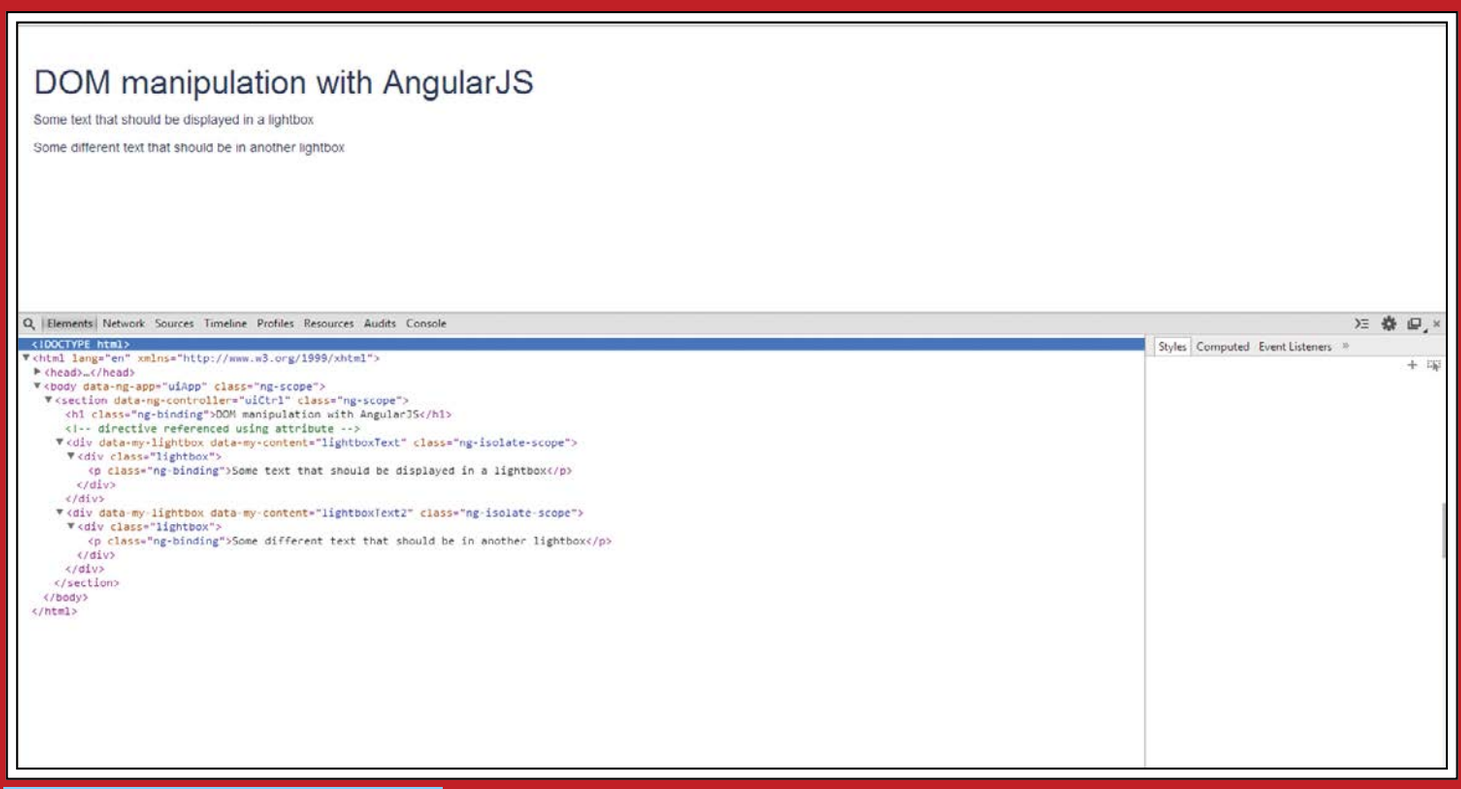
ADVANCED ANGULAR

ANGULAR GENIUS GUIDE

Getting started with AngularJS

Feature: Build smart, complex web applications and quickly develop dynamic user interfaces for your projects

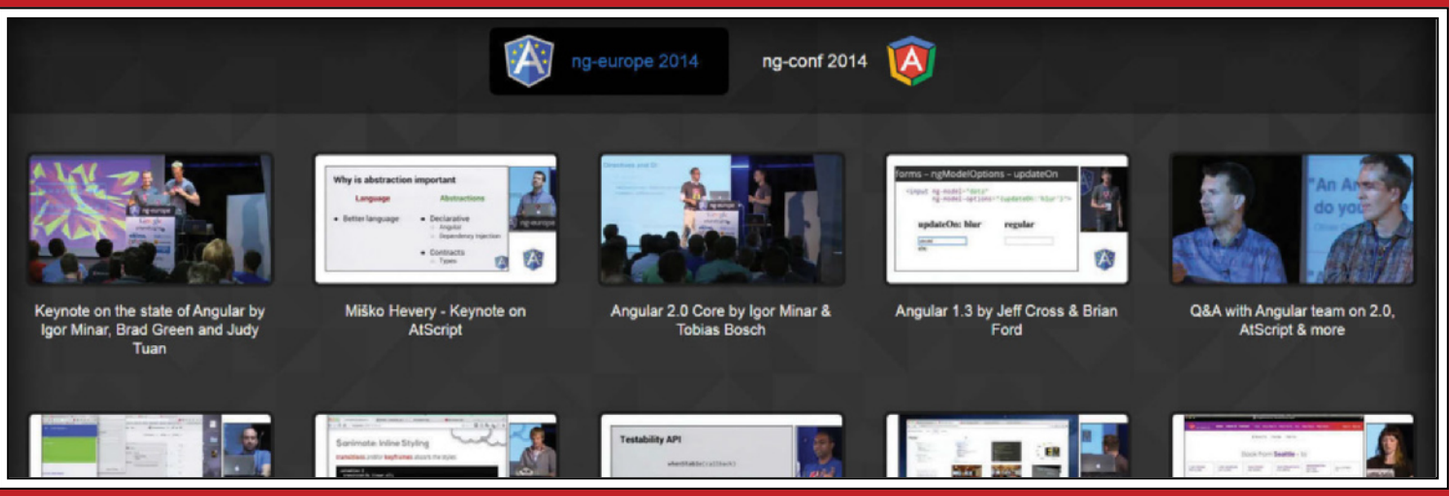


Manipulating the DOM with AngularJS

Tutorial: Create an Angular application, with custom directives to create lightbox and accordion UI components

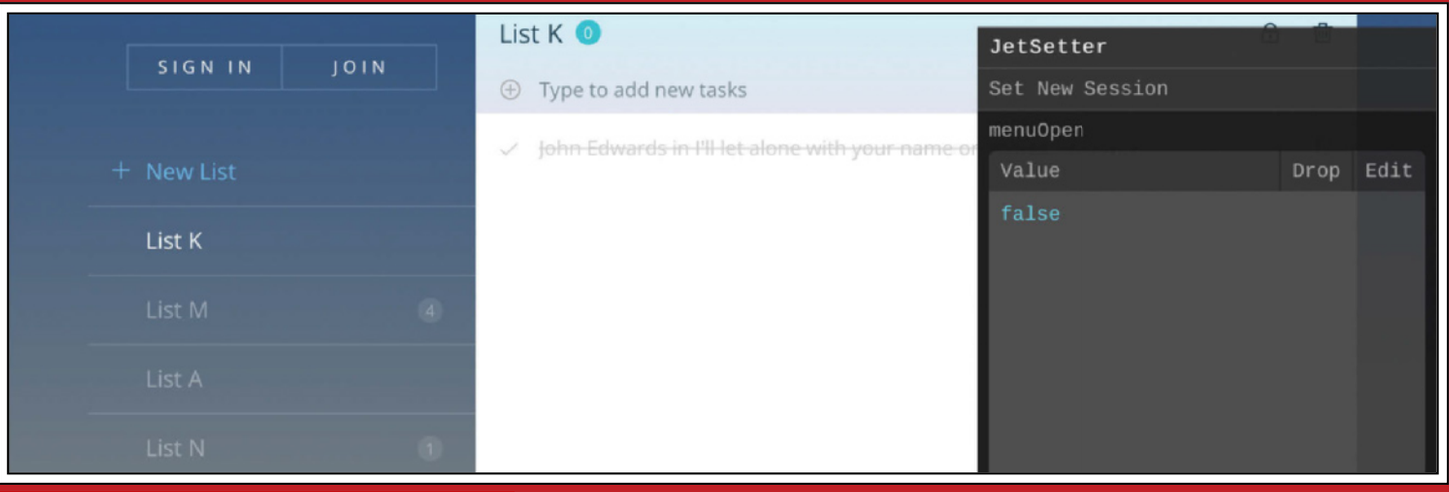
Contents

webdesigner@imagine-publishing.co.uk @WebDesignerMag www.webdesignermag.co.uk



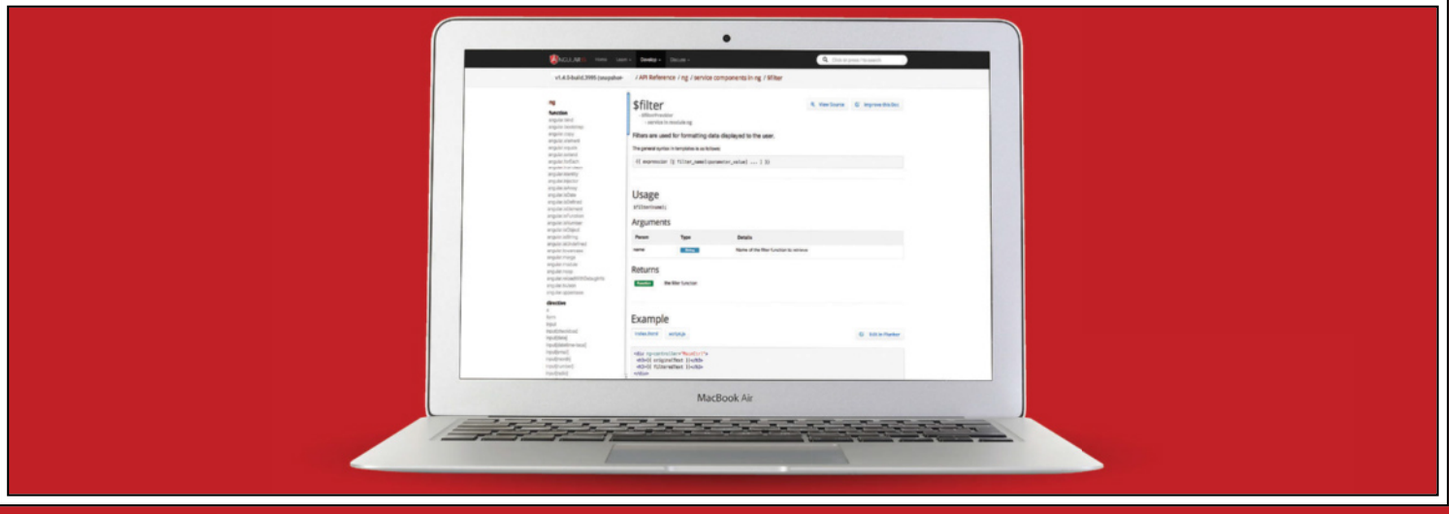
Construct an image gallery with CSS and AngularJS

