year of moo

View Archive View Tags
Search for Programming, AngularJS, Rails, Testing ...

More AngularJS Magic to Supercharge your Webapp

Make way for another amazing article which covers more of AngularJS

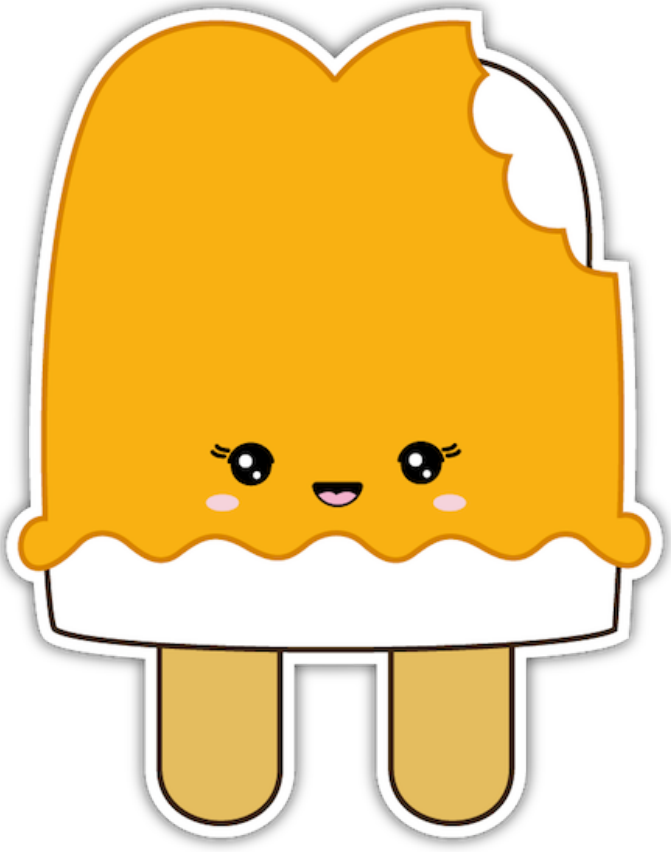
Due to the popularity of the previous article, Use AngularJS to power your web application, I've decided to cover more of AngularJS to make it fun and easy for all developers to play around with it. AngularJS is an incredible tool, but a lot of the more advanced features are hidden in deep within the documentation and others are too tricky to learn direclty. AngularJS is also a fairly new product and as a result there are many features that are yet to be discovered and blogged about.

This article will cover more of the hidden gems of AngularJS and introduce new development tricks and methods to supercharge your AngularJS application. Pleaes read onwards if you wish to become an AngularJS web guru :).

Last Updated

This page was first published on October 2nd 2012 and was last updated on January 30th 2013.

Table of Contents



About this article

This article is a sequel to the previous article titled Use AngularJS to Power Your Web Application. For those of you whom have not already read the article then be sure to fully absorb it prior to diving into this article. All of the topics in this article are based off of knowledge and topics introduced in the previous article so be sure to take a look.

To clarify things a bit, in the previous article the primary module for angularJS was loaded as follows (the App variable is used quite often in this article):

```
//you can leave out the ngResource array item if you want to
var App = angular.module('YOUR_APP_NAME', ['ngResource']);
```

AngularJS is huge and offers many features that have yet to be discovered by it's developer community. Not so much that the features are hidden, but more so the use for many of it's features aren't fully covered anywhere on the internet. This article goes over a lot. It will introduce many advanced tricks and secrets that you can use to make your AngularJS-powered web application even more insane than it already is.

Click here to view the previous article

AngularJS and Internet Explorer

Before we delve into more of the magic about AngularJS, some info about Internet Explorer should be covered.

It's not really recommended that IE6 and IE7 are used on an AngularJS application. The lack of support for custom tags and hashchange events makes the application itself very bloated and slow. Despite there being some support for them, AngularJS is recommeded only to be used with A Grade Browsers such as (Chrome, Firefox, IE8+, Safari and Opera). Try your best to avoid using AngularJS with IE6 and IE7 and you should be OK.

AngularJS metions that the use of custom tags are supported in IE8 and it's noted that AngularJS does work fine with IE8 when using any custom tags. I don't believe entirely that IE8 respects custom tags since a special HTML5 shiv file is required to make HTML5 tags work with IE8. Therefore you will need to define each of the custom tags you use at the top of your file (document.createElement('ng-pluralize'), document.createElement('ng-view'), etc...). Thus if can you work around using custom tags or if you can avoid IE8 all together then just stick to using regular HTML/HTML5 tags and that should work fine.

