# AngularJS Workbook

## Exercises take you to mastery

**By Nicholas Johnson**

**version 0.1.0 - beta**

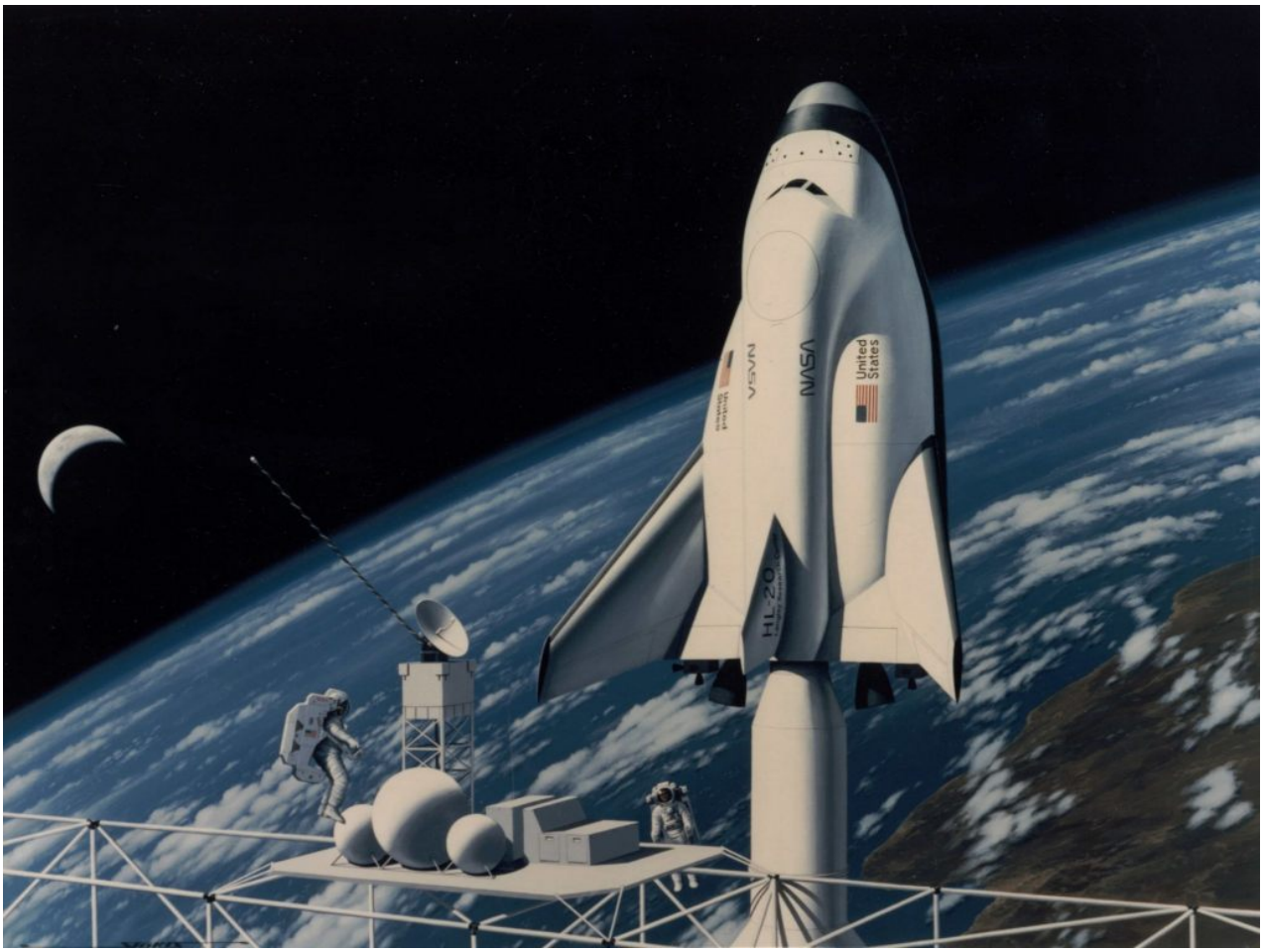


*Image credit:* *San Diego Air and Space Museum Archives*

# Welcome to the Angular Workbook

This little book accompanies the Angular course taught over 3 days. Each exercise builds on the last, so it's best to work through these exercises in order.

We start out from the front end with templates, moving back to controllers. We'll tick off AJAX, and then get into building custom components, filters, services and directives. After covering directives in some depth we'll build a real app backed by an open API. By the end you should have a pretty decent idea of what is what and what goes where.

# Template Exercises

The goal of this exercise is just to get some Angular running in a browser.

## Getting Angular

We download Angular from angularjs.org Alternately we can use a CDN such as the Google Angular CDN.

## Activating the compiler

Angular is driven by the template. This is different from other MVC frameworks where the template is driven by the app.

In Angular we modify our app by modifying our template. The JavaScript we write simply supports this process.

We use the ng-app attribute (directive) to tell Angular to begin compiling our DOM. We can attach this attribute to any DOM node, typically the html or body tags:

```
1    <body ng-app>
2      Hello!
3    </body>
```

All the HTML5 within this directive is an Angular template and will be compiled as such.

---

## Linking from a CDN

Delivering common libraries from a shared CDN can be a good idea. It may help our initial pageload as Angular may already be cached in the user's browser. This may also help our SEO as Google cared about initial download times.

However, our project is now tied to Google's uptime, and we can't easily develop on the train.

---

## Exercise - Hello Universe

Let's start with the simplest thing possible. Hello World.

1. Download Angular latest uncompressed from here: https://angularjs.org/
2. Concatenate the strings "Hello" and "World" to make a hello world app.
3. Create an Angular template which tells you how many seconds there are in a day, a year, a century.
4. Find out how many weeks there are in a human lifetime.

## Extension - if you are first to finish

You've downloaded Angular. Open it in your editor and have a quick browse through the Angular codebase. You'll find it clean, and well commented and easy to scan.

# Binding

Angular features super easy two way binding. Binding a value means setting up listeners so when the value changes, the front end updates to match, and when the front end is changed (say using a form) the saved value is updated.

## $scope

Angular uses a special object called $scope to store data. All bound values are actually attributes of the $scope object. We will look at $scope in some detail shortly.

## ng-model

We can bind an input to a model using the ng-model directive. This automatically creates watchers (listeners) on the input element. When the input element changes the watchers fire and set an attribute of a special object called $scope.

```
1    <body ng-app>
2      <input ng-model="test" />
3    </body>
```

$scope is available elsewhere in the view, and also in the controller as we shall very soon see.