Tutorial: Combine AngularJS and CSS transitions to create an impressive image gallery UI quickly



Build a reactive web app with Angular-Meteor

Tutorial: Combine the best of both worlds with Meteor and Angular to unleash three-way binding



Advanced Angular

Feature: Get your hands on an unmissable collection of tips and techniques to take your Angular code to the next level

GETTING STARTED WITH ANGULARJS

BUILD SMART, COMPLEX WEB APPLICATIONS AND QUICKLY DEVELOP DYNAMIC USER INTERFACES FOR YOUR PROJECTS

Angular is an application framework that provides a solid MVC (model-view-controller) structure within HTML and JavaScript. This allows you to separate your presentation (view), logic (controller) and data (model) layers of your application; the presentation is held within your HTML, the logic within your JavaScript and the data is held within the scope of your application.

If you are used to working with jQuery, then you will be familiar with selecting HTML elements from the DOM, storing these as variables within your script and then binding methods to them. This is not the most efficient way of binding to HTML as Angular proves.

To begin with, you will need to start thinking a little differently, when working within an MVC design pattern your view should not be directly referenced within your controller, so no more DOM elements stored in variables. Instead your data and methods should be referenced within your HTML from your JavaScript. Once you get going with this approach, Angular will come into its own and you will be building complex applications in no time.

The way Angular achieves this is by leveraging attributes on HTML elements, and these are called directives. Directives can do anything from setting the text within an element to rendering a repeating list of elements, based on an array of data objects, all held within your data layer. A wide range of directives are available out of the box, but what stands Angular's framework apart is that anyone can create their own, and the development community are continually working on expanding these. It's this kind of dedication that will keep Angular useful both now and in the future.

Angular also uses dependency injection to manage functionality available within your controllers. This makes each module of functionality testable in isolation, which is vital in front-end development today.

THE BENEFITS OF USING ANGULAR

BEFORE INVESTING TIME INTO LEARNING A NEW FRAMEWORK, IT IS IMPORTANT TO UNDERSTAND THE BENEFITS AND BE CERTAIN IT FITS WITH YOUR PROJECTS

EFFICIENCY

CODING SPEED AND THE FINAL APPLICATION PERFORMANCE INDICATE AN EFFICIENT DEVELOPMENT APPROACH

Angular provides so much functionality as standard, that many requirements of a web application can be covered with very little bespoke code. The clear separation between your views and controllers in the build process keeps a solid structure that makes it clear which functionality is bound to which HTML elements. When working on your views, the syntax is straightforward as it is simply an extension of HTML5.

The scope of controllers is strictly defined, which ensures that conflicts of variables and methods are eradicated; less coding issues means a faster development process and a better end product. In terms of performance, Angular binds data superfast and is immediately responsive within the user interface. As you explicitly declare data and method bindings on your HTML elements, there is no need to loop through the DOM with JavaScript and store these elements as variables. This is one of the most browser-intensive tasks to undertake and the performance benefits of avoiding this are huge. Angular binds data two ways as default so there's no need to write custom event listeners. With dependency injection, you control the methods available within a controller. In turn your code won't be bloated by unnecessary functions.

MODULARITY

THE MODULAR NATURE AND CLEAR SEPARATION OF PRESENTATION AND LOGIC ARE KEY TO A MVC APPROACH

Angular does not define which code should be separated into modules for you, it simply provides the means for you to define and manage this yourself. There are a range of module types for you to use: a controller being one of them. Each type serves a specific purpose, for example, a filter module receives an array of data, runs any custom queries you want to filter the data by and then returns the filtered data for use in your view. You can create a service module which is excellent for providing common functionality across your application avoiding the need to duplicate code within each of your controllers.

This approach also means that any developer working with the application code can easily see what dependencies a controller may have, making it quick to identify missing components to an application. Without this technique these kinds of problems can take you a longer time to debug within complex applications. It is easy to test separate areas of functionality within your code because you can roll out fixes or enhancements on modules across multiple projects. This helps you in the long run because it makes your builds more stable for use and makes them easier to maintain in the future.

THINK OUTSIDE YOUR CURRENT PROJECT

If you start to write a module or directive, ask yourself whether this would be useful for other projects. If so create it and test it outside before integrating it.

REUSABILITY

WRITE ONCE, TEST AND USE MANY TIMES: AN ADAGE OF A FORWARD-THINKING DEVELOPER

If you find yourself writing the same methods time and again, if you have a wide range of code snippets you use in your projects, you will find that the modular approach of Angular provides a stable way to reuse and manage your code. For example you could create a module to use for any local/cookie storage providing set, get and remove methods or a module with get, post, and put methods for communicating with API web services. Once you have created and tested these modules, they are ready to be reused across applications or even for projects and other developers.

As previously mentioned you can also create your own directives. These are used to generate and manipulate the markup to produce the application you want and, again, are reuseable as they are completely separated from any other application code.

As you work with Angular more and more you will start to create your own collection of reuseable components; if you work with a group of developers this process may be even faster. It won't be long before you find that your development processes involve piecing these components together rather than building them all from scratch. This will make your builds faster, more reliable and allow much more scope within your projects for trying new things.

COMMUNITY

THE PARTICIPATION OF THE DEVELOPMENT COMMUNITY CAN LEAD TO SUCCESS OR DOWNFALL

An active development community is essential not only to make it easier for beginners to get started with a framework or approach, but also to ensure that the framework itself evolves over time and continues to fulfil the needs of the industry. Input into how Angular is being used and extended is not only deriving from Google's Developers but the global community as a whole, which will strengthen Angular's future.

The API reference, tutorials and forums (check them out on **angularjs.org**) hold a wealth of information, not only to get you started but also help you through the most complicated of application builds. The Angular project itself is available at **github.com/angular/angular.js** and has reached 1000 contributors and 5959 commits. Being an open source project means that bugs are found and fixed very quickly, and the rate of development and enhancement is great.

This large community also has a huge range of modules and directives already created for Angular; one of the best sources is **ngmodules.org**. This takes the reuse of testable code and functionality to a global scale and can save a lot of build time within your projects. As you delve into the gamut of modules available, it is important to understand what they are doing for you, so don't be afraid of creating your own and getting involved.

GETTING STARTED

GETTING YOU HEAD AROUND $SCOPE

Each controller within your Angular application has its own scope, referred to within your JavaScript as $scope. Any methods or variables defined within your controller are done so as properties of the $scope object. This means that you can have methods and variables with matching name spaces in different controllers and they will not conflict with each other.

Controllers can also be nested within your view. When you do this, a controller will have access to its parent scope, which is denoted as $parent.

If you wanted to share data across multiple controllers that are not to be nested, you can inject $rootScope: the root scope of your entire application. Any property set on this scope will then be available to any controllers it is injected into. Scopes allow you to call API methods such as $watch, to listen for changes in data or $apply to update any elements bound to data. When Angular compiles your application (on initial load) any required $watch and $apply events are bound between your views and controllers.

SETTING AND UPDATING DATA

Data can be set and updated from either your controller or view; as mentioned any data set within your JavaScript is done so within the $scope object. Within your views, this is slightly different. There is no need to reference $scope as your view is bound to a controller explicitly. You can use data-ng-model to set/get data on elements such as text inputs. When editing the text input in the browser, the scope data will immediately be updated.

You can set simple text values to elements, such as <p> tags, using the data-ng-bind directive, or alternatively inline using a handlebars format.

If a data property is not found within the scope of your controller, Angular will automatically traverse up through any parent controllers or the root of the application to find a match. This means that you can reference a $rootScope variable within your view without explicitly referencing $rootScope.

CROSS-COMMUNICATION

YOU CAN COMMUNICATE ACROSS YOUR APPLICATION TO RESPOND TO SPECIFICALLY DEFINED EVENTS WHEN YOU CHOOSE

Listening for events fired within a browser is nothing new in JavaScript development. Angular leverages this standard approach, by providing methods to communicate with custom events that you can define and fire when you wish, within any part of your application.

By using a combination of $broadcast, $emit and $on methods throughout your application this communication can be achieved. $broadcast dispatches an event downwards to any child scopes, whereas $emit dispatches upwards to any parent scopes. Each of these methods receive a name (as a string) and args (as an object). The $on method receives a name (as a string) and a listener (as a function). When the event with the matching name is identified by the $on method, the listener function is run and passes any args from the $broadcast/$emit method. This is a powerful combination of Angular API methods that really opens up the possibilities of your application development.

THE CORE ANGULAR DIRECTIVES

WITH THESE CORE DIRECTIVES YOU CAN QUICKLY GET STARTED BUILDING YOUR VIEWS WITH ANGULAR

NG-APP Designates the root element for your application with the namespace set in your root module. This is usually used on the <html> or <body> tag of your page.

NG-CONTROLLER Attaches the controller to view and links a controller within your application to a section of your HTML, using the namespace for the controller. This applies the controller scope to the HTML element and its contents.

NG-BIND Sets the text content of a HTML element from the value provided or the result of a given expression.