Click here to read more about IE compatibility with AngularJS

Dataing Binding and $scope changes

Data Binding works by echoing the values contained in the $scope variable to the DOM using various binding approaches. Whether it be directly with the curly braces {{ some_value }}, or directly within an attribute using a binding model, AngularJS is designed to do this well and to do it quickly.

AngularJS handles changes in the $scope ~~with a poll and watch approach but it's not exactly what you expect polling to be~~ whenever a major event occurs within the application, and then, once the event occurs, a digest operation is issued and the scope is updated. The AngularJS $scope variable acts as a big key/value storage hash which is internally looped ~~on a timeout interval~~ and dirty checked against it's former value(s) each time a digestion occurs. If there is a change with the value compared to it's former value, then AngularJS fires off a change event and renders the necessary DOM handles to make that DOM binding render based on the value of that variable. These major events that AngularJS pays attention to are user input events (mouse, keyboard, etc...), any kind of important browser events (timeout, blur, etc...), and when any data returned is returned from the server ($http calls, template downloads, etc...). This is why whenever something happens outside of the AngularJS code (a third party plugin does something, or when the URL is manually changed) a direct call to the $scope.$apply() must be made to inform AngularJS to recognize that change (all of this is explained in more detail later on).

The nice thing about all this is that AngularJS isn't greedy and only runs when it has to, so if your webpage is idling as a background tab then, unless you have implemented something to loop in the background, then AngularJS should not fire off any digest calls.

Click here to find out more about these major events on StackOverflow.com

You would think that any application that is ~~constantly~~ periodically checking the same data over and over again would slow down very quickly, but no. AngularJS performs remarkably well in comparison to other JavaScript MVC frameworks and this same dirty checking approach is pretty common with other types of programs such as video games.

Click here to find out more about this on StackOverflow.com Click here to find out more in the Documentation

Root Scope and Extending Scope Members

The $rootScope acts as the parent scope object of all other $scope objects. This means that when a controller is is executed then the $scope variable that is provided to it will have it's contents linked/cloned from the $rootScope object. It's best just to think the $scope variable as a child class of the $rootScope ($scope extends $rootScope). Undestanding this is useful for when you want to have special methods attached to your scope variable for use across your entire application (such as session information, flags, states, etc...).

The following example below is an example of how you can attach different libraries or code objects to your $scope instance.

```
App.run(['$rootScope',function($rootScope) {

  //this will be available to all scope variables
  $rootScope.includeLibraries = true;

  //this method will be available to all scope variables as well
  $rootScope.include = function(libraries) {
    var scope = this;
    //attach each of the libraries directly to the scope variable
    for(var i=0;i<libraries.length;i++) {
      var key = libraries[i];
      scope[key] = getLibrary(key);
    }
    return scope;
  };

}]);
```

And then inside of your controller or directive you can do the following.

```
var Ctrl = function($scope) {
  if($scope.includeLibraries) { //the flag was set in the $rootScope object
    $scope = $scope.include(['plugin1', 'library1']);
  }
};
Ctrl.$inject = ['$scope'];
```

Try not to set too much data into the $scope and $rootScope variables. Afterall, AngularJS deals with $scope data very very often and you wouldn't want to overwhelm it ;)

$apply, $digest and $$phase

This is the most important thing to know about AngularJS. There will come a time and place when you need to integrate a 3rd party app into your website and you will find that it breaks or doesn't get picked up by Angular. To get this to work you will need to understand how the $digest and $apply methods work.

Each time a major event occurs in a web application that is angular is running (when a webpage is first loaded, when new AJAX request is recieved, URL changes, etc...) Angular picks up the change and then prepares a digestion (which is the internal loop which is run on the $scope memeber). This only takes a few milliseconds, but angular only runs this process once at a time. You can manually kickstart this process by running the $scope.$apply() method (this is useful for triggering updaets when 3rd party applications do something with your webpage that angular needs to know about). Also if you set your own bindings and run the $scope.$apply() method then an exception may be thrown which may cause your code to stop (this happens when an existing digestion is going on in the background). So you will need to aware when a digestion is going on by checking the $$phase variable (this is explained below). The $apply method runs the $diest() method which is an internal method which triggers angular to poll all it's $watch methods.

