THINKFUL PRESENTS

AngularJS Tutorial

Build a Gmail Clone

Introduction

Build a simple email application and learn core AngularJS concepts. By the end of this tutorial you'll be able to see (fake) emails, search by subject line, and read / delete emails.

Prerequisites:

- Understand how to build a basic Javascript application with jQuery
- Know how to launch a basic HTTP server (e.g. `python -m SimpleHTTPServer`
- Be able to clone a GitHub repo

Topics covered:

- Single page applications (SPA)
- Client-side MVC patterns
- Two way data-binding
- Routing with templates
- AngularJS building blocks: Directives, Factories, & Controllers

✓ You'll notice that there are code checks included throughout this guide. We recommend looking at them and making sure you understand the core concepts, but don't worry about replicating them just yet. Save that for the last section, which has a code-along video included. Use the video as a guide when coding up your own Gmail clone!

Note: This guide is open to the public. You can view the source code on GitHub where you'll find a sample app to folow along.

Ready? Let's dive in!

Get notified when new guides are released

Enter your email address here

Send Me Your Next Guide »

Client side MVC

Let's start with a concept that' s core to Angular: client-side MVC

MVC stands for Model, View, Controller. Let's explore each concept individually:

- **Model:** That's the data; the business information of the application.
- **View:** The HTML and presentation of the data. That's what the user sees and interacts with.
- **Controller:** The connector that makes all the different pieces of our application work together.

The MVC pattern is a proven approach to organizing application code that's been refined over many years.

Fun fact: Ruby on Rails is an MVC framework for the Ruby programming language.

Angular likes to use MV* techniques, where the * stands for 'whatever' (often refered to as MVW). In other words, the Controller part is different to usual, but this is beside the point, lets get something working.

Getting Started with AngularJS

Adding Angular to your page

Angular comes as a single `.js` file that needs to be included at the bottom of your HTML page. I'd advise putting it there rather than in the `<head>` for better document rendering.

Note: You need to add the AngularJS library before we can get any of the Angular goodies working and see what they do. Let's do just that!

For this app we're going to use `version 1.2.22`, which at the moment of writing this is the most recent. This is recommended for now as other third party plugins will have not been tested on the latest beta versions.

To download AngularJS go to AngularJS.org and click on the blue "Download" button. Select the 'legacy' branch, which is the latest stable build, and click "Download".

Your HTML should look like this:

```
<html>
    <head>
    </head>
    <body>
        <div></div>
        <script src="lib/jquery-v1.11.1.js"></script>
        <script src="lib/angular-v1.2.22.js"></script>
    </body>
</html>
```

Note: We're also including jQuery.

Angular comes with something called 'jQLite' built in. It's a jQuery-like micro-library that Angular packages internally. jQLite is very light (as the name implies) and doesn't have many of the great jQuery methods that you might need. It's also good to consider that many plugins you might need will likely depend on the full jQuery library.

Good to know: Angular uses the full jQuery library if it's loaded - so I've loaded it before Angular so it can detect it and use it instead of jQLite.

✓ **Code check:** 01-include-angular

Setup Scopes and Directives

Setting up our app

When learning a new framework it helps to have something working as soon as possible. This enables you to play around and dig into what does what. But, before we can get there, we need to talk about Angular scopes and our first two directives: `ng-app` and `ng-controller`.

One of the most fundamental parts of Angular is scopes. Scopes hold your Models (that's your data), they cooperate with your Controllers, and they give the Views everything they need (that's what the user sees and interacts with). Angular scopes operate in a very similar way to the common JavaScript concept of scope.

The first scope we'll need is the application scope, this is exactly what you'd expect it to be: that's the scope your Angular application can operate in. We set this up in our HTML using the `ng-app` attribute.

```
<html ng-app="myApp">
    <head></head>
    <body></body>
</html>
```

Notice how we gave our app a name of `'myApp'`. This will be usable in our whole html file.

Note: We can use ng-app without the "myApp" to declare a default app. In other words, just `<html ng-app>`.

The second scope is `ng-controller`; this will determine where our controller can operate. We can have multiple controllers within our application. Each controller will have its own scope. For example, we may have an `inbox.html` file, containing the below code. It will give responsibility to a controller named `'InboxCtrl'` (in our JavaScript) for this scope.

```
<div ng-controller="InboxCtrl">
    <!-- inside InboxCtrl scope -->
</div>
```

Both `ng-app` and `ng-controller`, are Angular directives. Think of an Angular directive as something that allows you to extend your HTML. Here's a more nerdy explanation in case you're interested.

YOUR FIRST CONTROLLER: A QUICK EXAMPLE

Let's put everything we just learned into practice.