NG-MODEL Binds form elements (eg input, select, textarea) to a scope value. If this scope value is set within the controller the form element will display as such. When the form element is updated, the scope value will update automatically.

NG-CLASS Sets a class attribute on a HTML element based on a given expression, databinding it and then representing all classes to be added. This expression can be as simple or as complex as you require, it just has to return a string to be set as the resultant class.

NG-IF Works very similarly to ng-show, but only renders the HTML element if the given expression returns true, whereas ng-show will still render the HTML element within the page source.

NG-CLICK Binds a method to the click event on a HTML element. There are a range of similar directives available such as ng-dblclick, ng-mouseover, ng-keydown and many others.

NG-REPEAT Very powerful directive. It repeats the HTML element and its contents based on a given array of data items, each repeated HTML block has the scope of its data item.

USE VALID HTML BINDINGS

Prefix your directive references within HTML with data-ng opposed to just ng- to ensure you are using valid HTML5 syntax throughout your project.

BUILD AN ANGULAR APP

IN THIS GUIDE WE WILL CREATE AN ANGULAR APPLICATION TO RANDOMLY GENERATE TEAMS FROM A LIST OF USER-DEFINED NAMES

SET NG-APP AND NG-CONTROLLER

Start by creating a JavaScript file and referencing it in your HTML along with AngularJS. In your script file create an Angular module to hold your application with a controller as shown. Then set data-ng-app and data-ng-controller within your HTML.

```
001 // teamApp.html
002 <html lang="en" xmlns="http://www.
w3.org/1999/xhtml">
003 <head>
004   <meta charset="utf-8" />
005   <title>Angular Team App</title>
006   <link href="css/normalize.css"
rel="stylesheet" media="screen" />
007   <link href="css/main.css"
rel="stylesheet" media="screen" />
008   <script src="scripts/angular.min.js"></
script>
009   <script src="scripts/teamAppModule.
js"></script>
010 </head>
011 <body data-ng-app="myTeamApp">
012     <section data-ng-
controller="teamCtrl">
013     </section>
014 </body>
015 </html>
016 // teamAppModule.JS
017 var teamApp = angular.
018 // teamAppModule.JS
019 var teamApp = angular.module('myTeamApp',
[])
020 .controller('teamCtrl', ['$scope', function
($scope) {
021 }]);
```

BIND A TEXT VALUE

Now add a $scope property to display some text within our HTML page. This is added within the controller by extending the $scope object. This can then be bound within your HTML using the data-ng-bind directive.

```
001 // teamApp.html
002 <body data-ng-app="myTeamApp">
003   <section data-ng-controller="teamCtrl">
004   <h1 data-ng-bind="pageTitle"></h1>
005   </section>
006 </body>
007 // teamAppModule.js
008 .controller('teamCtrl', ['$scope', function
($scope)
009   $scope.pageTitle = "Team Generator";
010 }]);
```

CREATE SOME DATA

Add a text input to your HTML and use data-ng-model to bind it to your $scope. Add a button bound to an add() method. Now define your empty array and add/remove methods within your controller. If you log out $scope within the add() method you can see the people array building up.

```
001 $scope.people = [];
002 $scope.add = function(name){
003     $scope.people.push(name);
004     $scope.newName = "";
005     console.log($scope);
006 };
007 $scope.remove = function(index){
008   $scope.people.splice(index, 1);
009 };
```

DISPLAY THE DATA

Now use data-ng-repeat to display the people array within your HTML and include a delete button for each of them. You can pass $index into the remove method from within an Angular repeater.

```
001 <ul>
002   <li data-ng-repeat="person in people">
003     <h2>{{person}}</h2>
004     <button data-ng-
click="remove($index)">delete</button>
005   </li>
006 </ul>
```

CREATE THE TEAMS

Add three more methods into your controller. These will shuffle an array, chunk an array and call these and generate the teams. With the first method, use angular.copy ($scope.people) to avoid shuffling the array itself.

BIND THESE METHODS

Using the same technique as adding people, create an input field and button to generate the teams by calling the $scope.generateTeams() method and passing it the number defined. You can log out $scope.teams to view the generated teams within the console.

```
001 <div class="teams">
002   <form>
003     <label>No of teams:</label>
004     <input type="number" data-ng-
model="teamNumber" />
005     button data-ng-click="generateTeams(te
amNumber)">Create teams</button>
006   </form>
007 </div>
```

DISPLAY THE TEAMS

Now use the same data-ng-repeat directive as before, but this time we will nest repeaters. You can use the $index value of the repeater to label the teams accordingly.

```
001 <div data-ng-repeat="team in teams">
002   <h3>Team {{$index + 1}}</h3>
003   <ul>
004   <li data-ng-repeat="person in
team">{{person}}</li>
005     </ul>
006   </div>
```

RESET FUNCTIONALITY

The final method needed is to reset the teams, this can be added to your controller, simply empty the $scope.teams array and in doing so the view will be immediately updated. This method should also be called when generating the teams.

```
001 $scope.generateTeams = function(number){
002   $scope.resetTeams();
003   var shuffled = $scope.shuffle(angular.
copy($scope.people));
004   $scope.teams = $scope.chunk(shuffled,
number);
005   console.log($scope.teams);  };
006 $scope.resetTeams = function(){
007   $scope.teams = []; };
```

ADD A RESET BUTTON

Finally add a reset button to your HTML and bind $scope.resetTeams() method using data-ng-click. You can set this button only to display when $scopeTeams array has items within it using data-ng-show

```
001 <button data-ng-show="teams.length > 0"
data-ng-click="resetTeams()">Reset teams</
button>
```

HOW TO INTEGRATE

SO YOU GET HOW ANGULAR WORKS BUT HOW DOES IT FIT WITHIN A LARGE WEB PROJECT?

Most website builds aren't full JavaScript-driven applications: they may have a small functional component that uses JavaScript or a range of different components across a large CMS driven platform. Your Angular controllers can be integrated very easily by setting one application namespace on your <body> or <html> tag across your site. Then you can reference any controllers within your HTML.

It is best practice to separate your Angular modules into individual files (this makes reuse and maintenance simple), and then you can assign collections of script files to HTML pages or blocks of HTML as a result of doing this. Use a file compilation tool to combine these into a single HTTP request within your live site. Many CMS platforms have these built in and there is a wide range available for both PHP and .NET.

One thing to look out for is any JavaScript functionality that does not sit within your Angular controllers; if any data from functional components is needed within your application make sure it is defined within it. If this is not done it will not be possible to leverage Angular's two-way binding of data and events effectively and can become difficult to unit test and maintain later on.

Learn more about two-way binding on **angularjs.org**

THE API REFERENCE

THE QUALITY OF A FRAMEWORK IS OFTEN JUDGED PARTLY ON ITS DOCUMENTATION AND IN ANGULAR'S CASE, THIS IS EXCELLENT

The API reference covers core directives, functions, services, providers and filters as well as any similar components included within available modules. Each item covered has a clear definition and context with example code usage and reference tables.

One of the best things about the documentation is that any related items are cross-referenced, making it really easy to solve quite complex problems you may come across. In addition there is a big selection of tutorials, developer guides and presentation videos available online to help you get fully immersed in coding the Angular way. There is continual progression across the Angular community generally in the release of more and more modules/directives; many integrate with common APIs and online services, such as the Google Map directives available at **angular-google-maps.org**. This API reference is a great place to continue learning how you can use Angular. It is worth familiarising yourself with all the core components available and keeping this site bookmarked as you experiment further.

The directives for Google Maps include markers and windows to name a few

BEST PRACTICES

HERE ARE A FEW OF THE BEST PRACTICES, TIPS AND TUTORIALS AVAILABLE

NAME YOUR DIRECTIVES CORRECTLY

Prefix your custom directive namespaces with something other than ng. That way, your directives will never conflict or get confused with standard Angular directives or conflict with possible developments in future HTML spec.

WRITE SIMPLE CONTROLLERS

Keep the amount of logic within your controllers simple and clearly separated from each other. This ensures ease of maintenance and reusability as well as helping you to create a well-structured and testable application.

ARRANGE CODE INTO SEPARATE FILES

Keep your modules in separate files that can be bundled together accordingly. It is a good approach to have a separate folder to hold your Angular files, separate from any other JavaScript files. It is also useful to have separate folders for things like controllers, directives and filters.

USE DIRECTIVES EFFECTIVELY

If you find yourself trying to store or access a HTML element within a controller, stop and consider using a directive: you probably need one. Also avoid using directives as a HTML tag, always use them as attributes as this is valid HTML5.

USE NG-CLOAK

This stops HTML elements from rendering prior to compiling your application, and it means you can avoid elements being displayed in their raw form during page load for a smoother user experience.

KEEP UP TO DATE

Use Twitter, read blogs and regularly check on developments with Angular. This will make sure you are getting the most out of Angular and keep abreast of the possibilities.

Manipulating the DOM with AngularJS

Create an Angular application, with custom directives to create lightbox and accordion UI components

tools | tech | trends AngularJS, JavaScript, HTML, CSS
expert Luke Guppy

AngularJS is an application framework that clearly separates the DOM from any logic, by means of a view (your HTML) and a range of controllers (your JavaScript logic). This is all held together using directives, and AngularJS has a huge range of directives available. These are used to join the DOM elements with any data and logic held within your controllers and are added to the markup as attributes. With jQuery for example, the DOM elements are referenced from the JavaScript and not the other way around. This means that to get these DOM references, the entire (or a significant portion) of the markup must be looped through for each element. As well as utilising the standard directives, custom directives can be built. This tutorial will cover the creation of such directives to provide both lightbox and accordion UI components that can be easily reused. Following this you will be able to create your own custom directives for any custom UI interactions you may need as part of your AngularJS application.