## Curlies

We can output any attribute we have in $scope using the handlebars curly brace syntax. When we do this, Angular will create more watchers in $scope that recompile the DOM when the 'test' attribute changes.

Like so:

```
1    <body ng-app>
2      <input ng-model="test" />
3      {{test}}
4    </body>
```

## ng-bind

We can also explicitly bind any element to scope using the ng-bind directive. In most circumstances this is entirely equivalent to the curly brace syntax.

```
1    <p ng-bind="test"></p>
```

## Exercise - Evil Siri

You are a crazy scientist building an AI for world domination. For reasons known only to you, you have chosen Angular to help you in this epic task. Your first job is to teach your new creation to greet you by name.

Write an Angular template which contains an "enter your name" form. As you type, it updates the page with an h1 tag, like this:

```
<h1>
  Welcome Dave. How may I serve you today?
</h1>
```

## Exercise - Handy instant maths calculators

As an evil genius, you need a calculator to help you work how many helidrones you will need to steal the houses of parliament.

Let's build one now. There are two input fields, each with a number, like this:

```
<input placeholder='Enter an evil number' />
<input placeholder='And another' />
```

Use the curly brace syntax or ng-bind to output the sum of the two numbers.

When you fill in the fields, output a little list in HTML that looks something like this:

```
<ul>
  <li>
    2 + 2 = 4
  </li>
  <li>
    2 - 2 = 0
  </li>
  <li>
    2 * 2 = 4
  </li>
  <li>
    2 / 2 = 1
  </li>
</ul>
```

- 3 + 2 = 5
- 3 - 2 = 1
- 3 * 2 = 6
- 3 / 2 = 1.5

When I type in the input fields the list should update in real time.

- You might like to extend this to create a VAT calculator, a currency converter or a unit converter.

## Exercise - Time remaining

In the last section we wrote code to let us see the number of seconds in a year.

Add a little input box that lets us type in a your age in years.

Output the number of weeks in your life so far. Have it output the approximate number of weeks remaining. Output a percentage completion.

# More on binding

We can bind an attribute of $scope to anything in the DOM, an element, an attribute, a class name, even a portion of an attribute value.

## Binding to an attribute

Say we have a value bound to an input element. We could use that value to form part of a style attribute:

```
1    <input ng-model="color" />
2    <p style="color{{color}}">Hello</p>
```

## Binding to a class

We can use values in scope to construct a class attribute using the ng-class directive. A class attribute can have many values, so we pass an object and all values that evaluate to true are included in the compiled class.

```
1    <body ng-class="{red: color=='red', blue: color=='blue' }>"
```

## Binding visibility

We can conditionally show and hide elements using ng-show.

```
1    <input type="checkbox" ng-model="happyCat" />
2    <div ng-show='happyCat'>Cat is happy</div>
3    <div ng-show='!happyCat'>Cat is grumpy</div>
```

## Exercise - Profile Widget

You are an entrepreneur, creating the next LinkedIn. Users of your awesome service have said that they would like to be able to choose a profile picture just by entering a URL.

We are going to allow people to type in a profile picture url and have that picture appear immediately. Create a field and bind it to a model called something like "avatarUrl".

Now use the avatarUrl as the src attribute of an img tag, like so:

```
1    <img src="" />
```

## Exercise - Profile form

Extend your app so you can type in your name. This will update the the alt text on the image, and an h2 somewhere on the page.

Add in a description textarea. This will update the a paragraph of description text under the image.

If you were to back this with an API, this could be a real site component.

## Exercise - Make the form optional

We are going to hide the edit fields when they are not needed so that we either see the edit form or the profile.

Add in a checkbox. Bind it using ng-model. When the checkbox is checked, the edit fields are visible. When is is not checked, the edit fields are hidden. Use ng-show to make this work.

## Optional Exercise - Tabs

Use ng-show to create a simple navigation. Your page will contain several sections, each with an ng-show attribute. Choose a horizontal or vertical nav. Use ng-click to set the value of a model and show the sections.

## Further reading

- input[checkbox] - https://docs.angularjs.org/api/ng/input/input%5Bcheckbox%5D
- ngClass - https://docs.angularjs.org/api/ng/directive/ngClass
- Check out the other built in Angular directives here: https://docs.angularjs.org/api/ng/directive

# Controllers

All our data (models) are stored in $scope. $scope is an object managed by Angular in which we store our models. We can also store helper functions here. We can even store the controller itself in $scope. We can store any type of data in $scope including functions. This data will be available in our template.

$scope is available in two places, the template, and the controller.

In the controller we can manipulate $scope. we can initialise values and add helper functions. It provides an interface between the HTML and any JavaScript you might write.

## The purpose of the controller is to initialise $scope

**The primary job of the controller is to initialize $scope.** If you find your controller getting large and trying to manage too much you probably need to refactor.

## Creating a controller

A controller looks something like this:

```
angular.module('app', [])
  .controller("demoController", function($scope) {
    $scope.hello = "World";
  });
```

We hook this in to our template something like this:

```
<div ng-app="app" ng-controller="demoController">
  {{hello}}
</div>
```

## Controller Persistence

We can hold onto the controller if we want, but under many circumstances it's sufficient to just initialise $scope and then let it disappear.

However it is possible to get hold of a persistent scope object using the controller-as syntax. More on this later.

## When to create a controller

I generally expect to create one controller per unit of interaction. This means we typically create quite a lot of controllers. For example you might have a loginFormController, a profileController, a signupController, etc. Many small controllers is better than one massive multi-purpose monolith.

## Controller scope

A controller will apply to an html5 element and all of it's children. We can nest controllers inside

each other, in which case the child controller takes precedence if there is a conflict. This is decided using prototypical inheritance. More on this when we get to the section on $scope.

## Exercise - Control yourself

We are going to add a profileController to your super advanced profile form from the previous exercise.

1. Create a profileController. Use ng-controller to add it to your profile form.
2. In your controller, set a default name, description and profile picture url.

## Exercise - Helper function

Extend the calculator exercise. We are going to create a function to allow us to zero all the values.

1. Create a function in your controller which zeros number1 and number2. Add it to $scope. It is now available in your front end.
2. Add a button to your DOM and use ng-click to call the function.