1. Start with a blank HTML document
2. Link your Angular and jQuery files

```
<script src="lib/jquery-v1.11.1.js"></script>
<script src="lib/angular-v1.2.22.js"><script>
```

3. Add `ng-app="myApp"` to the HTML tag
4. Inside the body, create a sample controller

```
<div ng-controller="TestCtrl"><h1>{{title}}</h1>
   <input type="text" ng-model="title">
</div>
```

We'll explain what and `ng-model="title"` are in a minute.

5. Create an inline JavaScript script -- make sure this is below the script tag where you load the libraries

```
<script>
   function TestCtrl($scope) {
      $scope.title = 'Write a title here...';
   };
</script>
```

See how the function name is the same as the `ng-controller`'s value? Angular will be looking for a function with this name in our JavaScript so that it can act as a Controller. Nifty stuff!

Below is the final verion:

```
<!doctype html>
<html ng-app>
  <head>
    <title>Sample AngularJS Controller</title>
  </head>
  <body>
    <div ng-controller="TestCtrl">
        <h1>{{title}}</h1>
        <input type="text" ng-model="title">
    </div>

    <script src="lib/jquery-v1.11.1.js"></script>
    <script src="lib/angular-v1.2.22.js"></script>

    <script>
      function TestCtrl($scope) {
        $scope.title = 'Write a title here...';
      };
    </script>
  </body>
</html>
```

Code check: 02-sample-controller

This can be confusing! It often helps to see the code in action. View the result by downloading the Code check and opening the `index.html` file in your browser. Be sure to change the `$scope.title`'s value as well as play around with the input.

Interested in learning more about controllers? Here's a good guide from AngularJS.org.

ngView and Routes

Connect a URL to a scope

Another important building block that can connect certain URLs of our application to scopes is the `ng-view` directive.

```
<html ng-app="myApp">
    <head>
    </head>
    <body>
        <div ng-view></div>
    </body>
</html>
```

The `ng-view` attribute here will tell Angular where we wish to inject HTML based on the URL a user visits.

In this sample project, we'll want to have an inbox view. When a user visits the `/inbox` URL our (yet to be created) inbox.html file would be injected inside the `ng-view`. The inbox.html file would also have it's corresponding controller (`InboxCtrl`).

Angular encourages the "single page application" methodology, which means we never get a full page refresh, all "pages" are injected content via XHR (Ajax). Of course these aren't "pages" as such, they're "views" - which are loaded depending on the URLs a user visits or the application state.

Modules

Looking at angular.module()

Every app needs a module, and Angular provides us with a namespacing capability for this using `angular.module()`. The method both sets and gets depending how you use it. To create (set) a new module we do this:

```
angular.module('myApp', []);
```

Notice the empty array, this is where we could put any other named modules as dependencies. Here we tell angular we're creating the module named 'myApp' with no dependencies.

To get a reference to a module, for registering controllers, factories, filters, etc… we drop the array and just call the module by name:

```
angular.module('myApp');
```

Note: When we specified the `ng-app="myApp"`, the `"myApp"` part in the module should be the same. Another way to put it, the name of the app has to carry over.

The `angular.module()` method also returns a reference to the module which can make accessing it a little less verbose, however this pattern is discouraged due to potential problems with global variables:

```
var app = angular.module('app', []);
```

Routing and Dependency Injection

Routing

The next logical step is to configure our routing, which controls which views are injected into the app based on which URL we're hitting. For example when we're at `/inbox`, we'll want to inject the `inbox.html` view and assign a Controller. We can use Angular's `$routeProvider` for this.

This is where the real fun begins!

RouteProvider

Since Angular 1.2.0, the `$routeProvider` hasn't been included in the main Angular build, so we'll need to include it as a separate module.

To download the `$routeProvider` (which lives inside `angular-route`) go to [AngularJS.org]() and click on the blue "Download" button. Click on "Browse additional modules", select `angular-route.js` and save that file.

```
<body>
    <!-- ... -->
    <!-- Extra routing library -->
    <script src="lib/angular-route-v1.2.22.js"></script>
</body>
```

Note: Your angular-route version should match your angular.js version.

You can find the version at the top of any file downloaded from AngularJS.org. It looks like this:

```
/**
 * @license AngularJS __v1.2.22__
 * (c) 2010-2014 Google, Inc. http://angularjs.org
 * License: MIT
 */
```

This file will give us access to an additional module named `ngRoute`, which we need to include (set as a dependency) in our own module:

```
var app = angular.module('app', [
    'ngRoute'
]);
```

Config Stage

We saw before how to bind a Controller to the DOM (or to a specific view, to put it another way), which is just one way of doing it. Angular allows you to dynamically assign a Controller through the `$routeProvider` service, which is much more flexible. From now on let's use this instead of using `ng-controller=""` to declare Controllers.