01 Set up the HTML

Start by adding a reference to AngularJS from the Google CDN to your HTML page. Then add a reference to your application on the body tag using an attribute as shown. If you use the 'data-ng-' prefix for your Angular attributes as opposed to 'ng-' your HTML will be valid.

```
<!DOCTYPE html>
<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
  <meta charset="utf-8" />
  <title>Angular Listing App</title>
  <script src="http://ajax.googleapis.com/ajax/libs/
  angularjs/1.2.15/angular.min.js"></script>
</head>
<body data-ng-app="uiApp">
</body>
</html>
```

02 Create the scripts

In your scripts folder create a folder for modules. Then create a file named 'myApp.js' in the scripts folder and a file named 'ui.js' in the modules folder, reference these in your HTML page. Add the following code to these files; this defines your core application and uses dependency injection to add your module to this application.

```
//modules/ui.js
angular.module(ui.module',[])
//myApp.js
var myApp = angular.module('uiApp', ['ui.module']);
```

03 Add a controller

Define a controller in ui.js and pass in $scope. All data and methods defined in this controller will extend $scope. Add a reference to your new controller in your HTML page and you can then display any data defined in $scope. In a browser window you should now see the $scope.title value displayed in a H1 tag.

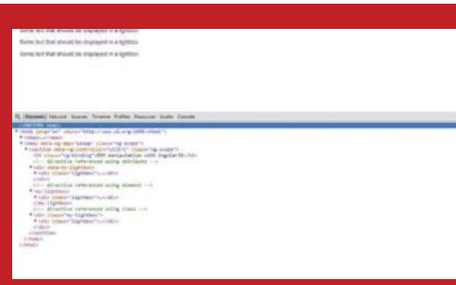
```
<!DOCTYPE html>
<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
  <meta charset="utf-8" />
  <title>Angular UI App</title>
  <script src="http://ajax.googleapis.com/ajax/libs/
  angularjs/1.2.15/angular.min.js"></script>
  <script src="scripts/modules/ui.js"></script>
  <script src="scripts/myApp.js"></script>
</head>
<body data-ng-app="uiApp">
  <section data-ng-controller="uiCtrl">
    <h1>{{title}}</h1>
  </section>
</body>
</html>
//modules/ui.js
angular.module('ui.module', [])
.controller('uiCtrl', ['$scope', function ($scope) {
  "use strict";
  $scope.title = "DOM manipulation with AngularJS";
  $scope.lightboxText = "Some text that should be
  displayed in a lightbox";
}]);
```

04 Create your first directive

Create a directives folder and a file named 'uiDirectives.js'. In this newly created file you can create a directive module named 'myLightbox' as shown. Then add this directive module to your application as shown and add a reference to this new script file within your HTML page, making sure myApp.js is the last script file referenced.

<Above>
• All file downloads are available from **https://angularjs.org** or on GitHub along with extensive documentation and tutorials on basic usage.

<Above>
• Properties defined within your controller $scope are available to render within your HTML using a straight forward 'handlebars' syntax.



<Above>
• The lightbox directive renders the markup defined by its template property in all instances, and this markup has access to the controller $scope.

```
angular.module('ui.directives', [])
.directive('myLightbox', function () {
  "use strict"
  return {
    restrict: "AEC",
    template: "<div class='lightbox'>
    <p>{{lightboxText}}</p></div>"
  }
};
//myApp.js
var myApp = angular.module('uiApp', ['ui.
directives','ui.module']);
```

05 Use your directive

Now you can add your directive to the HTML. This can be done by attribute (A), element (E), or class (C). This can be restricted within the directive itself by changing the 'restrict' property. After this step we will use attributes for the rest of this tutorial.

```
<section data-ng-controller="uiCtrl">
  <h1>{{title}}</h1>
  <!-- directive referenced using attribute -->
  <div data-my-lightbox></div>
  <!-- directive referenced using element -->
  <my-lightbox></my-lightbox>
  <!-- directive referenced using class -->
  <div class="my-lightbox"></div>
</section>
```

06 Isolating directive scope

This directive will only support one instance within the same controller as the same content will be displayed each time. To fix this we need to isolate the scope of the directive. Attributes can be passed to a directive and added to the scope property. We will use 'my-content' as an attribute.

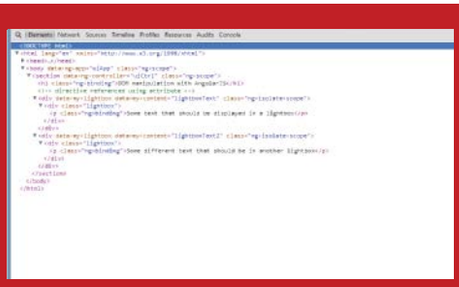
```
angular.module('ui.directives', [])
.directive('myLightbox', function () {
  "use strict"
  return {
    restrict: 'AEC',
    scope: {
      contentText: "=myContent"
    },
    template: "<div class='lightbox'>
    <p>{{contentText}}</p></div>"
  }
});
//modules/ui.js
$scope.lightboxText = "Some text that should be
displayed in a lightbox";
$scope.lightboxText2 = "Some different text that
should be in another lightbox";
```

```
//myApp.html
<div data-my-lightbox data-my-content="lightboxText">
</div>
<div data-my-lightbox data-my-content="lightboxText2">
</div>
```

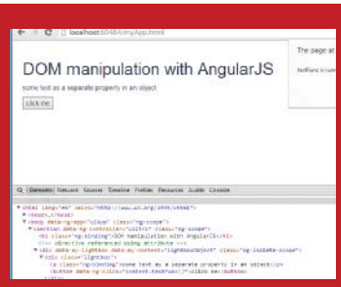
07 Passing functions

You are not limited to just passing strings into a directive; you can pass any object in from the controller $scope. Create a new object in your controller with a text property and function that fires an alert. Now adjust the directive template property with a data-ng-click binding and you will be able to run this function from your lightbox.

```
<section data-ng-controller="uiCtrl">
  <h1>{{title}}</h1>
  <!-- directive referenced using attribute -->
  <div data-my-lightbox data-my-content=
  "lightboxObject"></div>
</section>
//modules/ui.js
$scope.lightboxObject = {
  text: "some text as a separate property in an
  object",
  testFunc: function(){
    alert('testFunc is running');
  }
}
//directives.uiDirectives.js
angular.module('ui.directives', [])
.directive('myLightbox', function () {
  "use strict"
  return {
    restrict: 'AEC',
```



<Above>
• By isolating the directive scope you can render different content across multiple instances of a directive



<Above>
• Functions defined within the controller $scope can be run from within a directive

<Above>
• Combining the data-ng-show attribute and a Boolean property in the controller makes it really easy to hide and show the lightbox

```
    scope: {
      content: “=myContent”
    },
    template: “<div class=’lightbox’><p>{{content.
    text}}</p><button data-ng-click=’content.testFunc
    ()’>click me</button></div>”
  }
});
```

08 Toggling the lightbox

To hide and show the lightbox we need a Boolean property on the $scope.lightboxObject that defaults to false. Then change $scope.lightboxObject.testFunc to a toggleView function that switches this property.

```
$scope.lightboxObject = {
  text: “some text as a separate property in an
  object”,
  visible: false,
  toggleView: function () {
    this.visible = !this.visible;
  }
}
```

09 Add toggle controls

Within the directive template property we need to add a close button that calls our toggleView function and a data-ng-show attribute to set the display state of the lightbox. Then add a button within your HTML that calls the toggleView function. Use the data-ng-click attribute to bind the functions.

```
//directives/uiDirectives.js
template: “<div class=’background’ data-ng-show=
  ’content.visible’></div><div class=’lightbox’ data-
  ng-show=’content.visible’>><button data-ng-click=
  ’content.toggleView()’>close</button><p>{{content.
  text}}</p></div>”
//myApp.html
<section data-ng-controller=”uiCtrl”>
  <h1>{{title}}</h1>
  <!-- directive referenced using attribute -->
  <div data-my-lightbox data-my-content=
  ”lightboxObject”></div>
  <button data-ng-click=”lightboxObject.toggleView
  ()”>Show lightbox</button>
</section>
```

Directive naming

It is best practice to always prefix your directive names. This will avoid any conflicts with future changes in HTML elements. Do not us ‘ng’ as a prefix as this would then confuse custom directives with the standard set provided as part of AngularJS. Also note that when referenced in HTML the directive names are hyphenated opposed to being camel case.

10 The transclude property

The problem now is we would have the same HTML structure for the content of each lightbox. This is where the transclude property comes in. It passes the content of the directive reference into the directive itself including any bindings to the controller $scope. Set the transclude property to true within your directive and then add the attribute data-ng-tansclude in the directive’s template property.

```
angular.module(‘ui.directives’, [])
.directive(‘myLightbox’, function () {
  “use strict”
  return {
    restrict: ‘AEC’,
    transclude: true,
    scope: {
      content: “=myContent”
    },
    template: “<div class=’background’ data-ng-show=
    ’content.visible’></div><div class=’lightbox’ data
    -ng-show=’content.visible’><button data-ng-click=
    ’content.toggleView()’>close</button><div data-ng-
    transclude></div></div>”
  }
});
```

11 Add your lightbox content

You can now put any HTML content within your lightbox in myApp.html. This content has direct access to everything within controller $scope so you can reference your lightbox content from $scope.lightboxObject.text as well as any other functions or values set within the controller. Remember this HTML doesn’t have access to the directive scope so {{content.text}} will not work.

```
<div data-my-lightbox data-my-content=”lightboxObject”>
  <h2>The title of my lightbox</h2>
  <p>{{lightboxObject.text}}</p>
</div>
```

12 Simplify the HTML

We can simplify our HTML by passing $scope.lightboxObject to the directive attribute itself. Any attributes set on a HTML element are available within a directive but this will keep our markup a bit tidier.

```
//myApp.html
<div data-my-lightbox=”lightboxObject”>
//directives/uiDirectives.js
scope: {
  content: “=myLightbox”
}
```

13 Separate the directive template

You may find it gets difficult to read and manage the template property as a string if it gets larger. Alternatively a templateUrl property can be set that references a path to a HTML file that holds this structure. Separate the lightbox template HTML into a file and reference it from a templates folder as shown.

```
//directives/templates/lightbox.html

<div class=’background’ data-ng-show=’content.visible’></div>
<div class=’lightbox’ data-ng-show=’content.visible’>
  <button data-ng-click=’content.toggleView()’>close</button>
  <div data-ng-transclude></div>
</div>
//directives/uiDirectives.js
templateUrl: “scripts/directives/templates/lightbox.html”
```