Remember you never need to refer to $scope in your template. $scope is always implied.

## Further Reading

Read the controller guide here: https://docs.angularjs.org/guide/controller

# Scope

$scope is a shared object which we use to pass data to our front end. We can store objects such as models, helper functions and even the controller.

$scope is a tree structure that loosely follows the structure of the DOM. Whenever we use the ng-controller directive we get a new $scope object which prototypically inherits from it's parent.

When we use Angular to set a value on a model, all the bindings are fired automatically.

## Exercise - Multiple profile widgets

Extend the profile exercise from the previous section. We should be able to have multiple profile widgets on the same page. Copy and paste a few out. Type in one. Notice how they are separate units of interaction, each with it's own controller and $scope.

## Exercise - Nested controllers and controller isolation

Instead of having the profile widgets next to each other, nest them inside one another. Have a play and see how they interact. This might be surprising for the user.

Modify your controller so that all of the $scope attributes which make the form work are nested inside a JSON object called profile. You can initialise this in your controller like this:

```
1    .controller('profileController', function($scope) {
2      $scope.profile = {}
3    })
```

In your front end you can write code like this:

```
1    <p>{{profile.name}}</p>
2    <img ng-src="{{profile.url}}" />
```

You should now have prevented undesirable scope inheritance. Verify this is indeed the case.

## Harder Exercise - Storing data on the controller

You can use your controller as a data store, like this:

```
.controller('myController', function($scope) {
  var vm = $scope.vm = this;
  vm.profile = {}
})
```

We are now using the controler as a handy model store. In your controller store your controller (this) in $scope. Now add your person model as an attribute of this.

You can now access your controller and associated model in your view, like this:

```
<p>{{vm.person.name}}</p>
```

Try this out now.

## Reading

Read the scope documentation here: https://docs.angularjs.org/guide/scope

# Watching and Applying

We can watch an attribute of $scope using:

```
1    $scope.$watch('test', function(newVal, oldVal) {});
```

Now whenever the value of $scope.test changes the function will be called.

## Watchers and the digest cycle

Watchers are added to the $scope.$$watchers array. This array contains a list of all current watchers, the expression they are evaluating and the last known value of that expression.

Whenever an Angular watcher fires it triggers something called a $scope digest. This means that the $scope tree is traversed and all the watch expressions are evaluated. If any come up 'dirty' i.e. changed the corresponding functions are executed.

## Digest limit

The digest cycle will repeat over and over until the entire $scope hierarchy is clean or the digest limit is reached. The default digest limit is 10.

## Exercise

We are going to hack some quick validation into our profile form. We'll see the right way to do validation using directives in a bit.

Extend your profile form with a $scope.person.name field. Let's make name mandatory. use $scope.$watch to watch the 'person.name' property of scope.

If the value is not blank, set $scope.errors = false.

Otherwise set $scope.errors = {name: "should not be blank."}.

Now in your template, use ng-show to show a nice warning message if errors != false.

## Downloads

- Code from the board

# Dependency Injection

We can inject a dependency into a controller by simply receiving it.

```
angular.module('demoModule', [])
  .controller('demoController', function($log) {
    $log.log('Hi!');
  }
```

## Modules

We can include a module into another module by placing it's name between the square braces:

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

# Exercise - NgDialog

We are going to inject the ngDialog service into a controller. This will allow our controller to create popup dialog boxes.

First go here and grab the ng-dialog.js and ng-dialog.css. Link them in the header of your document in the usual way.

http://ngmodules.org/modules/ngDialog

## Include it in your app.

We need to include ngDialog as a dependency of the app, like this:

```
angular.module('app', ['ngDialog'])
```

## Inject into your controller.

Create a little controller and use the ng-controller directive to hook it to the DOM.

```
<div ng-controller="myController">
```

Inject the ngDialog service into your controller.

```
.controller('myController', function($scope, ngDialog) {

})
```

You now have access to ngDialog.open. Call this according to the documentation to create a dialog box when the page loads:
https://github.com/likeastore/ngDialog#api

e.g.

```
ngDialog.open({
  template: '<p>my template</p>',
  className: 'ngdialog-theme-plain',
  plain: true
});
```

## Exercise - Extension

Create a form. Create a method on scope that opens the dialog. Call the method when the button is pressed.

Have a go at the minification safe DI syntax.

## Further extension

If you finish first, have a read through the DI documentation here:

https://docs.angularjs.org/guide/di

# Unit Testing with Karma

In this exercise we are going to use Karma to unit test a controller.

## Installation

We use Node to install Karma. If you don't have Node installed, you can grab it from here:

http://nodejs.org/

Alternately if you are on a Mac you can install it using Brew.

Once you have it, we use NPM (Node Package Manager) to install Karma, like so:

```
1    npm install -g karma
2    npm install -g karma-cli
```

## Initialise Karma

We need to create an init file in our current directory. Keep all the defaults, but tell Karma to watch *.js.

```
1    karma init
```

We can now start the Karma runner using:

```
1    karma start
```

## Exercise - destroy the helidrones

Return to the evil calculator exercise. Your helidrones have been destroyed by United Nations special forces. You have a reset button which resets the inputs to zero.

Write a test which lets you know if the reset button works.

## Documentation

You can view the Jasmine documentation here:

http://jasmine.github.io/2.1/introduction.html

## Downloads

- Code from the board

# Day 1 Homework:

## Part 1 - Watch a Video

Watch Misco Hevery talk about Angular

https://www.youtube.com/watch?v=khk_vEF95Jk

## Part 2 - Read the Book

Review today's topics in the AngularJS book, also available online:

http://nicholasjohnson.com/angular-book

## Part 3 - Prep your server

From this section onwards we are going to start to need a server. It's not going to be possible to continue accessing our html pages directly from the filesystem.

You could use:

- Apache (OSX / Linux)
- IIS (Windows)
- NodeJS (all platforms)
- Tomcat (Java)
- Rails / Sinatra / Middleman (Ruby)
- CodeKit (OSX)
- etc, etc.

As long as you can access it via an http or https request (not file://)

# Repeat and Filter

Models are typically just plain old JavaScript objects such as arrays or JSON objects. We can iterate over these in the template using:

```
1    <li ng-repeat="item in items"></li>
```

We can pipe through filters to modify the array before rendering it.

```
1    <input ng-model="filterString" />
2    <input ng-model="orderString" />
3    <li ng-repeat="item in items | orderBy:orderString | filter:filterStri
```

## Exercise - List your Assets

As an evil genius, you will have a list of henchmen that you rely on. Create such a list in JSON and add it to $scope from inside your controller.