Angular modules each have a `.config()` function, we give this a callback that is run before most other function callbacks in Angular. This is where we must configure our routes (the different URLs visitors will be able to access).

```
app.config(function () {/*...*/});
```

Dependency Injection

We'll need to use the `$routeProvider` to setup our routes inside the config callback; this is made possible via some magic inside the Angular framework (using function definitions and regex). We can simply accept `$routeProvider` as an argument to the config function and Angular will understand that we've asked for it.

```
app.config(function ($routeProvider) {
    /* Now we can use the routeProvider! :D */
});
```

Configuring Routes

Now we can setup our routes using the `$routeProvider` in the `.config()` callback, see how we accept it as an argument and Angular will give us what we need:

```
app.config(function ($routeProvider) {
    $routeProvider
        .when('/inbox', {
```

```
      templateUrl: 'views/inbox.html',
      controller: 'InboxCtrl',
      controllerAs: 'inbox'
    })
    .when('/inbox/email/:id', {
      templateUrl: 'views/email.html',
      controller: 'EmailCtrl',
      controllerAs: 'email'
    })
    .otherwise({
      redirectTo: '/inbox'
    });
});
```

The `$routeProvider` is really declarative and easy to use, we just declare what view template to use when the URL is pointing to a particular path. Working hand in hand with the `ng-view` attribute we previously set in our `index.html`, these template files will now be injected for the described routes.

Our app will have an inbox view, and a single view for injecting the clicked on Email into.

The first view will be injected at `/inbox`, and the second will be `inbox/email/:id`. You'll notice the second route has `:id` at the end - this is a dynamic view. It means that an object with an id as a property (with a value) will be used in the URL, which makes all views dynamic. We'll then use this to make a server call to get the email that corresponds with the id.

If you look closely, you'll notice each view has a particular Controller. Later versions of Angular (we're using one of the latest) ship with a new Controller syntax, the `"Controller as"` syntax, which instantiates the Controller like an Object and binds it to the current scope under a namespace. The namespace we've chosen for `InboxCtrl` is `inbox`.

Note: You can also declare "Controller as" in-line. It'd look like this: `ng-controller="InboxCtrl as inbox"`.

Controllers

Connecting the model and the view

Controllers are the middleman between the Model and the View, they drive the Model and View changes. Imagine that the controller is given some html from the route and a javascript object from the dependency injection; with these two things, the controller will tell the view (the html) what it can do by giving it scope variables and maybe a few functions.

Let's take a peek at what a Controller looks like.

A good Controller will have as little logic in it as possible, and should only be used for two things: Binding the Model to the View (initializing the View) and adding helper functions to the View.

```
app.controller('InboxCtrl', function () {
  // Model and View bindings
  // Small helper function not needed anywhere else
});
```

If you go through the Angular documentation examples (available at AngularJS.org) you'll notice Model data being declared in the Controller. While this is okay for examples, the Controller easily becomes the Model as well - which is very bad for many reasons:

- All the pieces start to get more coupled together
- More difficult to share business logic
- Makes things difficult to test

Remember: A good Controller will have as little logic in it as possible.

Each controller has access to a `$scope` and some html. `$scope` is the most documented way of adding properties and methods to our view. Remember how we said each 'ng-controller' attribute specifies a scope of HTML to manage? Well, that scope has access to the same `$scope` as the Controller function.

Note: `$scope` isn't the only way to pass data to the front end. Many developers use a combination of the "Controller As" configuration options along with the `this` keyword. For the purpose of this tutorial, we will stick with `$scope` as it's been around for much longer than 'Controller As'.

```
app.controller('InboxCtrl', function ($scope) {
   // initialize the title property to an array for the view to use
   $scope.title = "This is a title";
});
```

Note: Notice we're injecting `$scope` inside the function.

We can then use this like so:

```
<div ng-controller="InboxCtrl">
   {{ title }}
</div>
```

Note: Here we're accessing the title directly, however it is encouraged to always have at least one dot (.) in our view expression properties. Using the "controller as" with `this` syntax would solve this giving us the `.` like so . More info here.

In order to keep our controllers more reusable, we would hook up data in our controller via a Factory or Service.

✓ **Code check:** 03-application-controller

Note: To run this code check you'll need to:

- Make sure you've downloaded the code. Do this by going here and either cloning the reop or clicking "Download Zip".
- In your terminal, navigate to the project folder (e.g. `/Users/carl/Downloads/guide-intro-to-angular/app/03-application-controller`)
- Run a simple local server. On a Mac, you can do this by running `python -m SimpleHTTPServer`. If you're on windows, try doing this by installing Mongoose.
- In your browser, go to `http://localhost:8000/`

Factories

What's a Factory?