To get around the $apply exception you will need to pay attention to the $scope.$$phase flag to see if a digestion phase is going on in the background. If a phase is going on then you can just set the $scope values directly and they should get picked up by the current digestion. Here is a combined method which I use to get around this.

```
//when you add it to the $rootScope variable, then it's accessible to all other $scope variables.
$rootScope.$safeApply = function($scope, fn) {
  fn = fn || function() {};
  if($scope.$$phase) {
```

```
    //don't worry, the value gets set and AngularJS picks up on it...
    fn();
  }
  else {
    //this will fire to tell angularjs to notice that a change has happened
    //if it is outside of it's own behaviour...
    $scope.$apply(fn);
  }
};

//and you can run it like so.
$scope.some_value = 'value...';
$scope.$safeApply($scope, function() {
  //this function is run once the apply process is running or has just finished
});
```

You may use the method above in any situation when you manually need to set a binding yourself which is out of the natural order of things within angular services (for example when you run any $http method then angular will pickup the changes automatically).

In the event that you wish to change the URL of the webpage, then you will have to also pay attention to the $$phase variable to see if you're "allowed" to change the URL. If a digestion phase is going on then you can just fallback to changing the URL the old fashion way using window.location.

```
//be sure to inject $scope and $location somewhere before this
var changeLocation = function(url, force) {
  //this will mark the URL change
  $location.path(url); //use $location.path(url).replace() if you want to replace the location instead

  $scope = $scope || angular.element(document).scope();
  if(force || !$scope.$$phase) {
    //this will kickstart angular if to notice the change
    $scope.$apply();
  }
};
```

This should ensure that your webpage URL will get changed no matter what.

Communication Between Services and Controllers

Whenever you have an event take place in your application which affects all controllers and directives it's best to use the $emit, $on and $broadcast methods provided by AngularJS. Examples of this include permissions and session changes (like when a user logs out or when a flag is raised).

When you need to have a parent controller or scope instruct all child controllers about a change then you can use the $broadcast method.

```
//get the topmost scope
var $scope = angular.element(document).scope();

//logout event
var logoutEvent = 'logout';
var logoutArgs = ['arg'];
$scope.$broadcast(logoutEvent, logoutArgs);

//login event
var logoutEvent = 'logout';
var logoutArgs = ['arg'];
$scope.$broadcast(logoutEvent, logoutArgs);
```

Then within your controllers or directives do this:

```
//
//in your controller
//
var Ctrl = function($scope) {
  $scope.$on('logout', function(args) {
    alert('bye bye');
  });
  $scope.$on('login', function(args) {
    alert('hello there');
  });
};
Ctrl.$inject = ['$scope'];

//
//in your directive
//
App.directive('sessionStatus', function() {
  return function($scope, element, attrs) {
```

```
    $scope.$on('login', function() {
      element('html', 'You are logged in!');
    });
    $scope.$on('logout', function() {
      element('html', 'You are logged out!');
    });
  };
});
```

You can also fire events back upwards using $scope.$emit.

```
var Ctrl = function($scope) {
  $scope.onLogoutClick = function() {
    $scope.$emit('logout');
  }
};
Ctrl.$inject = ['$scope'];

//upper $scopes will respond to this with the same $on method.
```

Using these inter-controller communication methods there is no need to create shared code between controllers.

Additional features of Controllers and Routes

AngularJS also comes with other less known methods of handling requests between controllers. But, before I get into that I would just like to point out an alternative way of creating controllers.

```
App.controller('Ctrl', ['$scope', function($scope) {
  //exactly same result as creating a controller with an explicit function
}]);
```

OK so onto business. When a request happens you can create a resolving action as apart of the route which is basically a function that is fired just before the request is shipped off to the controller. Here's an example of it:

```
//checkout the route code below first before reading this
var Ctrl = function($scope, $http, argument1, argument2, argument3) {
  alert(argument1); //someDependeny's value or toString method
  alert(argument2); //"some value"
  alert(argument3); //the response.data value returned from GET /path/to/some/url
};
Ctrl.$inject = ['$scope', '$http', 'argument1', 'argument2', 'argument3'];

//here's where the magic happens
App.config(['$routeProvider',function($routeProvider) {
  $routeProvider.when('/some/page/with/an/:id',{
    templateUrl: '/path/to/some/template.html',
    controller: Ctrl,
    resolve: {
      argument1: 'someDependency', //this is a registered dependency and works the same as a dependeny injection would within a controller
      argument2: function() { return 'some value'; }, //this is resolved instantly since there is nothing going on
      argument3: function() {
        return $http.get('/path/to/some/url',function(response) {
          return response.data; //this is what is returned as the value for argument3
        }
      }
    }
  });
}]);
```

When the application makes it's way to the path /some/page/with/an/:id then angular wil start to resolve the dependencies each outlined in the resolve:{} block. Once ready (when all the promises are met), then it will forward the request to the controller and work as usual. This is nice for when you want to abstract your data requirements outside of your controller. And within the resolve function you can also inject any service as usual (this is useful for when you need to have data be fetched prior to your controller being run). Also keep in mind that if you set the controller directly as a ng-controller attribute then the resolve block itself is skipped (since it depends on the path [route] of the page instead of the controller binding).