Now use ng-repeat to loop over the array and output it to the DOM. Add sort and search to help you choose the right villain for the job.

n.b. If you have a particular love for another domain, please feel free to use that.

## Further Exercise - Rank the henchmen

You have a problem. You have to kidnap a rocket scientist and you need to work out the best henchman for the job.

Briefly review the ng-repeat docs here:
https://docs.angularjs.org/api/ng/directive/ngRepeat. Familiarise yourself with the available variables.

Now use the $index variableto show the current position in the sorted array.

## Exercise - Sort the Objects

We are going to apply a sort.

1. Use the orderBy filter to sort the array sensibly. Review the docs here to find out how: https://docs.angularjs.org/api/ng/filter/orderBy

## Exercise - Write Unit Tests

Write a unit test with Karma. All this needs to do is instantiate the controller and check that scope contains the array of objects.

You'll know that:

a) your controller can be instantiated, and b) your scope is set up as you'd hope.

## Exercise Extension

We are going to allow the user to choose which field we sort on.

1. Create a dropdown select element containing several options. Review the documentation if you are not sure how to do this: http://www.htmldog.com/guides/html/beginner/forms/.
2. Bind your select element to a 'sort' model, so when you select an element in it, the sort value is updated.
3. Use the sort model as the expression in the orderBy filter. When you update the sort dropdown, the list should reorder itself.

## Search Exercise

We are now going to add a dynamic search.

1. Create a new input field and bind it to a searchTerm model.
2. Add a filter to allow you to search according to the term.
3. Review the filter documentation here:
   https://docs.angularjs.org/api/ng/filter/filter

## Extra Exercise (if you finish first)

Implement the strict search example in the filter docs:
https://docs.angularjs.org/api/ng/filter/filter

## Further reading

Angular Filters API - https://docs.angularjs.org/api/ng/filter/filter

# The ng-include Directive

From this section onwards we are going to start to need a server. It's not going to be possible to continue accessing our html pages directly from the filesystem.

You could use:

- Apache (OSX / Linux)
- IIS (Windows)
- NodeJS (all platforms)
- Node http-server
- Tomcat (Java)
- Rails / Sinatra / Middleman (Ruby)
- CodeKit (OSX)
- etc, etc.

As long as you can access it via an http or https request (not file://)

## Repeat Exercise

ng-include accepts an expression which can be a string literal, a function call, or a variable in scope.

Modify your repeat and filter exercise to use a template that displays a single cat.

Try using an inline template in a script tag.

## The template cache.

When a template is downloaded, it is saved to the template cache so it available again quickly. It's possible to write directly to the template cache. This allows you to pre-compile portions of your app into JavaScript, making your app more responsive at the expense of a longer initial load time.

We do this in a run block like this. Run blocks are executed once when the app is first initialised:

```
angular.module('templates', [])
  .run(function($templateCache) {
    $templateCache.put('cachedTemplate', '<h1>Lovely caching!</h1>')
  });
```

## Writing to the template cache.

Write your template directly to the template cache. Verify that no AJAX request is made when loading the tab.

# AJAX

We access remote services using the $http service, which we can inject into our controller.

```
var url = 'http://www.mycats.com/api/cats';
$http.get(url);
```

We use a success function to add a success callback. In the callback, we generally push the response in to $scope, then allow the template to take over.

```
$http.get(url).success(function(data) {
  $scope.cats = data
})
```

## Exercise - update your exercise to use JSON

1. Have a look through the $http API documentation here:
   https://docs.angularjs.org/api/ng/service/$http
2. Refactor your repeat and filter exercise to pull data from a JSON file via AJAX. Remember, your JSON must be well formed to work which means double quotes around all the keys.

## Exercise - Karma

Write Karma tests. Use $httpBackend to mock out the AJAX call and verify that the data is retrieved.

# JSONP Exercises

JSONP is a hack which lets us make cross site AJAX requests. We use the $http object to make a request. We use the .success promise to respond when an AJAX request completes.

```
angular.module('app', [])
  .controller('ajaxController', function($scope, $http) {
    $http.jsonp('http://example.com/cats?callback=JSONN_CALLBACK').suc
      $scope.cats = data;
    })
  });
```

Choose one of the following exercises:

## Flickr Exercise

The flickr JSONP endpoint looks like this:

http://api.flickr.com/services/feeds/photos_public.gne?tags=cat&tagmode=any&format=json&jsoncallback=JSON_CALLBACK

Pull the Flickr feed from the internet and display the images.

## Extension

Allow the user to search for a tag.

Test with Karma.

## The Weather - Exercise

You can pull in data from all over the internet. Here's the JSONP url for Open Weather Map:

http://api.openweathermap.org/data/2.5/weather?q=London,uk&callback=JSON_CALLBACK

Use Open Weather Map to get the weather for a specific location

## Extension

Allow the user to search for a city. You can find the Open Weather Map API here:

http://openweathermap.org/wiki/API/2.1/JSON_API

## Extension

Test with Karma.

# Animation with transitions

## Upshot

- To make $ngAnimate work, you must download the ng_animate.js module from http://angularjs.org. Include it in your main app module to make it available.
- The ng_animate service automatically hooks into any CSS elements animations or transitions applied to the element. It causes enter end leave events to happen on a timeout, allowing you to apply CSS effects.

## Transitions

Transitions provide a way to move smoothly from one state to another. For example, a hover effect could be applied slowly giving a pleasing fade.

Transitions are cut down animations. You can't loop them, or set keyframes. They are simple, easy to apply, and work nicely.

To make an attribute transition smoothly from one state to another, use the transition property:

```
1    a {
2      transition:all 2s linear;
3    }
```

You can animate all properties, or just a specific property. You can also specify a timing function.

```
1    a {
2      transition:color 2s ease;
3    }
```

Choose from the following timing functions:

- ease
- linear
- ease-in
- ease-out
- ease-in-out

## Animatable properties

Transitions don't work with all properties, but do with most of the ones you would care about. A full list of animatable properties (for Firefox) can be found here

## Angular Exercise

We're going to extend our Flickr exercise from before by adding animations

1. Download the correct ng_animate module for your version of Angular from here: https://code.angularjs.org/
2. Include the module in your application module
3. Modify your flickr exercise so that the images animate in smoothly

## Further Exercise

Stagger the animation using ng-enter-stagger and ng-leave-stagger

## Full Keyframe Animation

Animation goes further than simple transitions by adding full keyframe based animation to CSS. This has several advantages, particularly in terms of performance. Because the browser is in charge of the animation it can optimise performance in a variety of ways, taking full advantage of the hardware.