Angular Factories can be used for many different things. Some of the most common use-cases are server-side communication via HTTP and abstracting Models to persist application state and changes across controllers. Angular Factories are a great way to create reusable features and code blocks throughout our application.

Key takeaway: If you want to communicate with a RESTful API, do it through a factory! If you want to store a 'CurrentUser' with authentication information, do it in a factory!

Note: You may have heard of factories as a design pattern in many programming languages, but Angular's factories are different to those in practice (maybe not in spirit :).

You can create a factory using the `angular.factory()` method like so:

```
app.factory('ExampleFactory', function ExampleFactory($rootScope, $http, $location) {
   return function myReusableFunction() {
      // do something fancy
   };
});
```

Notice how we're injecting a few different things here, such as: `$rootScope`, `$http`, `$location`. We'll cover these in a bit.

It's often good practice to create an `exports` Object inside your Factories and return it. This helps with explicit internal naming which helps you see which methods / variables are private or not (closures ftw).

In this app, we'll need to get our messages, so let's create a function to do that. Angular uses the `$http` service to communicate with the server, so we'll inject it:

```
app.factory('InboxFactory', function InboxFactory ($http) {
   var exports = {};

   exports.getMessages = function () {
      return $http.get('json/emails.json')
         .error(function (data) {
            console.log('There was an error!', data);
      });
   };

   return exports;
});
```

$http

This will use $http to make a GET request to our "json/emails.json" file; here we also set a default error handler for the http request by chaining a method onto the promise returned by `$http.get()`. Yes that's right, `$http()` returns a promise! So we can use all of the Angular promise API, the same as we would use a `$q.defer().promise` object, with a few extras: namely `error(fn)` and `success(fn)`. `success(fn)` and `error(fn)` are two "sugar" methods that are very similar to `then(fn)` and `catch(fn)` but specific to `$http`. Don't worry if this goes a little over your head- we'll go into promises more in the next section!

Note: What are "sugar" methods? In simple terms, syntactic sugars are a feature in a programming language that lets you express an idea in a more convenient way.

You can view Angular's $http documentation here.

✓ **Code check:** 04-inbox-factory

Note: to run this Code check you'll need to:

- Make sure you've downloaded the code. Do this by going here and either cloning the repo or clicking "Download Zip".
- In your terminal, navigate to the project folder (e.g. `/Users/carl/Downloads/guide-intro-to-angular/app/04-inbox-factory`)
- Run a simple local server. On a Mac, you can do this by running `python -m SimpleHTTPServer`. If you're on windows, try doing this by installing Mongoose.
- In your browser, go to `http://localhost:8000/`

Promises

Connecting our Controller and Factory T ogether

Promises are very important inside Angular, they allow you to organize functions that take a long time to do things (e.g. HTTP requests). Promises in Angular are implemented with `$q`. The `$q` implementation was inspired by Kris Kowal's Q.

To be honest, `$q` is quite a strange beast. Here's a great cartoon explaining promises. If you'd like to see another practical example, checkout the Angular $q documentation.

This is how promises work:

- "Do something when this HTTP request — or another function that takes a long time to complete — has finished"
- "If it all goes well, please do the `success` function I give to `.then()`"
- "If it goes wrong, please do the `catch` function I give to `.then()`"

Sample code:

```
var deferred = $q.defer();
   deferred.promise.then(
      function whenThingsGoSunny(){},
      function whenThingsDontGoSoSunny(){}
   )
```

Notice that the first function we pass is the `success` function and the second one is the `error` function.

Promises have quite a few interesting behaviors that you can read about here.

When making HTTP requests in Angular you'll often use the `$http` service which is based on `$q`.

As we just saw, `$q` allows you to use `.then()` to include a `success` and an `error` function. When using `$http`, you can do something very similar (but without the `.then()`:

```
$http({method: 'GET', url: '/someUrl'})
   .success(function(data, status, headers, config) {
      // this callback will be called asynchronously
      // when the response is available
   })
   .error(function(data, status, headers, config) {
      // called asynchronously if an error occurs
      // or server returns response with an error status.
   });
```

Read the $http documentation for more details.

Hooking up the Factory and Controller

We've got some basic email data instide a Factory setup, and a Controller, so let's connect the two:

```
app.controller('InboxCtrl', function($scope, InboxFactory) {
   InboxFactory.getMessages()
      .success(function(jsonData, statusCode) {
         console.log('The request was successful!', statusCode, jsonData);
         // Now add the Email messages to the controller's scope
         $scope.emails = jsonData;
   });
});
```

See how the `InboxFactory` factory is available as an injectable in our controller? That's dependency injection helping us out again!