Click here to view an existing application using route resolving

The $route variable can also be injected directly into the controller and this can be used to fetch information about the current route:

```
var Ctrl = function($scope, $route) {
  $route.current.templateUrl; //the URL of the template
  $route.current.params; //same as $routeParams
  $route.current.controller; //the name of the controller (Ctrl)
```

```
};
Ctrl.$inject = ['$scope','$route'];
```

You should be using Custom Services

Custom services are what make angular very manageable and easily testable. By using angular's dependency injection feature you can create a custom service anywhere within your application and include it elsewhere very easily. A common example of where a shared service makes sense is to use it as a special $http service which is tailored to fit your application.

```
App.factory('myHttp',['$http',function($http) {
  return function() {
    get : function(url, success, fail) {
      $http.get(url).success(function(response) {
        return response.data;
      }).error(fail);
    }
  };
}]);

//this service can now be called by doing...
$myHttp.get('/path', function(data) {
  alert(data);
});
```

Also, below is a description of how data is shared between services within the same model.

```
App.factory('myFoo',['$http',function($http) {
  //any variables defined in this area will be ACCESSIBLE
  //within the any of the other services which are defined
  //within the same module. So if a variable called foo...
  var foo = 'bar';
  //then foo can be accessed in another service. Be sure to
  //keep this in mind since it's hard to debug
  return foo;
}]);

App.factory('myBar',['$http',function($http) {
  var bar = 'bar2';
  return foo.toString() + bar; //this should return either bar2 or barbar2
}]);
```

You can also inject any of your own services into any other services when created; this is very useful for code reuse and testing.

Show, Hide, Cloak and Init

You will find that showing and hiding values in your angular templates may be difficult since you cannot rely on the server-side programming language to build your template (templates are static). Here's an example of how something would work normally when using a something like PHP:

```
<div class="session">
  <?php if($isAdmin) { ?>
    <span class="admin">Hello Admin</span>
  <?php } else { ?>
    <span class="user">Hello User</span>
  <?php } ?>
</div>
```

The same effect can be created by using angular

```
<div class="session">
  <span class="admin" data-ng-show="isAdmin">Hello Admin</span>
  <span class="admin" data-ng-hide="isAdmin">Hello User</span>
</div>
```

Just be sure to set the binding value

```
$scope.isAdmin = true; //or false or whatever
```

This works, but when the page is still downloading (when first loaded) you may see both values at the same time so to get around this just use cloaking.

```
<div class="session ng-cloak">...</div>
```

And define the CSS for it as well:

```
.ng-cloak {
  /* this will change to block when scope and angular is ready */
  display:none;
}
```

Oh! And one more thing. If you wish to set the isAdmin value directly into your HTML, then do the following using the data-ng-init:

```
<div class="session ng-cloak" data-ng-init="isAdmin=false;">
  <span class="admin" data-ng-show="isAdmin">Hello Admin</span>
  <span class="admin" data-ng-hide="isAdmin">Hello User</span>
</div>
```

The data-ng-init attribute is useful for pre-setting values. The syntax works like you would be setting values in JavaScript directly (since it uses eval to evaluate the text within the attribute value).

Isn't this stuff just amazing!

Catching Errors

Catching errors is something that is important for a production application. Below are various approaches do doing so:

Catchall Route (otherwise)

Despite this being a useful method as a default page for a route, it's best to reserve this route as your 404 page handler in the event that a route is not recognized within your application.

```
$routeProvider.when('/404',{
  controller : ErrorCtrl
});
$routeProvider.otherwise({
  redirectTo : '/404'
});
```

When your routes fail

In the event that a route change fails (due to a missing templateUrl or something) then you can capture the event within your scope by doing the following:

```
App.run(['$rootScope','$location',function($rootScope, $location) {
  $rootScope.$on("$routeChangeError", function (event, current, previous, rejection) {
    //change this code to handle the error somehow
    $location.path('/404').replace();
  });
}]);
```