## Declaring Keyframes

To make this work, we must first declare the keyframes we want to use, like so:

```
@keyframes pop {
  from {
    font-size:100%
  }
  to {
    font-size:500%
  }
}
```

## Adding our Animation

Once we have declared our animation, we can add it to the page like so:

```
h1 {
  animation-name: pop;
  animation-duration: 3s;
}
```

## Declaring more keyframes

We can add in additional keyframes using percentages:

```
@keyframes pop {
  from {
    font-size:100%;
  }
  50% {
    font-size:1000%;
  }
  to {
    font-size:500%;
  }
}
```

Here, we have added another keyframe at 50% of the way through the animation.

## Exercise

Extend the Flickr search exercise.

Use Animation to make your images pop onto the page dramatically.

Optionally Make use of ng-enter-stagger to boost the effect.

# Filter Exercises

## Upshot

- Filters are provided for us by little factory functions which we define.
- Define a filter using myApp.filter('filterName', function() {});
- the function is a factory which should return another function which acts as the filter.

## For example

```
1   myApp.filter('filterName', function() {
2     return function(str) {
3       return "hello there " + str + "!";
4     }
5   });
```

## Exercise - Reverse Filter

You can reverse a string using something like:

```
1    "abc".split('').reverse().join('');
```

You might like to try this out in a console.

Create a reverse filter that reverses a string.

## Exercise - Filtering the first element from an array

We can filter an array in a similar way. The filter returns a function which receives an array, and returns a modified array.

Create a filter that returns only the first element in the array.

## Exercise - Pagination

Create a filter that returns the first n elements from an array.

## Exercise - Tweets

Tweets often contain #hashtags and @mentions. Write a a filter that takes a raw tweet and hyperlinks the mentions and hashtags.

Something like this:

```
I'm sorry @dave, #icantdothat
```

becomes

```
I'm sorry
<a href="http://twitter.com/@dave">
  @dave
</a>,
<a href="http://twitter.com/search?query=#icantdothat">
  #icantdothat
</a>
```

## Karma

We can get hold of a filter using the $filter injectable service. Make use of this service to write a Karma test for your filter.

## Exercise - Redit style vote filtering

Given an array of comments and votes like this:

```
[
  {
    comment: "I really like cheese",
    votes: 10
  },
  {
    comment: "I'm not so sure about edam though",
    votes: -2
  },
  {
    comment: "Gouda properly rocks!",
    votes: 4
  },
  {
    comment: "I quite like a bit of mild cheddar",
    votes: -19
  },
  {
    comment: "Cheese is just old milk",
    votes: -8
  }
]
```

Create a vote filter that only shows comments which scored over a certain amount.

Hint - use the Array filter method: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter

Now add a number field and bind it to a value. When you change the value, only comments with a higher value will be shown.

for bonus points, use a range slider like so:

```
<input type="range" min="-100" max="100" />
```

# Services Exercises

## Upshot

- A service is a singleton object.
- Once created it will persist and you will get the same object back each time.
- Inject services into controllers using DI.
- Angular provides many services such as $http, $log and $rootScope.
- You can write your own services.

## Sample Code

```
angular.module('app', [])
  .service('github', function( $http ) {
    var url = 'https://api.github.com/events?callback=JSON_CALLBACK';
    this.getEvents = function () {
      return $http.jsonp(url);
    }
  });
```

## Simple Alert Service

1. Review the service documentation here:
   https://docs.angularjs.org/guide/services
2. Create a very simple alert service. You can inject it into your controller, and when you call it, it alerts the value you pass to it.

## Flickr Service

1. Create a Flickr service, that encapsulates the AJAX logic needed to pull data from the Flickr api. Return the $http object so that we can apply callbacks. You can use the code above as a template.
2. return to your Flickr exercise. Inject the Flickr service into your controller and use this instead.

## Further Exercise - Simple shopping cart service

1. Create a service to manage a shopping cart. Give it a cart variable which holds an array of objects.
2. Add an addToCart function to the service which pushes an item into the cart.
3. Write a controller to display the cart. Inject the cart into your controller.
4. In your controller, write a simple method on $scope which calls shoppingCart.addToCart.
5. in your controller, create an attribute of $scope that holds the shoppingCart.cart.
6. Write a view to render the cart.

# Directives

Directives are the places where Angular is wired into the DOM. In a directive we get full access to the DOM including the element, element children and attributes.

Try not to touch the DOM anywhere else. If you find yourself needing to do this, you may need to have a rethink.

## Built in ng directives

Angular comes with many built in directives. ng-repeat and ng-model are examples of directives.

Angular will treat form elements, input elements, hyperlinks, img tags and many others as directives.

You can get most of your work done using the built in directives. Upon finding a new hammer it's normal to treat everything like a nail for a while. Before you get stuck in creating a directive consider if you really need one or if the built in directives will do the job.

## Declaring a directive

We declare a directive by writing a constructor function that returns a definition object, like so:

```
1    angular.module('app', [])
2      .directive('usefulDirective', function() {
3        var directive = {
4          template: "<p>Hello from the directive!</p>",
5          restrict: "A",
6          scope: true,
7          controller: function($scope) {}
8        }
9        return directive;
10     });
```

- template - a string that will be used as a template. This is a real template and can include other directives or curly braces.
- templateUrl - use this instead of template if your template content is web accessible.
- restrict - Accepts the characters "AEC" or any subset thereof. Tells the directive to match an attribute, element or class.
- scope - pass true if you want a new child scope.
- controller - your directive can have a controller which can receive injectors in the normal way.

Use your directive like this:

```
1    <div useful-directive></div>
```

## Directive naming

Because HTML is not case sensitive, directive names are camelCased in the JavaScript, and hyphenated-in-the-html. Angular automatically translates between the two forms.

This is very important to remember. This will bite you if you forget it.

## Precomposing templates

It's common to use Gulp or Grunt to compose your template strings, perhaps from a directory of html.

Precomposing your templates will limit the number of AJAX requests your app needs to make to get hold of the content.