We then call the `getMessages` method on the factory and using the `$http()`'s `success` method (`$http()` was returned from `getMessages()`), we can then add the list of emails / messages to our controller's `$scope` and use it in the view. i.e. The `success(fn)` is available from chaining when we do `http()` or `http.get()`, etc...

Templating

Rendering our Views

Let's talk a little more about the View files we've been using. You may have noticed that inside our template we were using the properties that we added to scope directly inside . These are called expressions and we can put some basic JavaScript in there too, as well as a bunch of filters.

```
<!-- JavaScript inside expression -->
<h1>{{ [ firstName, lastName ].join(" ") }}</h1>
<!-- currency filter applied to a multiplication of 2 numbers -->
<div class="total-info">{{ cost * quantity | currency }}</div>
<!-- Using a ternary expression inside the expression -->
<div class="budget">{{ ( total > budget ) ? "Too Expensive" : "You can buy it!" }}</div>
```

Angular will interpolate these expressions: above we have made use of 3 little examples of how powerful expressions are.

In the first one, we create an array and put two String `$scope` variables inside it, then we join the two together with a `" "` to display the name in a `<h1>`.

In the second, we multiply two Number `$scope` variables together and then apply a currency filter on them. You can read more about filters here in the Angular documentation.

In the third line of code we make use of a JavaScript ternary operator inside the expression to check whether the Number `$scope` property `total` is greater than the budget property and display the appropriate message in the div.

The JavaScript inside an assigned controller would be as so:

```
$scope.firstName = "John";
$scope.lastName = "Doe";
$scope.cost = 1;
$scope.quantity = 2;
$scope.total = 3;
$scope.budget = 4;
```

These templates are an essential aspect to Angular as we can not only make the mappings of data that you've seen already, but we can also introduce custom elements completely! In Angular these are called directives and come with their own templates (either in another html file or a string, the same as how controllers operate).

As mentioned before, it often helps to see the code in action:

Code check: 05-templating

View the result by downloading the Code check and opening the `index.html` file in your browser. Play around with the `$scope` values inside `index.html` and refresh your page to see how everything updates.

Directives Overview

Custom HTML Elements

Directives are Angular's take on custom HTML elements. They offer a very reusable way to encapsulate data, templates, and behaviors.

In our View file we might have something like this:

```
<div id="someview">
   {{ data.scopePropertyAsUsual }}
   <my-custom-element></my-custom-element>
</div>
```

Where the `<my-custom-element>` above will be injected with our directive template and logic. Here's what the basic directive structure might look like:

```
app.directive('myCustomElement', function myCustomElement() {
   return {
      restrict: 'EA',
      replace: true,
      scope: true,
      template: [
         "<div>",
         "      <h1>My Custom Element's Heading</h1>",
         "      <p>Some content here!</p>",
         "      <pre>{{ ctrl.expression | json }}</pre>,"
         "</div>"
      ].join(""),
      controllerAs: 'ctrl',
      controller: function ($scope) {
         this.expression = {
            property: "Example"
         }
      },
      link: function (scope, element, attrs) {}
   }
});
```

As you can see the name of the directive is camelCased (`myCustomElement`) whereas the html element is hyphen-separated (`<my-custom-element></my-custom-element>`). This is an angular convention for directive naming.

Next, we return a JavaScript Object with various properties on it to describe how the directive works. Here's a quick run through of what some of the Directive properties mean (feel free to skim through this):

- `restrict`: allows you to restrict how the element is declared, `E = Element`, `A = Attribute`, there are other ways of declaring (such as a comment) but I wouldn't recommend them as they're not that friendly with older browsers.
- `replace`: replaces the Directive's root element, in this case would be `<inbox></inbox>`.
- `scope`: tells Angular to use an isolated or inherited scope, these concepts are quite tricky to grasp, but setting to `true` inherits the parent scope and keeps sibling directives more independent, which makes things easier to work with. There are cases where isolated scopes (`scope: {}`) are the better choice depending on what we're doing.
- `template`: I've recently used an `[].join('')` style for templating, it's much cleaner to work with and we can start the first line of the template on a new line, rather than having a huge indent.

- `templateUrl`: the same as `template`, but will point to a template file such as `tmpl/inbox-directive.html` instead of being a string.
- `controllerAs`: creates a namespace for our Controller when instantiated
- `controller`: inject dependencies and bind logic, you may also expose an API for other directives to interact with this one through the `'controllerAs'` and `this` of the controller.
- `link: a place to write non-Angular logic, but tie it in with Angular. You'll have access to the current scope, the template element root (first element in template rather than Directive root) and the attributes on the element declared.`

These are some of the most commonly used properties, though you can check the documentation (found under the `$compile` section of the API) for other properties. You can also browse Angular's documentation on directives.