14 Add an accordion directive

Now we have our lightbox directive all working we can start the accordion component. This will require two nested directives: myAccordion and myItem. Firstly create the myAccordion directive with transclude set to true and a very simple template.

```
.directive(‘myAccordion’, function () {
  “use strict”
  return {
    restrict: ‘AEC’,
    transclude: true,
    template: “<div class=’accordion’ data-ng-transclude></div>”
  }
})
```

15 Add a controller

This directive will need some functionality defined within it so we need to create a controller inside the directive itself. Inside this controller create an empty array called items and an addItem function to push items to this array. We will use these shortly.

```
controller: function($scope, $element){
  this.items = [];
  this.addItem = function (item) {
    this.items.push(item);
  };
}
```

16 Create the item directive

Now create the myItem directive with the require property set to “^myAccordion”. This denotes that this directive can only be used as a child of myAccordion. We will need some things to be set up when this directive is called so add a function to the link property as shown.

```
.directive(‘myItem’, function () {
  “use strict”
  return {
    restrict: ‘AEC’,
    require: “^myAccordion”,
    transclude: true,
    link: function(scope, element, attrs, parentCtrl){
      // code here will run when the directive is called
    },
    template: “<div class=’accordionContent’ data-ng-transclude></div>”
  }
})
```

17 Add your accordion HTML

Now add some accordion items to your HTML using these two directives. Once this is done you should see the HTML structure changed to match the directives templates when you inspect in a browser. You can define any desired content within the accordion items.

```
<div data-my-accordion>
  <div data-my-item>
    <p>Some intro text to this accordion item</p>
    <ul>
      <li>Bullet item 1</li>
      <li>Bullet item 2</li>
      <li>Another bullet item</li>
    </ul>
  </div>
  <div data-my-item>
    <h3>A sub heading</h3>
    <p>Some text for the second accordion</p>
    <ol>
      <li>Numbered item 1</li>
      <li>Numbered item 2</li>
      <li>Another numbered item</li>
    </ol>
  </div>
  <div data-my-item>
    <p>Some intro text to the last accordion item</p>
    <ul>
      <li>Bullet item 1</li>
      <li>Bullet item 2</li>
      <li>Another bullet item</li>
    </ul>
  </div>
</div>
```

18 Add titles

Now add a title attribute to each myItem in the HTML and create a title for each item within the myItem directive. The title attribute can be added to the directive scope with an @ symbol if using the same namespace as the attribute.

```
<div data-my-item title=”Accordion item 1”>
//directives/uiDrectives.js
.directive(‘myItem’, function () {
  “use strict”
  return {
    restrict: ‘AEC’,
    requires: “^myAccordion”,
    transclude: true,
    scope: {
      title: “@”
    },
    link: function(scope, element, attrs, parentCtrl){
      // code here will run when the directive is called
    },
    template: “<h2>{{title}}</h2><div class=’accordionContent’ data-ng-transclude></div>”
  }
});
```

19 Extend the accordion items

To set and control the open state of an item we need to extend the scope of the item within the link function. Add scope.open as a Boolean defaulted to false and scope.toggle as a function toggling this state. Then use data-ng-show and data-ng-click within the template.

```
link: function(scope, element, attrs, parentCtrl){
  scope.open = false;
  scope.toggle = function () {
    if (scope.open) {
      scope.open = false;
    }
    else {
      scope.open = true;
    }
  };
},
template: “<h2 data-ng-click=’toggle()’>{{title}}</h2><div class=’accordionContent’ data-ng-transclude data-ng-show=’open’></div>”
```

20 Automatically close items

You can now open and close accordion items but only one item should open at a time. To fix this you need to give the myAccordion directive a reference to each item. You can call parentCtrl.addItem() from the myItem directive and push your item scope to this array.

```
parentCtrl.addItem(scope);
```

21 Add a closeMe function

The myAccordion directive needs to be able to close all items. So now add a closeMe function to each item keeping the close functionality within the item scope. Your myItem directive should now be ready. See the full code on the disc for more information.

Code library

The power of directives

As they give you access to both controller scope and HTML elements, directives are extremely powerful and can be leveraged to produce some great results.

```
var myApp = angular.module(‘uiApp’, [‘ui.directives’,’ui.module’]);
```

As well as your main ui.module any directives your application needs must be injected into your application. This file should only define the application structure.

```
angular.module(‘ui.directives’, [])
.directive(‘myLightbox’, function () {
  “use strict”
  return {
    restrict: ‘AEC’,
    transclude: true,
    scope: {
      content: “=myLightbox”
    },
    templateUrl: “scripts/directives/templates/lightbox.html”
  }
})
.directive(‘myAccordion’, function () {
  “use strict”
  return {
    restrict: ‘AEC’,
    transclude: true,
    controller: function($scope, $element){
      this.items = [];
      this.addItem = function (item) {
        this.items.push(item);
      };
      this.closeAll = function () {
        angular.forEach(this.items, function (item) {
          item.closeMe();
        })
      }
    },
    template: “<div class=’accordion’ data-ng-transclude></div>”
  }
})
.directive(‘myItem’, function () {
  “use strict”
  return {
    restrict: ‘AEC’,
    require: “^myAccordion”,
    transclude: true,
```

Your final uiDirectives file contains all three directives we have created. Any custom directives should be separated into separate modules/files to make reuse easy across projects.

Construct an image gallery with CSS and AngularJS

Combine AngularJS and CSS transitions to create an impressive image gallery user interface quickly

tools | tech | trends AngularJS, JavaScript, HTML, JSON
expert Luke Guppy

Gallery UIs are often required within clients' website projects, and having a reuseable, modular solution can save valuable time in your builds. There are many jQuery gallery plugins available, but these are often quite large. These plugins rely on jQuery's DOM first approach to UI interactions, requiring your content data to be held and referenced from the DOM. AngularJS separates your data and presentation so the UI can load and run superfast and any functionality is bound directly from your DOM.

This tutorial will cover the basic application setup, creation of a directive with its own HTML template, and the setting of data for that directive within a controller. Angular directives do more than just manipulate the DOM, they can also receive arguments through HTML attributes, enabling config option adjustment for your resultant UI. In addition the CSS3 transition property will be used for efficient transitions. After completing this tutorial you will be able to take this structure and approach to build up a range of reuseable directives for your common interactive components.

01 Create the main HTML page

Start by adding a reference to AngularJS from the Google CDN to your HTML page. Then add a reference to your application on the body tag using an attribute as shown. If you use the data-ng- prefix for your Angular attributes opposed to 'ng-' your HTML will be valid.

```
001    <!DOCTYPE html>
002    <html lang="en" xmlns="http://www.w3.org/1999/xhtml">
003    <head>
004    <meta charset="utf-8" />
005    <title>Angular Gallery App</title>
006    <link href="css/styles.css" rel="stylesheet" media="screen"
/>
007    <script src="http://ajax.googleapis.com/ajax/libs/
angularjs/1.3.10/angular.min.js"></script>
008    </head>
009    <body data-ng-app="galleryApp">
010    </body>
011    </html>
```

02 Create the core application

In your scripts folder create a file named 'myGallery.js', this file will define your core application and any dependencies will be added to your application here. Ensure the namespace defined here matches the namespace set within the 'data-ng-app' attribute in the HTML. Reference this file within your HTML.

```
001    <head>
002    <meta charset="utf-8" />
003    <title>Angular Gallery App</title>
004    <script src="http://ajax.googleapis.com/ajax/libs/
angularjs/1.3.10/angular.min.js"></script>
005    <script src="scripts/myGallery.js"></script>
006    </head>
```

```
007    //scripts/myGallery.js
008    var myGallery = angular.module('galleryApp', []);
```

03 Add your CSS

Add references to your CSS files in your main HTML page. Within the tutorial files there are both 'normalize.css' and 'styles.css', feel free to work with these or your own presets.

```
001      <link href="css/normalize.css" rel="stylesheet"
media="screen" />
002      <link href='http://fonts.googleapis.com/
css?family=Poiret+One' rel='stylesheet' type='text/css'>
003 <link href="css/styles.css" rel="stylesheet" media="screen" />
```

04 Add a directive

Now add a directives folder in your scripts folder and create a file named 'galleryDirective.js'. In this file create a new angular.module and add a 'myGallery' directive as shown. A directive returns a range of properties, for now set restrict to 'A' (it makes the directive available as an attribute) and templateUrl as shown.

```
001    angular.module('gallery.directive', [])
002    .directive('myGallery', function () {
003    "use strict"
004    return {
005    restrict: 'A',
006    templateUrl: "scripts/directives/templates/gallery.html"
007     }
008    });
```

05 Create your directive template

Within the directives folder add a folder named 'templates' with a file named 'gallery.html' and put in a piece of sample HTML in this file. Now add a reference to your new directive within your main HTML page and add the

<Above>
• You should see your Gallery Title displayed through your directive. Any text value can be rendered using simple handlebars syntax

<Above>
• Angular repeaters are very useful for many UI scenarios, the context of each object within an array is the scope within a repeater

<Above>
•Arrays held within $scope can be handled just as in any JavaScript syntax. In this case we select the first item using [0]

<Above>
•Box-shadow can be a really useful CSS property. It can add great depth to your designs and, if combined with pseudo elements, can produce some interesting effects

Why use CSS animations?