## Directive with Template Exercise

1. Review the Angular directives guide - https://docs.angularjs.org/guide/directive
2. Remember the little profile widget that you made earlier in the course? Create a directive which inserts this widget template onto the page.

If you use the templateUrl option you will need to remember this is making an AJAX call, so you'll need to be running a server.

## Directive with Scope exercise

1. Your widget does not have it's own scope. If you insert it on the page more than once, you will have a problem.
2. Set scope: true on your directive. Now your directive will spawn a new child scope. You can now insert multiple widgets on the page.

## Flickr Directive Exercise

We are going to create a directive which displays the flickr application.

1. Review the Angular directives guide - https://docs.angularjs.org/guide/directive
2. Create a directive which has a controller and a service. The controller calls the service, which should do a JSON request to Flickr and get the a cat feed.
3. It should be possible to add a flickr element to the page, and have it output a list of cat pictures.

```
1    <div flickr></div>
```

Don't worry about parameterising your directive yet, we'll get to that.

# Directive Compilation and the link function

The link function is where we can make direct DOM manipulations, access attributes, etc.

```
1    angular.module('app', [])
2      .directive('usefulDirective', function() {
3        var directive = {
4          link: function(scope, element, attributes) {
5            element.append('Hello Angular!')
6          }
7        }
8        return directive;
9      });
```

The parameters are:

- scope - the current directive scope. Note this will be more complex if you have a transclusion, more on this later.
- element - the element that your directive has been applied to. This is a jqLite element, you can talk to it like jQuery.
- attributes - the element attributes. You can optionally use this to pass values to your directive. You can also pass values using an isolate $scope. More on this later.

## JQLite or templates

Directives give us access to JQuery or JQLite. If jQuery is available, angular will automatically make it available. If not, Angular will use jQLite which is a cut down version of jQuery.

However in most cases you will find that you can get your work done faster and more cleanly using templates, and this is the approach you should generally favour.

## Exercise - a link function

Create a simple directive that uses the link function to append the string "hello from the directive" to your directive using jqLite.

## Exercise - accessing attributes

Create a simple greeting directive. Add an attribute "name" to your element. The directive should look in the attributes array and append 'hello dave' to the DOM, assuming the name was "dave".

# Angular is a compiler

Angular is a real compiler which will traverse your DOM executing directives as it goes. Understanding the order of compilation is crucial to understanding directives.

Angular will treat your DOM as a depth first tree.

If your directive has a controller it will instantiate this on the way down the DOM tree.

If your directive has a link function it will execute this on the way back up the DOM tree, after all the controllers have been instantiated.

```
angular.module('app', [])
  .directive('usefulDirective', function() {
    var directive = {
      controller: function($scope) {
        // this will be instantiated on the way down
      },
      link: function() {
        // this will be instantiated on the way up
      }
    }
    return directive;
  });
```

If you want to execute link functions on the way down the tree declare pre and post link functions like this:

```
angular.module('app', [])
  .directive('usefulDirective', function() {
    var directive = {
      controller: function($scope) {
        // this will be instantiated on the way down
      },
      link: {
        pre: function() {
          // this will be instantiated on the way down
          // but after the controller
        },
        post: function() {
          // this will be instantiated on the way up
        }
      }
    }
    return directive;
  });
```

## Exercise - Parameterise Flickr

We're going to use the link function to access the attr array.

Use Attrs to allow the directive to receive a hardcoded value for a tag.

Call it like this:

```
1    <div flickr tag="london" />
```

## Hard Extension

Use the $parse service to let you parse an expression from $scope.

```
1    <input ng-model='tag' />
2    <div flickr tag="tag" />
```

## Bonus Exercise - Random quote

Create a directive which renders a random quote on the page. Use the link function to replace the content of the current element with the joke.

Bonus, pull the quote from an API, such as the Chuck Norris randome joke API:
http://api.icndb.com/jokes/random

# Isolate Scopes

When we create a directive we often (but not aways) want it to act like it's own little application, separate from its parent. In short, we want it to have its own $scope.

Angular allows us to do this. We can create a directive which has an isolate scope. We can optionally pass parameters to the chald scope, seeding it, and we can also optionally bind values from the document scope, so that when the document scope updates, those parameters in the child scope also update.

## Creating an isolate $scope

Creating an isolate $scope in a directive is simple. Set the $scope to be an empty object, like so:

```
.directive('myDirective', function() {
  return {
    scope: {}
  }
})
```

This will create a little application completely divorced from its parent.

## Passing parameters to an isolate $scope with @

We can parameterise our directive using @. We can tell our directive to take an initial value from an attribute of the directive element, like so:

```
.directive('myDirective', function() {
  return {
    scope: {cheese: '@'},
    template: "<input ng-model='cheese' />{{cheese}}"
  }
})
```

The isolate $scope will be seeded with the cheese attribute from the directive element.

```
<div my-directive cheese="Wensleydale"></div>
```

## Two way isolate $scope binding with =

@ will allow you to pass a variable into your isolate scope.

= will allow you to bind a value in the isolate scope to a value outside of the isolate scope.

This works using simple watchers. Watcher are set up which will watch the values on both $scopes. When one changes, the watcher will update the other and vice versa.

## When to use isolate scopes

Isolate scopes should be used with care. They are not suitable for every case as they break the simple $scope inheritance tree. It's worth noting that none of the built in Angular directives use isolate scopes.

Use an isolate scope when you have created a reusable component that you might want to use in multiple places, for example, a map, or a login form component.

While isolate scopes give us portability they do this at the expense of some flexibility.

## Isolate scopes with transclusion

Isolate scopes will typically be used with a template to work properly. Only template code will gain access to the isolate scope. Transcluded content (content nested in the directive) will still use the parent scope.

## Exercise - Isolate the Flickr app

Give your flickr app an isolate scope. It should be able to receive a tag from it's parent scope.

I want to be able to call it like this:

```
<input ng-model="search" />
<flickr tag="search"></flickr>
```

# Homework

Watch Crockford: JavaScript the good parts.

JavaScript: The Good Parts

# Transclusion

Transclusion allows you to take content that is already in the element and modify it in some way. Say you have a div like this:

```
<div my-directive>
  World of Wonder
</div>
```

Now you apply a directive which includes a template.

```
.directive('myDirective', function() {
  return {
    template: "<div>World of Template</div>"
  }
});
```

The innerHtml is set on the div and "World of Wonder" is no more. Sad. This is where transclusion comes in.