Note: Here's a great video introducing Angular directives.

Directives are not just custom elements, they are much more... you can use them for all sort of custom logic, especially when you want to reuse some code. (Gotta keep it DRY.)

Factories and Directives Working Together

Completing our Directive

In our demo app, here's what our `InboxFactory` looks like:

```
angular.module('EmailApp')
   .factory('InboxFactory', function InboxFactory ($q, $http, $location) {
      'use strict';
      var exports = {};

      exports.messages = [];

      exports.goToMessage = function(id) {
         if ( angular.isNumber(id) ) {
            // $location.path('inbox/email/' + id)
         }
      }

      exports.deleteMessage = function (id, index) {
         this.messages.splice(index, 1);
      }

      exports.getMessages = function () {
         var deferred = $q.defer();
         $http.get('json/emails.json')
            .success(function (data) {
               exports.messages = data;
               deferred.resolve(data);
            })
            .error(function (data) {
               deferred.reject(data);
            });
         return deferred.promise;
      };

      return exports;
   });
```

And here's what our finished Directive looks like:

```
app.directive('inbox', function () {
   return {
      restrict: 'E',
      replace: true,
      scope: true,
      templateUrl: "js/directives/inbox.tmpl.html",
      controllerAs: 'inbox',
      controller: function (InboxFactory) {
         this.messages = [];
         this.goToMessage = function (id) {
            InboxFactory.goToMessage(id);
         };
         this.deleteMessage = function (id, index) {
            InboxFactory.deleteMessage(id, index);
         };
         InboxFactory.getMessages()
            .then( angular.bind( this, function then() {
               this.messages = InboxFactory.messages;
            })  );
      },
      link: function (scope, element, attrs, ctrl) {
         /*
         by convention we do not $ prefix arguments to the link function
         this is to be explicit that they have a fixed order
         */
      }
   }
});
```

Now we can use this directive anywhere in our application's scope by creating an `<inbox></inbox>` element (using `restrict: 'E'` to specify an element directive).

When the application runs, Angular will replace the `<inbox>` element with the template at the `templateUrl`. We then make an alias for the controller under the name of `inbox`. This alias is then accessible inside the controller as `this` and inside the template as `inbox`. If you look inside the template you'll see expressions like `inbox.messages` and `inbox.deleteMessage(id, $index)`. These are the same `this.messages` and `this.deleteMessage` we can see in the InboxFactory.

Finally we have a link function that will run straight after the controller runs. The link function will then receive the aliased controller as the fourth argument, here we named it `ctrl`. Yes that's right, the link function has fixed positions for it's arguments i.e. scope is always first.

Note: This is different behavior to the controller's arguments which are injected and therefore can take any order. For the link function we don't use the $ prefix for scope, element and attributes to make it clear that they aren't under the control of dependency injection.

Here's the complete HTML template (view) for the directive:

```
<div class="inbox">
  <div class="inbox__count">
    You have {{ inbox.messages.length && inbox.messages.length || 'no' }} messages
  </div>
  <div ng-hide="!inbox.messages.length">
    <input type="text" class="inbox__search"
      ng-model="inbox.search"
      placeholder="Search by 'from', e.g. TicketMaster">
  </div>
  <ul class="inbox__list">
    <li ng-show="!inbox.messages.length">
      No messages! Try sending one to a friend.
    </li>
    <li ng-repeat="message in inbox.messages | filter:{ from: inbox.search }">
      <div class="inbox__list-info">
        <p class="inbox__list-from"> from: {{ message.from }} </p>
        <p class="inbox__list-date"> {{ message.date | date: 'dd/MM/yyyy' }} </p>
        <p class="inbox__list-subject"> {{ message.subject }} </p>
      </div>
      <div class="inbox__list-actions">
        <a href="#" ng-click="inbox.goToMessage(message.id);">
          Read
        </a>
        <a href="" ng-click="inbox.deleteMessage(id, $index);">
          Delete
        </a>
      </div>
    </li>
  </ul>
</div>
```

Notice the use of built-in angular directives such as `ng-hide`, `ng-show`, `ng-click`, `ng-repeat` and `ng-model`. We won't go into much detail about these directives as each can be quite a big topic but you can read about them here.

✓ **Code check:** 06-first-directive

Note: To run this Code check you'll need to:

- Make sure you've downloaded the code. Do this by going here and either cloning the repo or clicking "Download Zip".
- In your terminal, navigate to the project folder (e.g. `/Users/carl/Downloads/guide-intro-to-angular/app/06-first-directive`)

- Run a simple local server. On a Mac, you can do this by running `python -m SimpleHTTPServer`. If you're on windows, try doing this by installing Mongoose
- In your browser, go to `http://localhost:8000/`

The 3 files we just saw are located here:

- Inbox Factory: `js/factories/InboxFactory.js`
- Inbox Directive: `js/directives/inbox.js`
- Inbox Template: `js/directives/inbox.tmpl.html`

Built In Directives

ng-show, ng-repeat, and ng-click

Angular ships with some really useful Directives, such as `ng-show`, `ng-repeat` and `ng-click`. There are many built-in Directives, and more get added each release to help enhance how we build our apps.