Wrap Services Around your HTTP Requests

Earlier in the article I explained the importance of custom services for code reuse. When you set a custom service to wrap all your AJAX calls then you can catch errors before they're passed onto other parts of your application.

```
App.factory('myHttp',['$http','$location',function($http, $location) {

  var onEmpty = function() {
    window.location = '/404';
  };

  return function() {
    get : function(url, success, fail) {
      $http.get(url).success(function(response) {
        var data = response.data;
        if(!data) {
          onEmpty();
          return;
        }
```

```
        success();
      }).error(onEmpty);
    }
  };
}]);
```

Click here to read more about this on StackOverflow.com

Be sure to only use this method when you access resources and data that is required within your application (like JSON data for a specific view).

More About Loops

Loops are tricky and funky in anguarJS. Much of their details were covered in the previous article, however some things were not fully covered.

To get access to the index of a loop in angular you can access it from the $index value directly.

```
<ol>
  <li data-ng-repeat="option in options">
    <h2>Option #{{ $index + 1 }}: </h2>
  </li>
</ol>
```

The $index + 1 is used since the index value always starts from zero.

The default syntax relies on an array being defined within your scope. But what happens when you do not have an array set and you simply want to construct a grid with simple max and min values? You will need to setup a filter that prepares the looping array for you. Here's an example of how to do that:

```
App.filter('range', function() {
  return function(input, total) {
    total = parseInt(total);
    for (var i=0; i<total; i++) {
      input.push(i);
    }
    return input;
  };
});
```

And then access it within your loop as a filter (this will create 100 divs from 0 to 99):

```
<div ng-repeat="n in [] | range:100">
  {{ $index }} - do something
</div>
```

Keep in mind that there are also other options available such as $first, $middle, and $last. All of these and more are covered within the angularJS documentation

Click here to read more about AngularJS ngRepeat

Keeping Track of the Path/URL

To grab the current path of the page regardless of it being a hashbang or a direct path (whether by HTML history or not) you can get it by accessing the $location getters.

```
var path  = $location.path();
var url   = $location.absUrl();
var hash  = $location.hash();
```

To keep track of the URL when it changes, you will need to setup a polling event.

```
$scope.$watch('$location.path()', function(path) {
  //new path!
  alert(path);
});
```

Additionally you can set these events explicitly within your scope variable

```
$scope.$on('$locationChangeStart', function(event, newUrl) {
  alert('new location');
});
```

Filters and Custom Filters

There are two ways to define a filter in AngularJS: You can define it as a filter or as a service.

```
App.filter('my', function() {
  return function(data) {
    return data;
  };
});
```

Or you can define it as a service (same thing):

```
App.factory('myFilter', function() {
  return function(data) {
    return data;
  };
});
```

You can use of these filters directly in your HTML:

```
<span class="some-data">{{ value | my }}<span>
```

Or you can also access these filters directly in your services and controllers via dependency injection.

```
App.factory('someService', ['$filter', function($filter) {
  return function(data) {
    return $filter('my')(data);
  };
}]);
```

More about Directives

Directives are usually provided using a link method within the response of a directive block. However there are other options when setting directives:

```
App.directive('myDirective', ['$location', function($location) {

  return {
    restrict : 'ECA', //Element, Comment and Attribute instances are all scanned
    scope : {
      //these values will be apart of the scope variable passed into the link method
      key : 'value',
      key2 : 'value2'
    },
    compile : function() {
      return {
        pre : function() { ... }, //this is called before the directive element is attached to the DOM
        post : function() { ... } //this is called after the directive element is attached to the DOM (same as link)
      };
    },
    link : function(scope, element, attrs, controllerObject) {
      //this is what is normally used and is the same as the compile:post function.
    }
  };

}]);
```

You can also avoid all the clutter by providing a function as the directive. This is the same as passing only a hash that contains the link value.

```
App.directive('myDirective', function() {
  return function($scope, element, attrs, controller) {
    //less code is nice
  };
});
```

There's alot more about directives, but this covers about 90% of the cases for when you would use them.

Click here to read more about AngularJS Directives

Forms and Form Validation

Quick Difference between Form Models and Form Bindings

In the previous article I introduced bindings and form inputs. However, one thing to keep track of is that when you have a form field input (such as a input field or textarea) AngularJS needs to be aware that there is a two way relationship between the field, so if possible (if it works for you) then setup your form field elements as so (instead of using {{ value }}).

```
<input type="text" data-ng-model="name" />
<textarea data-ng-model="name"></textarea>
```

If for some reason it won't work then you can stick to using {{ value }}, but it may not update with it's scope since there isn't a definite two-way binding between the two.