If we were building this gallery with jQuery we could use the fadeIn and fadeOut methods to animate the main image appearing. CSS transitions perform much better in browsers and are easy to adjust.

directive as a dependency in your core app file as shown. Your directive template should be rendered within your HTML page in the browser.

```
001    <body data-ng-app="galleryApp">
002    <div data-my-gallery></div>
003    </body>
004    //scripts/myGallery.js
005    var myGallery = angular.module('galleryApp', ['gallery.
directive']);
006    //scripts/directives/templates/gallery.html
007    <h2>My New Gallery</h2>
```

06 Create a controller

To add the data needed for the gallery we will use a controller to hold it within a $scope object. Within your scripts folder add a folder named 'modules' with a file named 'galleryModule.js'. Within this file create a controller and set your title text within the $scope object.

```
001 angular.module('gallery.module', [])
002 .controller('galleryCtrl', ['$scope', function($scope) {
003 "use strict";
004 $scope.galleryTitle = "My New Gallery";
005 });
```

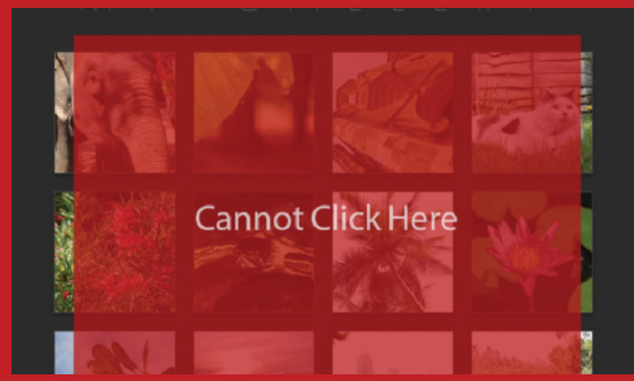
07 Use your new controller

Add your controller as a dependency to your core app file, then reference your controller within your main HTML page. Extend your 'galleryDirective.js' file with the 'scope' property. This will isolate the scope of the directive and enable you to reference your 'data-my-title' attribute.

08 Add a controller to your directive

Now the scope of the directive is isolated add a controller. This controller will hold any functionality directly tied to the gallery interface. The controller accepts arguments for the current scope, the HTML element the

<Above>
• The transition property is great to add efficient animation transitions to your UI. It is also really quick to experiment with



<Above>
• When using opacity to hide HTML elements the browser will still behave as if they are there. This can block click events on lower elements

<Above>
•Incorporating z-index into the CSS transition gives you more control over the hidden element. Chaining transitions can be really useful

<Above>
• Try visiting **css3gen.com/box-shadow**. It is a great tool for creating box shadows as well as a host of other CSS3 effects

File separation of your application

The folder structure within this tutorial may seem excessive, but when integrated with a larger application, a well-defined structure makes maintenance and extension very easy to manage.

directive is bound to, and any attribute on that element. Then reference your new 'title' property and set in the scope within your directive Template.

```
001    scope: {
002    title: "=myTitle"
003    },
004    controller: function($scope, $element, $attrs){
005    }
006    //scripts/directives/templates/gallery.html
007    <h2>{{title}}</h2>
008
```

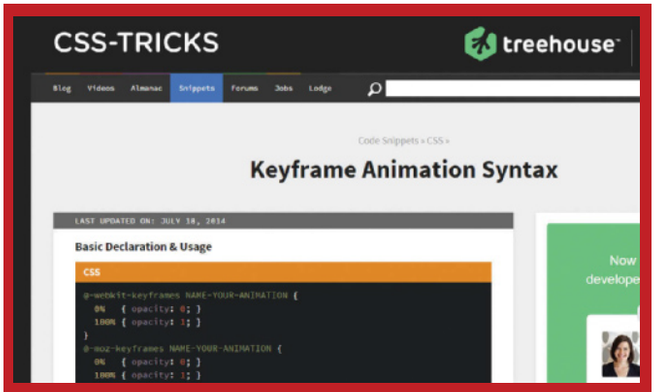
09 Add some images

Now the structure of the application is set up you can add some images. Add image files (both full size and thumbnail) to an images folder. Then create a JSON object within your 'galleryModule.js' file, include both the main and thumbnail paths to each image, along with a short description. This data could be generated from a web service provided by a CMS platform.

10 Pass your images to your directive

Now use the same technique as with the 'galleryTitle' property to pass the images array to your directive, using a new attribute named 'data-my-images'. Extend the scope of your directive to set these images and make them available to your gallery.

```
001    <body data-ng-app="galleryApp" data-ng-
controller="galleryCtrl">
002    <div data-my-gallery data-my-title="galleryTitle" data-my-
images="galleryImages" class="gallery"></div>
003    </body>
004    //scripts/directives/galleryDirective.js
005    scope: {
006    title: "=myTitle",
007    images: "=myImages"
008    }
```



<Above>
• To create more complex animation sequences try using keyframes in CSS. This is a good place to start **css-tricks.com/snippets/css/keyframe-animation-syntax**

11 Build your thumbnails

To display your thumbnails you can use an Angular repeater. This is a core Angular directive used for repeating a block of HTML based on an array of data. Add an unordered list to your directives template file, with 'data-ng-repeat' bound to the LI. You should now see your thumbnails in your gallery.

```
<h2>{{title}}</h2>
<ul class="inline">
<li data-ng-repeat="image in images"><img src="{{image.thumb}}" alt="{{image.description}}" /></li>
</ul>
```

12 Use an image selection method

Now add a 'data-ng-click' attribute to the repeated LI item calling a method named 'selectImage' and passing the 'image' property into it. Then add a new function to the directive controller to receive this image data. Once received you can log out the data and check in your browser console to check the correct image is being passed.

13 Add a main image

Within your directive template file gallery.html, add a div to hold a main image enabling you to display the larger version of the selected image. Then create a new $scope property within your directive controller, setting $scope.mainImage to the first item within the $scope.images array. Then change this $scope.mainImage within the $scope.selectImage function.

```
<h2>{{title}}</h2>
<div class="mainImage">
<img class="main" src="{{mainImage.main}}" alt="{{mainImage.title}}" />
</div>
<ul class="inline">
<li data-ng-repeat="image in images" data-ng-click="selectImage(image)"><img src="{{image.thumb}}" alt="{{image.description}}" /></li>
</ul>
//scripts/directives/galleryDirective.js
controller: function($scope, $element, $attrs){
$scope.mainImage = $scope.images[0];
$scope.selectImage = function(image){
$scope.mainImage = image;
}
}
```

14 Hide the main image

Set the '.mainImage' div as hidden in your CSS and then create a class of 'show' to display it. Now change the class attribute on this div to a 'data-ng-class' attribute. You can write a shorthand if statement directly into this attribute and bind the class to a new '$scope.showMainImage' property within your directive's controller.

15 Show the main image when selected

Within your directive controller set the '$scope.showMainImage' property to 'false' as default. Then set this property to 'true' within the selectImage function. Create a new function named '$scope.close' to set this property back to 'false'. Now add a close link into div.mainImage within your HTML and call the 'close' method using a 'data-ng-click'.

```
<div data-ng-class="showMainImage ? 'mainImage show' : 'mainImage'">
<a href="#" class="close" data-ng-click="close()">Close</a>
<img class="main" src="{{mainImage.main}}" alt="{{mainImage.title}}" />
</div>
//scripts/directives/galleryDirective.js
controller: function($scope, $element, $attrs){
$scope.mainImage = $scope.images[0];
$scope.showMainImage = false;
$scope.selectImage = function(image){
$scope.mainImage = image;
$scope. showMainImage = true;
};
$scope.close = function(){
$scope.showMainImage = false;
};
}
```

16 Handle default events

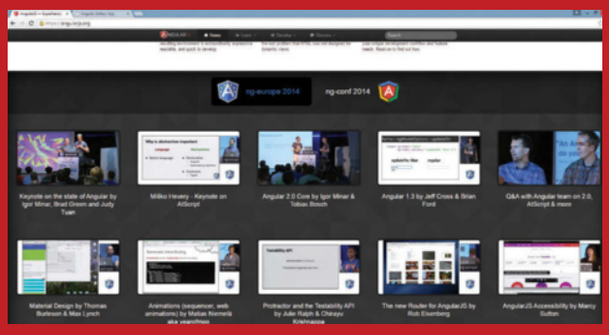
As the close functionality is now bound to an anchor link with a # href attribute, the browser window will scroll to top when clicked. You can use the preventDefault method to stop this. Simply pass $event to your 'close' method and then call preventDefault() on it within your directive controller.

```
<a href="#" class="close" data-ng-click="close($event)">Close</a>
//scripts/directives/galleryDirective.js
$scope.slide = {
$scope.close = function(e){
e.preventDefault();
$scope.showMainImage = false;
};
```

17 Add some depth

Your gallery may be looking quite flat at this point, to add some depth to your gallery you can use CSS3 box-shadows on both the thumbnails and main image. Create a shadow class within your CSS and apply it to the thumbnail images. Also add a larger box-shadow to the main image.

```
.shadow {
-webkit-box-shadow:0 4px 4px rgba(0, 0, 0, 0.3), 0 0 40px rgba(0, 0, 0, 0.1) inset;
```



<Above>
There is a wealth of insight and information available if you want to learn to develop with this framework. For more information, check out **www.angularjs.org**

```
    -moz-box-shadow:0 4px 4px rgba(0, 0, 0, 0.3), 0 0 40px
rgba(0, 0, 0, 0.1) inset;
    box-shadow:0 4px 4px rgba(0, 0, 0, 0.3), 0 0 40px rgba(0, 0,
0, 0.1) inset;
    }
    .mainImage img {
    -webkit-box-shadow: 0 0 80px #000;
    -moz-box-shadow: 0 0 80px #000;
    box-shadow: 0 0 80px #000;
    }
```

18 Position the main image

Within the CSS file provided with this tutorial the entire gallery is centred, however centring the absolutely positioned main image takes a little extra CSS. You may have used this technique in the past, but it is often forgotten and if you haven't used it before it can save you lots of CSS hacking.

```
    .mainImage {
    position: absolute;
    top: 130px;
    left: 50%;
    margin-left: -500px;
    width: 1000px;
    }
    }
```

19 Add some animation

To add an animation to your main image appearing you can use the CSS3 'transition' property. This will only work with CSS properties that have numerical values, so you will need to switch from using 'display' to 'opacity' to hide/show the main image. Then add a transition property to .mainImage to animate this change over one second.

```
    .mainImage {
    position: absolute;
    z-index: 10;
    display: block;
    top: 130px;
    left: 50%;
    margin-left: -500px;
    opacity: 0;
    width: 1000px;
    -webkit-transition: opacity 1s;
    -moz-transition: opacity 1s;
    transition: opacity 1s;
    }
    .mainImage.show {
    opacity: 1;
    }
```