Our Directives will evaluate based on data. For example `ng-show="someData"` will evaluate when the value of `someData` is `true`. If the value is `false`, it will hide the element.

One of the most powerful Angular Directives is `ng-repeat`, which iterates over Objects (or items) in an Array:

```
<ul>
  <li ng-repeat="item in items">
    {{ item.name }}
  </li>
</ul>
```

You may have spotted this in our directive's template where we iterate over our messages and also filter them based on the `search`.

```
<li ng-repeat="message in inbox.messages | filter:{ from: inbox.search }">
```

But where did `inbox.search` come from? Well this was created by the `ng-model` directive that we attached to an input field.

```
<input type="text" class="inbox__search" placeholder="Search"
  ng-model="inbox.search">
```

Now, whenever someone types into this field, it will set the `inbox.search` property and update the `ng-repeat`'s filter. Pretty wild, right?!

So all these `ng-` attributes are directives created just the same as how we make our own directives. We can use them anywhere in our application.

✓ **Code check:** 07-ng-repeat

View the result by downloading the Code check and opening the `index.html` file in your browser. Play around with the `fruits` values inside `index.html` and refresh your page to see how everything updates.

Cloaking

Note: Cloaking is a bit of an advanced trick but we thought you might enjoy it :) Feel free to skip over this section if you'd like.

Angular has a really neat built-in feature called cloaking, which you can declare on an Element to tell Angular it needs cloaking (hide it) whilst it's still rendering. Cloaking allows you to hide any handlebars you might see flicker before Angular has rendered and loaded your data. E.g. before the user data has had time to load.

When `lib/angular.js` has loaded, AngularJS automatically appends an inline `<style></style>` element to your document with cloaking styles to hide elements during load. This can sometimes take a second to load as it's got a JavaScript dependency too, so I recommend setting up the `<head></head>` of your document manually like this:

```
<!doctype html>
<html ng-app="app">
  <head>
    <style>
    [ng\:cloak], [ng-cloak], [data-ng-cloak], [x-ng-cloak], .ng-cloak, .x-ng-cloak {
      display: none !important;
    }
    </style>
  </head>
  <body>
  </body>
</html>
```

This is 100% reliable and hide any cloaked elements immediately, even without Angular loaded. Applying cloaked styles is easy:

```
<div ng-cloak>
   {{ someData }}
</div>
```

Debugging

Debugging in Angular can be very difficult and frustrating. This section will give you the foundation you need to simplify the process.

Stack Trace Function Names

Angular uses a lot of anonymous functions in the call stack. When these throw an error (yes, you'll see a lot of errors during development), to make things easier to debug, I've named anonymous functions.

Here it is before:

```
app.factory('InboxFactory',
   function ($q, $http) {}
):
```

...and now after:

```
app.factory('InboxFactory',
   function InboxFactory ($q, $http) { }
):
```

This will then show errors `at InboxFactory` rather than `at anonymous`. A handy tip for developing Angular apps.

Routing Problems

Another debugging tip when working with routes is to listen for Routing Events to be triggered on the `$rootScope`. A suitable place for this is the `run` function, found here, which can be thought of as a 'controller for the module', but try to keep this section as small as possible!

```
app.run(function($rootScope) {
   $rootScope.$on('$routeChangeError', function(event, current, previous, rejection) {
      console.log(event, current, previous, rejection)
   })
});
```

The above example sets an event listener for any errors that occur during a route change.