**Transclusion allows us to access the element's original content which has been displaced by the template**

When we tell a directive to transclude, the original content of the element is saved. Why would you want to do this?

## Uses of transclusion

- **ng-if** - the transcluded content is only visible when a condition is met.
- **ng-repeat** - the transcluded content is repeated for each element of an array.
- **wrapping an element** - for example, wrapping a row of buttons in a menu bar.

## Scope of transcluded content

Transcluded content will have the $scope in which it was declared. If your directive has an isolate $scope, the transcluded content will not share it unless you explicitly compile against the isolate scope.

Consider carefully whether you need an isolate scope in this case.

## Two ways to transclude

### 1. the transclude directive

If we tell our directive to transclude, the content that was originally in the element will still be available to us via the transclude directive in our template, like so:

```
myApp.directive('transclusionDirective', function() {
  return {
    transclude:true,
```

```
    template: "Here is the transcluded content: <span ng-transclude></
  }
});
```

### 2. Use the transclude function in the link function

If you have specified transclude: true, The link function will receive a handy transclusion function as it's 5th parameter. Call this function to compile the transclusion. Optionally pass in a scope to compile it with a scope object.

This will allow you to compile the transcluded content against the isolate scope. Beware, this might surprise your user as you will have broken the $scope inheritance tree in your HTML. Do this only occasionally and with proper thought.

```
myApp.directive('transclusionDirective', function($compile) {
  return {
    transclude:true,
    link: function(scope, element, attrs, controller, transclusion) {
      transclusion(scope, function(compiledTransclusion) {
        element.prepend(compiledTransclusion);
      });
    }
  }
});
```

## Exercise - A transcluded header directive

Create a directive that will add a header and footer to any element. You can do this entirely using the a template and the ng-transclude directive.

## Exercise - repeater

Implement a really dumb little directive that simply repeats the content 5 times.

Bonus marks for making it configurable, so that we can repeat the content n times. You would do this by inspecting the attrs array in the pre-link function.

Bonus points for using the $parse service to parse the transcluded template as Angular code.

Use the attr parameter in your link function to receive the value.

Call it like this:

```
<div repeat='5'>
  Hey there!
</div>
```

## Exercise - ng-if

Reimplement ng-if. The transcluded content is shown if the passed in expression evaluates to true. You will need to use $parse to evaluate the passed in expression.

You will not need an isolate $scope here.

Call it like this:

```
<input ng-model="val" type='checkbox' />
<div if='val'>
  Hey there!
</div>
```

# CRUD

The simple API provides an API which allows you to list, view and manipulate Articles, Authors and Comments. It's a fully functional CRUD API built on Rails that get's flushed each night.

- You can review the API spec here
- You can find the source code here

## Exercise - Create a service to access the articles

1. Create a service which can access the article api, doing a get request for the list of articles.
2. Make a controller to call the service and get the articles and add them to $scope.
3. Write a template which will display all of the articles.

## Exercise - Create articles

Extend your service so it can post to the articles API to create a new article. Call it manually from within the controller to test it works.

Create form template and ng-include it on your page. Create a controller to manage form submission. Optionally add a little link or button to show and hide the form. Create a new article object in your controller and bind the form fields to it.

Now in your controller, write a submit function. Cass this function with ng-submit. This function should send the new article to your service and save it.

## Hard Exercise - Write a transcluded directive to add edit links

Transclusion allows us to wrap the content of an element inside a template.

Write a transcluded directive which adds edit links to your articles. When you click it, it should make the content editable in some way by revealing a form. If possible, reuse the article form template you wrote before.

## Bonus Exercise - Comments

If you finish first (and are therefore amazing), consider making a comment service to allow people to comment on an article.

# Form Validation

The form tag is an Angular directive. When you create a form on the page, Angular will create a new scope. If you give your form a name attribute, Angular will add a variable to your new scope with that name which contains validation info.

## Form Exercise

You may wish to extend your earlier CRUD exercise here, or start afresh.

1. Create a form. Give it a name attribute. Turn off browser validation with the novalidate property.
2. Give each of the input elements a name and bind them to scope using ng-model.
3. Now use curly braces to output a value with the same name as the form.

Something like the following:

```
1    <form name="loginForm" novalidate>
2      <input name="email" ng-model="email" />
3      <input name="password" ng-model="password" />
4      <pre>{{loginForm | json}}</pre>
5    </form>
```

Use curly braces to output a scope attribute with the same name as your form and see what Angular has given you.

# Validation

Each input gets an attribute in the loginForm object. There is an $error object for the form and each input gets it's own error object as well.

We use html5 form validation attributes to tell the form how to validate, for example:

- type="email"
- type="number"
- required

You can also use pattern to match against an arbitrary regex:

- pattern="https?://.+"

We also get some custom validation directives from Angular:

- ng-min-length
- ng-max-length

## Exercise - adding validation

Add sensible validation to your form. Use an ng-if to show an error if any fields are invalid.

If you are extending your CRUD exercise, modify your submit method so that it won't submit the form unless the form is valid.

# Styling forms

Each input gets some new classes: ng-dirty is set when the field has been typed in. Don't highlight any error fields until ng-dirty is set. ng-invalid is set when the field is invalid.

## Exercise - Style invalid fields

Use a combination of ng-dirty and ng-invalid to style fields that have invalid values, perhaps with a red border.

Use ng-valid to style fields that have valid values, perhaps with a green border.

## Further Front End Exercise - ng-messages

ngMessages is a new module which standardises messages based on key value pairs. Read about it here:
https://docs.angularjs.org/api/ngMessages/directive/ngMessages.

You can use ngMessages to display form errors based on the $error object.

Do this now. Display errors at the top of the form.

Optionally display messages above each invalid form element.

## Further Hardcore Coder Exercise - Custom validation

We can provide custom form validation with a directive.

The form directive binds a controller to the form. This controller has a method $setValidity. We can get this controller in the link function of a directive as the 4th parameter. Call this method to manually invalidate the form based on the value of an element.

We can get the value of an element using el.val();

Here's a validator directive that always makes the form be invalid.

```
angular.module('app', [])
  .directive('myValidator', function() {
    return {
      require: 'ngModel',
      link: function(scope, el, attrs, controller) {
        scope.$watch(attrs.ngModel, function() {

          controller.$setValidity('always invalid', false);

        });
      }
    }
  });
```

Create a custom validator that checks the password against a set of common passwords:

["password", "passw0rd", "mypassword"]

# Protractor

Protractor is an end to end test framework for Angular. It gives you a JavaScript enabled web browser (Selenium) and a simple JavaScript SDK that lets you drive it. The tests themselves are typically written in Jasmine.