20 Use pointer-events

You may notice that you cannot click on any of the thumbnails that are overlapped by the main image. This is because even though the main image is hidden with opacity, the browser interprets it as being an active HTML element, therefore blocking any click events. If you add 'pointer-events: none;' to .mainImage in your CSS this should be fixed.

```
    .mainImage {
    position: absolute;
    z-index: 10;
    display: block;
    top: 130px;
    left: 50%;
    margin-left: -500px;
    opacity: 0;
    width: 1000px;
    -webkit-transition: opacity 1s;
    -moz-transition: opacity 1s;
    transition: opacity 1s;
    pointer-events: none;
    }
```

21 Fix the close button

The only problem with using pointer-events is that they are inherited by child HTML elements. This means the close button no longer works. An alternative approach is to chain transition properties, set the z-index property on .mainImage to zero and then to ten on mainImage.show. Now you can add z-index as a second transition value and your animation should work fine.

```
    .mainImage {
    position: absolute;
    z-index: 10;
    display: block;
    top: 130px;
    left: 50%;
    margin-left: -500px;
    opacity: 0;
    width: 1000px;
    -webkit-transition: opacity 1s, z-index 1s;
    -moz-transition: opacity 1s, z-index 1s;
    transition: opacity 1s, z-index 1s;
    }
    .mainImage.show {
    opacity: 1;
    z-index: 10;
    }
```

22 Reuse your directive

Now you will have a gallery directive that is simple to integrate with any website build. Just ensure the gallery.directive file is injected into your application and then you can use the attribute 'data-my-gallery' wherever you need to. If you need to add further options to extend and configure your gallery you can add more to the scope of the directive itself.

GET THE CODE
http://www.filesilo.co.uk/bks-00009

Build a reactive web app with Angular-Meteor

Combine the best of both worlds with Meteor and Angular to unleash three-way binding

Meteor is a great way to add 'reactiveness' to your app. That means that as soon as a collection is updated all connected clients receive that data. Meteor uses a library called Blaze to perform live updating from the server. Blaze requires no effort from the developer and 'just works', but by design Blaze is simpler than Angular as it has a gentler learning curve. A side effect of this is that it makes architecting a large app in Blaze challenging.

Enter Angular. It's been around for six years now and has a myriad of 'best practices'. Whether you're building a larger app or have an existing Angular app that you wish to port, Meteor and Angular are a good team. Of course, Angular isn't the only option but it's the one that we're going to focus on today.

To see how the two work together we're going to build a collaborative list app. It'll take advantage of Meteor's reactivity and Angular's structure. By the end of the tutorial you'll have a good understanding of how they complement each other.

As of version 1.0 Meteor supports Windows but it will require a different installer, aside from that the steps should be identical.

1. Install Meteor

If you haven't installed Meteor already then copy the curl command. If you're on Windows then there's a standalone installer at **install.meteor.com/windows**. Then create a new project with 'meteor create', this will create three files and a .meteor folder. Run it with the meteor command (this is shorthand for 'meteor run').

```
$ curl https://install.meteor.com/ | sh
$ meteor create lys
$ cd lys
$ meteor
```

2. Angular Meteor

In a single step we've already created the project and got it running via a server! The secret glue to hooking Meteor and Angular up is to use 'angular-meteor' headed by Uri Goldshtein and nine other core contributors. We can add it to the project with 'meteor add' while running and Meteor will automatically restart.

```
$ meteor add urigo:angular
```

3. Lys markup

One of the files that Meteor created for us was lys.html. This is where the head and body content of our app will reside. We'll keep it simple for now with an Angular include pointing to a file we'll create shortly. The 'base' tag is required for Angular's routing service to work in HTML5 mode.

```
<!-- lys.html   ->
<head>
<title>Lys</title>
<base href="/">
</head>
<body>
<main ng-app="lys" ng-include="'index.ng.
html'"></main>
</body>
```

4. Module definition

Meteor automatically loads files and because of this it has a defined load order. Files in a folder called lib are loaded before other files so this makes it a good place to put our module definition file. Note that the files within the client folder are only delivered to the client and are not run on the server.

```
// client/lib/app.js
angular.module('lys', ['angular-meteor']);
```

5. Showing a list

We'll display a list of lists on the page in a section controlled by the ListController. The 'track by' expression is important here because it means that each item won't have an internal $$hashKey object added to it. This is important because properties starting with $ are protected in MongoDB and can't be added.

```
<!-- lys.html   ->
<section class="list-container" ng-
controller="ListController">
<ul>
<li ng-repeat="list in lists track by
list._id"></li>
</ul>
</section>
```

6. Toggle buttons

Within each list item we will now add two buttons: this will be one button to toggle editing and one button to remove it from the list. We are also using Angular's 'switch' directive to change which elements to show when we are editing. Because the input is tied to a model, updates will immediately be visible to Meteor and thus to all other users.

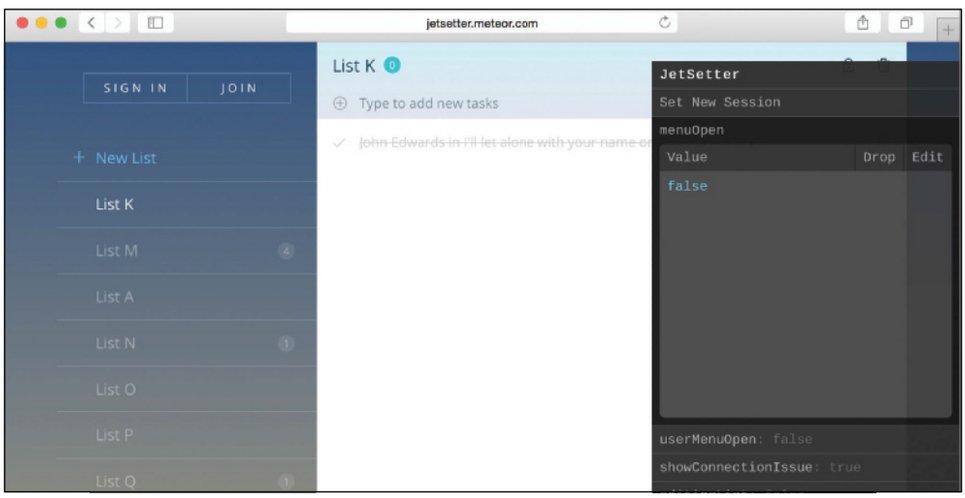
```
<!-- lys.html   ->
<span ng-switch="edit">
<input ng-switch-when="true" ng-model="list.
name">
<p ng-switch-default>{{list.name}}</p>
</span>
<button ng-click="edit = !edit">Edit</
button>
<button ng-click="remove(list)">Remove</
button>
```

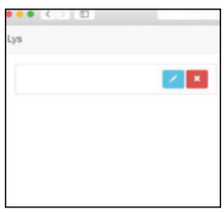
7. List controller

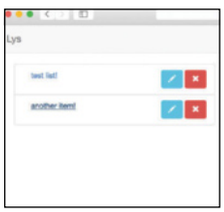
Our list controller is pretty standard but as well as passing in $scope we're also injecting $meteor. This is what angular-meteor provides through the extra dependency that we added to the module which gives us Angular-ified Meteor methods. Note that we're

Best of both worlds

You can still use Blaze in your application with Angular-Meteor, more documentation can be found at **angularjs.meteor.com**. You can change the delimiter from '{{' to '[[' as well.







Left
Mongol has a sister project called JetSetter which is a package that manages and visualises Session variables while developing

Top left
The buttons use icons from Twitter Bootstrap's glyphicon component aided with a title attribute

Top right
With a single line the collection is retrieved from the database and the work we've done so far renders it

referencing a variable called Lists, it's not in this file so where is it coming from?

```
angular.module('lys').
controller('ListController', ['$scope',
'$meteor',
function ListController ($scope, $meteor) {
$scope.lists = $meteor.collection(Lists).
subscribe('lists');
$scope.users = $meteor.collection(Meteor.
users).subscribe('users');
}
]);
```

8. List collection

Lys.js is another file that Meteor created for us when it created the project. This means that it's shared with both the client and the server. We're creating a global variable called Lists which is a database collection (a little like a table). This is what we referenced in our controller in the previous step.

```
// lys.js
Lists = new Mongo.Collection('lists');
```

9. List permissions

By default Meteor enables anyone to edit the database from the client (via the insecure package) and the database sends everything to the client (via the autopublish package). We'll remove these later but we need to specify under what conditions a list can be inserted, updated or removed. This will be based on either being an owner or collaborator.

```
Lists.allow({
insert: function (userId, list) {
return userId && list.owner === userId;
},
update: function (userId, list) {
return userId === list.owner || list.
collaborators.indexOf(userId) >   1;
},
remove: function (userId, list) {
return userId === list.owner || list.
collaborators.indexOf(userId) >   1;
}
});
```

10. Publish lists

As well as setting permissions we will also need to publish 'lists' which is what the client actually subscribes to. We only want to publish the lists that the logged in user has access to though so we will need to create a query here. Queries are built up through objects and arrays. Here we will look for lists where we're either the owner or a collaborator.

```
Meteor.publish('lists', function () {
return Lists.find({
$or: [{
$and:[{
owner: this.userId,
}, {
owner: {
$exists: true
}
}]
}, {
$and:[{
collaborators: {
$in: [this.userId]
}
}]
}]
});
});
```

11. Publish users

Similar to publishing lists we'll also want to publish all of our users. This is so that you can add collaborators to your lists. Note that we're returning everything (no conditions here) but only returning the field that will be used. This means that you can't accidentally leak data that the client doesn't need.

```
Meteor.publish('users', function () {
return Meteor.users.find({}, {
fields: {
emails: 1
}
});
});
```

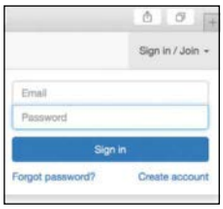
12. Add or remove packages

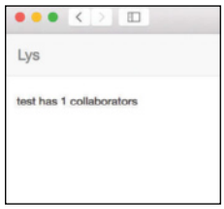
Next we'll add basic account handling with 'accounts-password' so people can log in (yes, it's that simple!). Then add Twitter Bootstrap and a version of accounts-ui which is specific to Bootstrap 3. We'll also remove the insecure and autopublish packages as described earlier. Chaining packages by adding a space between them is a useful and quick way to act on multiple packages.