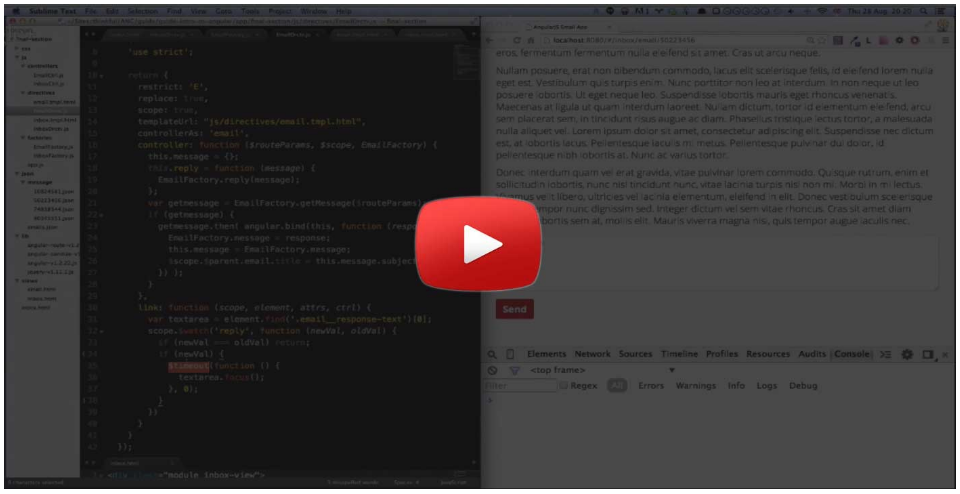
Debugging Scopes

Here's a great article for debugging scopes.

Complete Project

A working Gmail Clone

Make your own Gmail clone by following the code-along video below:



```
    'use strict';

    return {
      restrict: 'E',
      replace: true,
      scope: true,
      templateUrl: "js/directives/email.tmpl.html",
      controllerAs: 'email',
      controller: function ($routeParams, $scope, EmailFactory) {
        this.message = {};
        this.reply = function (message) {
          EmailFactory.reply(message);
        };
        var getmessage = EmailFactory.getMessage($routeParams)
        if (getmessage) {
          getmessage.then( angular.bind(this, function (respo
            EmailFactory.message = response;
            this.message = EmailFactory.message;
            $scope.$parent.email.title = this.message.subject
          }) );
        }
      },
      link: function (scope, element, attrs, ctrl) {
        var textarea = element.find('.email__response-text')[0];
        scope.$watch('reply', function (newVal, oldVal) {
          if (newVal === oldVal) return;
          if (newVal) {
            $timeout(function () {
              textarea.focus();
            }, 0);
          }
        })
      }
    }
  });
```

We've made a bunch of changes to the project to give it a more complete feel, here's a list of what we did:

1. We added a new route for specific Email messages that accepts an `:id` parameter
2. We added some slick CSS styles :)
3. We changed our controllers to all make use of the 'Controller As' syntax
4. We added the ngSanitize library to use the `ng-bind-html` directive in the Email directive
5. We uncommented the `goToMessage` method in our Inbox Factory that will navigate to the new route
6. We simplified our controllers further so they now only use the title on 'this' (Controller As)
7. We added the Email Controller and view
8. We added the Email Factory which adds a bunch of factory and JavaScript alerts where server communication would be appropriate
9. We added the Email Directive and its template as well as using it in our Email View
10. We added new JSON files for each of the messages
11. We added the `ng-cloak`, however it isn't used in this application as `ngView` covers this for us :)

I would encourage you to have a read through the code and try to understand what's happening. You may of course find much more suitable ways to accomplish the same results to your own liking, this is just an example for learning purposes but we've been sure to include a number of good thinking points for you to work from.

You may be able to find some other little tricks we've included such as `$timeout`, `ng-bind-html` and last but certainly not least `$watch`.

Code check: final-section

Note: To run this code check you'll need to:

- Make sure you've downloaded the code. Do this by going here and either cloning the reop or clicking "Download Zip".
- In your terminal, navigate to the project folder (e.g. `/Users/carl/Downloads/guide-intro-to-angular/app/03-application-controller`)
- Run a simple local server. On a Mac, you can do this by running `python -m SimpleHTTPServer`. If you're on windows, try doing this by installing Mongoose.
- In your browser, go to `http://localhost:8000/`

Thinking Angular

When moving from something like website development to Angular you can be tempted to use jQuery selectors (since you already know how to use jQuery). You need to avoid doing that. Angular takes care of the DOM. Do not modify it yourself, unless you really need to or are using a plugin. Anything you'll likely want to do, Angular can do.

Getting the value of a textarea? You might be tempted to do this:

```
<textarea></textarea>
<script>
    var elem = document.querySelector('textarea');
    var value = elem.value;
</script>
```

With Angular, the `ng-model` will keep this updated for us, we just reference it:

```
<textarea ng-model="myModel"></textarea>
<script>
    app.controller('SomeCtrl', function SomeCtrl($scope) {
        // binds and keeps the value updated at all times
       // no need to re-query the value at any time
       // which means we can pass the value straight
       // back to the server for example
       $scope.myModel = '';
    })
</script>
```

You can read more about two-way databinding here

Thinkful AngularJS Course

If you're interested in learning more about AngularJS, you should take a look at our mentor-led AngularJS Course.

Get notified when new guides are released

Enter your email address here

Send Me Your Next Guide »

Guides you might enjoy

- Javascript Best Practices, Part 1
- Intro to jQuery
- GitHub Pull Request Tutorial

Created by THINKFUL