Form Validation

AngularJS doesn't provide extensive form validation like you would expect a fancy UI form valdation plugin to provide, however it does provide enough support to validate your form and it's model data before it gets submitted further. If you are already using another form validation software on top of angular then don't both to use angular's form validation. Here's an example of it's usage.

```
<form novalidate class="simple-form">
  <ol class="fields">
    <li>
      <label for="input-name">Name:</label>
      <input id="input-name" type="text" data-ng-model="name" />
    </li>
    <li>
      <label for="input-email">Email:</label>
      <input id="input-email" type="email" data-ng-model="email" />
    </li>
  </ol>
  <button data-ng-click="submit()">Submit</button>
</form>
```

This basically validates that the value assigned to name is non-blank and the value assigned to email is a valid email. Once valid then the data is passed onto the submit method. I haven't gotten to use much of the functionality yet, but it works as expected. Also, this may not work directly if you use a input type="submit" or input type="button" element (since that submits the form natively. Give it a shot and let me know).

Click here to find out more about AngularJS Forms and Form Validation

Internationalization and Localization

Internationalization (commonly known as I18n) is a feature designed to map isolate translations from your application outside your application logic. Localization (commonly known as I10n) is similar, but it's designed to translate programatic data (such as numbers and dates) into localized formats (good examples of this include currency and date formats).

Angular isn't designed to be a full-scale internationalization and localization tool for the entire client-side area of your web application, but it does provide some useful features.

Pluralization (using ngPluralize)

Lets say you have a page that lists your inbox messages and you wish to display the total amount of messages in a human readable form.

```
<div data-ng-pluralize
  count="messagesCount"
  when="{
    '0' : 'No have no messages in your inbox',
    '1' : 'You have one message in your inbox',
    'other' : 'You have {{ messagesCount }} messages in your inbox'
  }"></div>
```

Ideally you wouldn't stick the english translations directly into your own webapge so you can switch this so that translations are looked up in a global translation lookup table.

```
<div data-ng-pluralize
  count="messagesCount"
  when="{
    '0' : translate('message.zero'),
    '1' : translate('message.one'),
    'other' : translate('message.many', { count : messagesCount })
  }"></div>
```

You would have to define your own lookup since this is something angular doesn't provide by itself.

```
var LOOKUP = {
  'messages.zero' : 'No have no messages in your inbox',
  'messages.one' : 'You have one message in your inbox',
  'messages.many' : 'You have %messagesCount% messages in your inbox'
}
var translate = function(key, args) {
  var data = LOOKUP[key];
  if(data) {
    for(var key in args) {
      var value = args[key];
      key = '%' + key + '%';
      data = data.replace(key,value);
    }
    return data;
  }
  return '';
};
```

Formatting Dates and Times

To format dates and times with angular, any Date and/or date string can be passed into a filter:

```
<!-- this shows up as Sep 9, 2012 -->
<span class="string-date">{{ '2012-09-01' | date:'medium' }}</span>

<!-- same thing but longer and using a variable  -->
<span class="string-date">{{ someDateVariable | date:'fullDate' }}</span>

<!-- and you can set your own -->
<span class="string-date">{{ anotherDateVariable | date:'EEEE, MMMM d,y' }}</span>
```

You can also access the date filter directly within your controllers and services with use of the $filter service.

```
var dateFilter = $filter('date');
var mediumDateString  = dateFilter('2012-09-01', 'medium');
var fullDateString    = dateFilter(someDateVariable, 'fullDate');
var anotherDateString = dateFilter(anotherDateVariable, 'EEEE, MMMM d,y');
```

Click here to find out more about AngularJS Date and Time Filters

How Promises Work

Promises are basically vertical callbacks without the need for nested functions. This is useful for when one part of the program needs to run after something else but is situated in a different area of your application. Examples would include running a permissions HTTP GET operation after checking to see that the user's session is still active (this could be another HTTP call). To get this to work without the use of promises would be a coding nightmare (lots of callbacks and useless coding fluff).

AngularJS promises work effectively to get around this by use of the $q service. They're inspired by Kris Kowall's Q and work similarly. Basically, anything that does something that requires a callback is wrapped into a defer and promise loop and then once that particular promise is resolved (whether by reject() or resolve()) then the next defered promise in the queue is dealt with. So long as the promises are linked with each other (much like the nodes in a queue are linked to each other) then everything will be loaded one after the other.

