QUIZ

What's wrong with this?

```
$http.get('http://api.example.com/things').then(function(response) {
    console.log(response.data);
});
$scope.results = response.data;
```

Should be...

```
$http.get('http://api.example.com/things').then(function(response) {
    console.log(response.data);
    $scope.results = response.data;
});
```

PROMISE

Just right!

```
$http.get('http://api.example.com/things').then(function(response) {
    $scope.results = response.data;
});
```

LET'S PLAY WITH IT

Demo & Exercises

BONUS

- Array.map()
- Chaining Promises
- Passing Promises

INTERLUDE: ARRAY.MAP()

ARRAY.MAP()

Creates a new array by changing each element. The original array is not changed.

```
var array = [ "red", "green", "blue" ];

var capsArray = array.map(function(element) {
    return element.toUpperCase();
});

console.log(array);     // > red, green, blue
console.log(capsArray); // > RED, GREEN, BLUE
```

ARRAY.MAP()

Another example

```
var array = [{ "name": "grass", "color": "green" },
  { "name": "corn", "color": "yellow" },
  { "name": "fire engine", "color": "red" }];

var names = array.map(function(element) {
    return element.name;
});

console.log(names); // > grass, corn, fire engine
```

ARRAY.MAP()

What will `newArray` contain?

```
var cities = [{ "name": "Detroit", "state": "MI" },
  { "name": "Grand Rapids", "state": "MI" },
  { "name": "New York", "state": "NY" }];

var newArray = array.map(function(city) {
    return city.name + ", " + city.state;
});

console.log(newArray);
```

ARRAY.MAP()

```
var cities = [{ "name": "Detroit", "state": "MI" },
  { "name": "Grand Rapids", "state": "MI" },
  { "name": "New York", "state": "NY" }];

var newArray = array.map(function(city) {
    return city.name + ", " + city.state;
});

console.log(newArray);  // > [ "Detroit, MI", "Grand Rapids, MI", "New York, NY" ]
```

CHAINING PROMISES

CHAINING PROMISES

```
var responsePromise = $http.get('http://api.example.com/things');

var dataPromise = responsePromise.then(function(response) {
    console.log(response);
    return response.data;
});

dataPromise.then(function(data) {
    console.log(data);
});
```

EXAMPLE REQUEST RESPONSE

```
[{
    "name": "grass",
    "color": "green"
},{
    "name": "corn",
    "color": "yellow"
},{
    "name": "fire engine",
    "color": "red"
}]
```

CHAINING PROMISES

```
// The result of this promise is the raw request
var responsePromise = $http.get('http://api.example.com/things');

// the result of this promise is the data (body) of the request
var dataPromise = responsePromise.then(function(response) {
    return response.data;
});

// the result of this promise is just an array of the names
var namesPromise = dataPromise.then(function(things) {
    return things.map(function(thing) {
        return thing.name;
    });
});

namesPromise.then(function(names) {
    console.log(names);
});
```

CHAINING PROMISES

```
$http.get('http://api.example.com/things').then(function(response) {
    return response.data;
}).then(function(things) {
    return things.map(function(thing) {
        return thing.name;
    });
}).then(function(names) {
    console.log(names);
});
```

PASSING PROMISES

In your service...

```
function myMethod() {
    var promise = $http.get('http://api.example.com/things').then(function(response)
        // Modify the response here. Pick what data you want and return it.
        return response.data;
    });
    return promise;
}
```

In your controller...

```
var promise = service.myMethod();

promise.then(function(data) {
    // Here data is what was returned from the `then` function in the service.
    $scope.result = data;
});
```

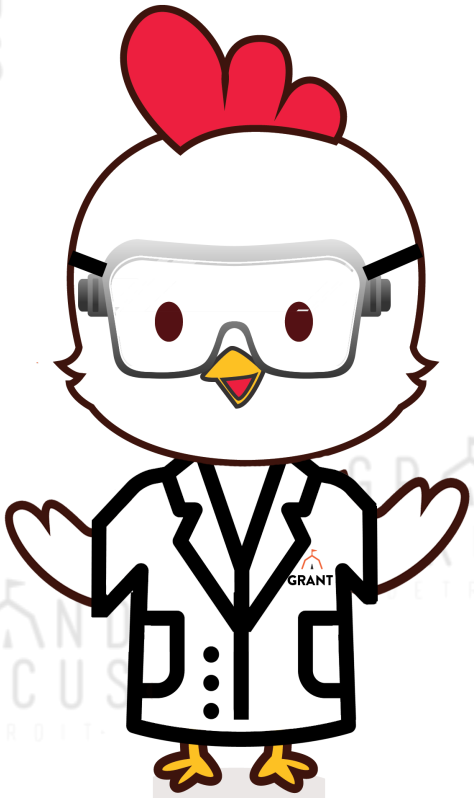
HOMEWORK

From ng-book:

- Promises: 233-241

LAB 17

RICH MAN'S REDDIT



RICH MAN'S REDDIT

Redo Poor Man's Reddit using Angular.

Think about the differences between the how you do this with jQuery versus AngularJS.

- Use the https://www.reddit.com/r/awww/.json API to pull JSON via Angular's $http service.
- Display the posts on a webpage using Angular.
- You may use the exact same CSS as you used before.