Protractor is the standard test harness used by the Angular core team. You should consider making use of it in your projects.

## Dependencies

Protractor runs as a Node module, so you'll need Node installed. You'll also need the Java SDK.

If you are running Windows you will also need the .Net framework.

## Browser

Tell your browser to hit specific paths and URLs, like this:

```
1    browser.get('http://nicholasjohnson.com/');
```

Your browser will literally navigate to this address.

## Element

Select elements on the page, for example form fields, buttons and links. Call methods to interact with them. For example:

```
1    element(by.id('gobutton')).click();
```

## Installation

Follow the instructions here to install protractor and start up webdriver:

http://angular.github.io/protractor/#/

Now create a config file. Save it as protractor.conf.js in your project directory.

```
1    exports.config = {
2      seleniumAddress: 'http://localhost:4444/wd/hub',
3      specs: ['selenium_specs/*.js']
4    };
```

## What to test

Protractor is for end to end testing user acceptance testing. The internals of your code are a black box. You don't need to check the internals of your app, or how a particular result was

achieved, you just need to know that a particular end goal has been reached.

## Exercise - Create a Test

Now try to create a test suite for your CRUD application. You should be able to drive the browser to create a new article, then navigate to the article and verify it exists.

You can:

1. Tell the browser to navigate to a URL.
2. Get elements and click on them or type in them.
3. Get a single element using element(by.css('.some_classname')) for example and check it's content with getText().
4. Get a list of elements using element.all(by.css('.some_classname')) for example and count it's length.

# $resource

ngResource is a module that provides us with a $resource service. We can use this to make API calls.

We can wrap our resource up in a factory, allowing us to generate a resource that will make an API call on our behalf and instantiate itself.

```
1    $scope.author = $resource('http://simple-api.herokuapp.com/api/v1/auth
```

The $scope.author object is in fact a promise. When the promise is fulfilled, the author will instantiate itself. This gives us this rather nice syntax, even though the author is instantiated asynchronously.

## Using a factory

It's common to create resources using a factory, like so:

```
1    app.factory('Author', function($resource) {
2      return $resource('http://simple-api.herokuapp.com/api/v1/authors/:id
3    });
```

We can now get an author:

```
1    var author = Author.get({ id:1 });
```

There's some magic going on here. The author object is in fact a promise that will configure itself when it is satisfied.

## Making overrides

We can make overrides to our resource routes by passing in a configuration object. This lets us use custom methods and URLs.

```
1    app.factory('Author', function($resource) {
2      return $resource('http://simple-api/api/vi/authors/:id', {}, {
3        update: {
4          method: 'PUT'
5        }
6      });
7    });
```

## Reading

Read the API spec here:

https://docs.angularjs.org/api/ngResource/service/$resource

## Resource Exercise

Extend your CRUD exercise. Create a comment resource and use it to pull comments for articles.

For extra bonus points create a comments directive. You might write:

```
<comments for="12" />
```

to get the comments for article 12.

# Routes

Routes are configured injecting the $routeProvider object into the config function.

Here's a simple example:

```
myApp.config(function($routeProvider) {
  $routeProvider
    .when('/home', {
      templateUrl: 'home.html'
    })
    .when('/about', {
      templateUrl: 'about.html',
      controller: 'aboutController'
    });
});
```

Use the ngView directive to output the content specified by the route:

```
<div ng-view></div>
```

## Entry Exercise - Create a two page AJAX site

1. Read the docs here:
   https://docs.angularjs.org/api/ngRoute/service/$route
2. Create a site with two pages of content, home and contact. Use routing to swap between them.

## Exercise - Flickr sharable URL

Extend the Flickr app so that the tag is in the URL.

When I click search, navigate to a url that contains the tag, like this:

localhost/flickr.html/#cats

This should show a list of cats.

You may need to use the $location service to do this ($location.path('/#cats');)

## Exercise - Simple API

Create a route which displays a single article from the simple API. It should receive an id parameter, then use this to make an api request to the server via your service.

You will need to inject the $routeParams service into your controller to retrieve the parameters in your controller.

From your articles index, link to the individual articles.

https://docs.angularjs.org/api/ngRoute/service/$routeParams

## Further Exercise - New article route

Create an "article_form" template in a separate file. Create a route that will display the form. The form fields should be bound to the scope. When you submit the form, use the values in scope to post to the API, creating a new article and displaying it.

# UI-Router

The default Angular router lets us define URLs and describe how the router should respond, which template it should pull in, and which controller it should use.

UI Router comes from a different angle. Instead of defining URLs, we define named states, then define the URL, controller and templates associated with that state.

We can have multiple named template insertion points for a single state. We can define nested states. We can drop into HTML5 mode and generate real URLs.

## There are 3 ways to change state

**Using the ui-sref directive**

From the template using the ui-sref directive like so:

link

**Using the $state service**

From JavaScript using the $state service:

$state.go('statename');

**By simply changing the URL**

From the browser by navigating to the URL. You can do this by typing in the address bar, or by setting document.location.

## A default route

We can create a default route using otherwise:

```
angular.module('app', ['ui.router'])
  .config(function($stateProvider, $urlRouterProvider) {
    $urlRouterProvider.otherwise("/");
  });
```

Any unmatched route will redirect to '/'

## A state

We define a state like this:

```
angular.module('app', ['ui.router'])
  .config(function($stateProvider, $urlRouterProvider) {

    $urlRouterProvider.otherwise("/");

    $stateProvider
      .state('home', {
        url: "/home",
        templateUrl: "home.html"
```

```
        })
    });
```

We use the ui-sref directive to generate hyperlinks based on the state names. We use the ui-view directive to locate our template.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Scope</title>
    <script src="angular.js"></script>
    <script src="ui-router.js"></script>
    <script src="demo.js"></script>
  </head>
  <body ng-app="app">

    <a ui-sref="home">home</a>
    <div ui-view></div>
  </body>
</html>
```

The state has a name, a URL and a template URL. It can also optionally have a controller.

## Exercise - Add a route to your CRUD

Add a router to your CRUD application that can show a particular article based on the URL.

Optionally add a route to show a list of articles with links to navigate.

comments powered by Disqus