Click here to find out more the Q plugin on Github

```
//be sure to inject the $q member
var lastPromise, defer = $q.defer();
function runCommand(success, fail) {
  if(!lastPromise) {
    lastPromise = defer.promise;
  }
  lastPromise =  lastPromise.then(success, fail);
}
```

This will create a chain of promises linked together sequentially.

```
runCommand(
  function(anyArguments) {
    //Success! Resolved!
```

```
  },
  function(anyArguments) {
    //Failure! Rejected!
  }
);
```

Now when you run defer.reject() or defer.resolve() then this will kick things into motion. If you run the resolve command then everything within the queue will run under the first method (the success method) and if you run the reject command then everything within the queue will run under the second method (the fail method).

```
if(allIsGood) {
  defer.resolve(anyArguments); //runs success()
}
else {
  defer.reject(anyArguments); //runs fail()
}
```

The functionality itself is the same as running a series of nested functions together linked with callbacks. If anything above it fails then any of it's nested child methods will also fail. The only difference is that you can organize the code much better. Angular also uses this internally for feautres such as interceptors.

Click here to find out more about $q on AngularJS

Includes and Custom HTML Views

Includes (using ngInclude)

If you have an area of your HTML page where you would like to include another snippet of text then you can do this by using the data-ng-include attribute.

```
<div data-ng-include src="/some/path/to/a/template.html">...</div>
```

You can also define it without the custom src attribute and use the include attribute directly.

```
<!-- this is same as above -->
<div data-ng-include="/some/path/to/a/template.html">...</div>
```

The above line will get loaded right away when angular parses the HTML. So to get this to load when you have data available then do this:

```
<div data-ng-show="someVariable" data-ng-include="someVariable">...</div>
```

And setup the $scope.someVariable variable to a value when you're ready to display the include.

Once the include has been downloaded and is ready then the $scope.$on('$includeContentLoaded'); will be emitted. Also if you include an onload attribute within the same element then that attribute will also be evaluated once the include is ready.

Click here to view more about AngularJS ngInclude

Custom HTML and $compile

There will be times when you may want to compile your own HTML as a view (modal windows, popups, etc...) so to get this to work you will have to hack the program around a bit using the $compile service.

Lets assume that you want to load a modal window and you want the contents of the modal window to be run through a controller called ModalCtrl. Here's what your HTML will look like.

```
<div data-ng-controller="ModalCtrl" id="modal-container">
  <header>
    {{ header_title }}
  </header>
  <div class="content">
    {{ content }}
  </div>
</div>
```

And your controller looks like so:

```
var ModalCtrl = function($scope) {
  $scope.header_title = 'My Modal';
```

```
  $scope.content = '...';
};

ModalCtrl.$inject = ['$scope'];
```

Now in your application somewhere you have a directive to load that modal. First it downloads it's content and then compiles it. Once it's compliled then the controller will be run automatically.

```
App.directive('modalLink', ['$http','$compile',function($http, $compile) {

  return function($scope, element, attrs) {
    element.click(function() {
      $http.get('/some-modal.html').success(function(html) {

        //you may need to trim the whitespace around this if you're using
        //jquery since sizzle breaks if you dont...
        if($ && $.trim) html = $.trim(html);
        if(html.trim) html = html.trim(); //or MooTools

        //this is custom HTML you have fetched
        var topScope = angular.element(document).scope();
        var elm = $compile(html)(topScope); //you should provide the top level scope so that it can find the modal in between

        //now just stick this somewhere in your body as an element
        document.body.appendChild(elm);
      });
    });
  };

});
```

Now angular will recognize the modal HTML and the controller set within it's data-ng-controller attribute. Then it will run through the code and execute the ModalCtrl controller. Now all of the logic contained within the ModalCtrl controller will be applied directly into the container of the modal HTML (it's the HTML where it has id="modal-container".

There's much more to this than just compiling HTML. The angular template and templateUrl attributes for directives use it in the background to prepare templates for controllers and so on.

Click here to read more about AngularJS $compile

Inline Templates

If you plan on reusing templates a lot within your application, then you can do so by using inline templates (script tag templates). By setting up a script element with type="text/ng-template" then you can setup your templating data and then use that directly within your HTML elsewhere. Here's an example of how to do that:

```
<script type="text/ng-template" id="heading-template">
  <h5>Hello There </h5>
</script>
```

Now you can summon this template within ng-view or ng-include. The data inside the script tags will be applied as the template html.

```
<!--
  keep in mind that there are quotes around the 'heading-template'
  so that angular won't confuse it with a variable
-->
<div class="something" data-ng-include="'heading-template'"></div>
```

If you set the template inner HTML as empty and then use an include or template with the path pointing to the ID of that template then AngularJS will download the file from your server just like any other template would be downloaded. There really is no point in making an empty ng-template script tag since there is no differnce between having it there or not having it there. AngularJS will still download the template from the server (whatever is set to templateUrl) if it's not found in its template cache.

Using inline templates is best used for pages that contain a lot of variance between items. Examples would include having a list of items that contain various partials that can be assigned to each item. Instead of having to load each partial within your response template you can just prefetch each template and keep them in the HTML. And you can also define inline templates within other templates (so that if they're downloaded from the server they stick around in the template cache.

How to Make Sure Your Directives are run After Your Scope is Ready

Your controller and any directives within the template loaded prior to the controller will all execute once the

template is loaded. This means that if you have data that needs to be fetched prior to your directive rendering then you will have to make sure you directive will only execute once it's $scope contains the dynamic data that is generated/fetched within the controller. Other issues such as early plugin loading can be a problem since you have your data being fetched after your plugins are instantiated. This could mean that the your plugins will display old data from the dom (maybe even raw binding data) because the new data didn't make it in time to the template. To get around this you will have to instruct your plugins to update themselves to load the new DOM data. This is something that you shouldn't have to deal with and this should be automatic.

To get around this issue you will have to either poll some special variable such as $scope.isReady using the $scope.$watch() method, but this tends to break sometimes. A much better solution is to use a promise using the $q variable and set this up within the $rootScope.

The following was originally apart of this article, however, due to it's complexity it has been ported to a plugin and hosted on Github. Here's an example of how to use the plugin to ensure that directives are run after your controller is ready.

```
//your controller
var Ctrl = function($scope, $http) {
  $scope.$prepareForReady();
  $http.get('/some.json')
    .success(function(json) {
      var someArgs = [json.data];
      $scope.$onReady(someArgs);
    })
    .error(function() {
      var someArgs = [null]; //just some argument
      $scope.$onFailure(someArgs);
    });
};

//your directive
module.directive('myDirective', function() {

  return function($scope, element, attrs, controller) {
    $scope.$whenReady(
      function(someArgs) { //called when $scope.$onReady() is run
        element.html('your data was loaded fine');
        element.show();
      },
      function(someArgs) { //called when $scope.$onFailure() is run
        element.html('something went wrong when fetching the data');
        element.show();
      }
    );
  };

});
```

The directive HTML is just like any other directive (just including it here so that everything is covered).

```
<div class="something" style="display:none" data-my-directive>
  ...this data will get replaced once it's ready...
</div>
```

Click here to download the plugin

More Angular HTML Tricks

There are a few other tricks that are useful with AngularJS that I didn't cover in the article fully. Take a look below:

```
<!--- when null then it won't be checked --->
<input type="checkbox" data-ng-checked="value" />


<!--- when null then it won't be selected --->
<input type="radio" data-ng-selected="value" />


<!--- when null then it won't be disabled --->
<input type="radio" data-ng-disabled="value" />


<!--- this will fire the $scope.onChange() method when changed (make sure you set the model) --->
<input type="input" data-ng-change="onChange()" data-ng-model="name" />


<!--- use this if your images are casuing errors --->
<img data-ng-src="{{ imagePath }}" />
```

```
<!--- use this if your links are casuing errors --->
<a data-ng-href="{{ linkPath }}">Link</a>


<!--
  This should select option elements properly,
  just make sure that the is_selected()
  method is defined as apart of your scope ($scope.is_selected()).
-->
<select>
  <option data-ng-selected="is_selected(option.value)" data-ng-repeat="option in options"></option>
</select>


<!--
  You can set alternating colors, but remember that the 'odd'
  and 'even' values include quotes since if they didn't then angular would
  think that they are variables and would evaulate their values and then apply
  them as class values.
-->
<div class="list">
  <div class="list-item" data-ng-class-odd="'odd'" data-ng-class="'even'" data-ng-repeat="item in items">...</div>
</div>


<!--- $scope.someClassName = 'something' --->
<div class="something" data-ng-class="someClassName"></div>
```

Conclusion

That's it for now. Thank you everyone for your interest in the previous article. Hopefully this fills more of the gaps of knowledge behind AngularJS. Please share and link this article and push it forward across the internet. Who knows, maybe another AngularJS article may be in works ;)

What an amazing tool! Thank you :)