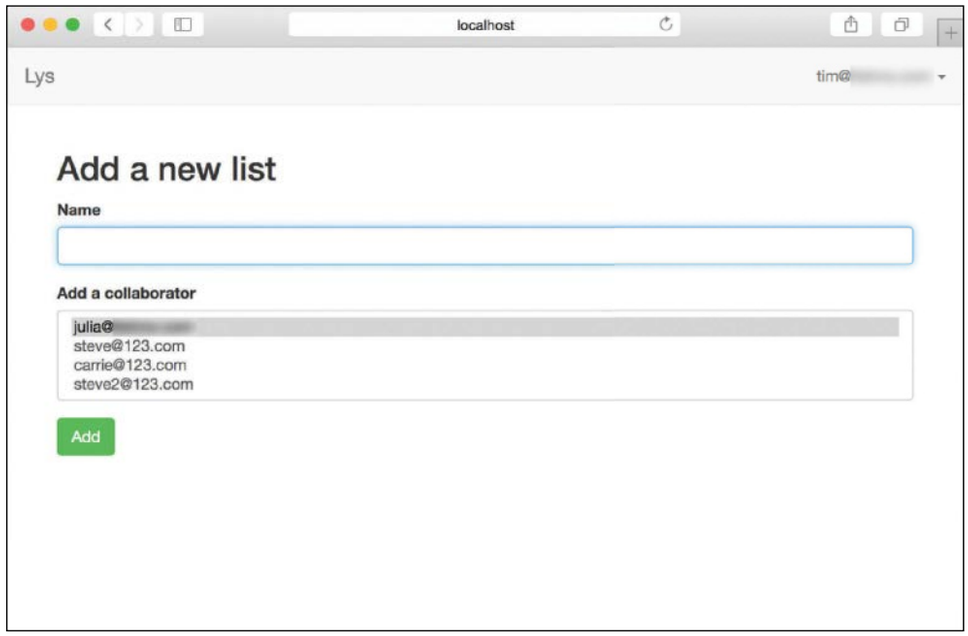
```
$ meteor add accounts-password
twbs:bootstrap ian:accounts-ui-bootstrap-3
$ meteor remove insecure autopublish
```

Still not convinced?

The Angular-Meteor Manifesto (**angularjs.meteor.com/manifest**) has a list of additional points that may convince you and also a roadmap for what's on the horizon.







Top left
The log-in buttons partial is very simple to use if you're happy with the default markup and styling

Top right
Although basic, the list detail view demonstrates how routing can be used in conjunction with state

Right
The collaboration list is filtered to remove the current user out of the list but this could be done server-side

13. Add login buttons

Accounts-ui gives us a partial called loginButtons. This is the same as the double curly brace syntax that Angular uses but this is Meteor's Blaze template engine. It's important that the two aren't confused. The rest of the structure is for Bootstrap's navbar component.

```
<!-- lys.html   ->
<nav class="navbar navbar-default">
<div class="container-fluid">
<div class="navbar-header">
<a class="navbar-brand" href="#">Lys</a>
</div>
<div class="nav navbar-nav navbar-right">
{{ > loginButtons }}
</div>
</div>
</nav>
```

14. Show form

$root here is the same as $scope.$root and accesses $rootScope without having to explicitly pass it to the template. The next steps will expand this form. $root also has the getReactively method which watches the variable for changes and notifies Meteor's Tracker library.

```
<form class="col-md-7" ng-show="$root.
currentUser">
<h2>Add a new list</h2>
</form>
```

15. Name input

The form will create an object we'll novelly call 'newList'. One of the pieces of data that we'll need is the name of the list so we'll make a form text input to capture this data. Note that we don't have to specify the type because the default type for an input is text.

```
<label for="nameInput">Name</label>
<input ng-model="newList.name"
id="nameInput" class="form-control"
required>
```

16. Collaborator list

The collaborator field will let the user select multiple users (but not themselves) to add as a collaborator via email address. Angular's select directive makes this trivial but the syntax is daunting. The key part is the ng-options attribute which says, "use the email address as the text and call each item in the users array user".

```
<label for="users">Add a collaborator</
label>
<select class="form-control" id="users"
name="users" ng-options="user.emails[0].
address for user in users | removeSelf"
ng-model="collaborators" multiple>
<option value="">Choose a collaborator</
option>
</select>
```

17. RemoveSelf filter

In the last step we referenced a filter called removeSelf and this filter will return all of the users that don't match the currently logged in user's ID. We are using a technique called 'duck typing' here by inferring from the fact that it has a length property – it'll be an array and also have a filter function.

```
angular.module('lys').filter('removeSelf',
function () {
return function (users) {
if (users.length) {
return users.filter(function (user) {
return user._id !== Meteor.userId();
});
} else {
return users;
}
};
});
```

18. Add button

To finish our form we'll create a button which will add newList to the Meteor collection. The code for this is a little more involved so to keep things tidy we'll call a method on the controller to deal with it rather than write it all inline. The semicolon is optional but insert them in case we add more, it's the new 'be kind rewind'.

```
<button class="btn btn-success" ng-
click="add(newList);">Add</button>
```

19. Add function

The 'add' function maps each collaborator to just their ID so rather than include the entire object for each collaborator we only store their ID which we can look up later on. We set the list owner to the currently logged in user by accessing the root scope and we will then push this to the list. This push will also automatically update the database for us!

```
$scope.add = function (list) {
list.collaborators = $scope.collaborators.
map(function (user) {
return user._id;
});
list.owner = $scope.$root.currentUser._id;
$scope.lists.push(list);
};
```

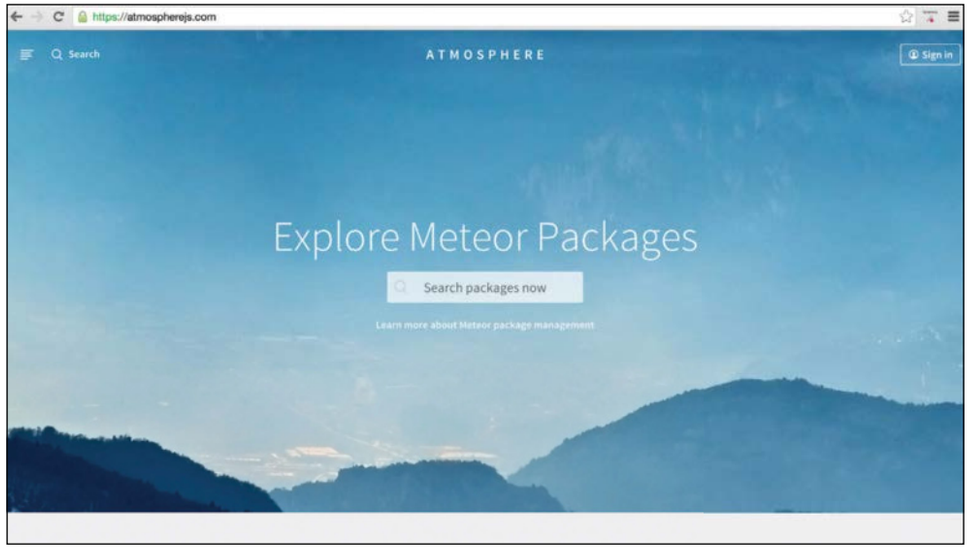
20. Remove a list

Removing a list is far simpler than adding it because all we need to do is call the remove function on the collection. This receives the list object and voila, it's removed from the database and each connected client sees the list vanish. Meteor collections have all sorts of exotic methods for you to play with like updateCursor and unregisterAutoBind.

```
$scope.remove = function(list) {
$scope.lists.remove(list);
};
```

21. Jumbotron message

You may have noticed that we had to create a 'special' HTML file suffixed with .ng.html. This is because otherwise Meteor will try to parse the Angular expressions and break. Next we'll create a message for logged out users explaining the benefits of what they'll



Where to find community code for your project

Using Angular and Meteor you have the opportunity to use packages from both communities. The de facto place for finding Meteor packages is Atmosphere (**atmospherejs.com**). Angular doesn't have a single place for modules but a good place to look is Angular Modules (**ngmodules.org**). Atmosphere is the Meteor equivalent of npm and also includes handy things like the command to paste in to add the package.

In development mode, Meteor can easily serve about 5MB of files (Firefox reckons it's closer to ten) but in production mode this is radically decreased to 240KB. This is done by running your app with the production flag, '$ meteor – -production'. Remember that each package that you add is going to increase this so ensure that you're using the package for the right reasons.

be able to do with an account. This works because currentUser is 'null' when there is no user.

```
<!-- index.ng.html   ->
<div class="col-md-12">
<div class="jumbotron" ng-hide="$root.
currentUser">
<div class="container">
<p>Login to create and share lists with
your friends and family!</p>
</div>
</div>
</div>
```

22. Angular routing

So we can now create a list and add collaborators to it. Let's introduce routing to our little app. Create a file called 'routes.js' inside of the client folder. HTML5 mode rewrites URLs to 'normal' URLs which fallback to hashbangs in older browsers. The next steps will sit inside this function.

```
// client/routes.js
angular.module('lys').
config(['$urlRouterProvider',
'$stateProvider', '$locationProvider',
function ($urlRouterProvider,
$stateProvider, $locationProvider) {
$locationProvider.html5Mode(true);
$urlRouterProvider.otherwise('/lists');
}]);
```

23. List route config

Angular's state provider comes from the ui-router package. Here we can specify what is rendered at a given URL and which controller it should be hooked up to. More information on controlling state can be found at **github.com/angular-ui/ui-router/wiki**.

```
$stateProvider
.state('lists', {
url: '/lists',
templateUrl: 'client/lists/views/lists.ng.
html',
controller: 'ListController'
})
```

24. List detail config

Likewise we'll now be doing the same for the details view. Note that this will take a URL parameter denoted by the colon. ListId will be the ID of the list in the database which we will use to get further information about it. Being able to specify the template and controller separately is extremely useful, especially if we want to reuse any of the templates.

```
.state('listDetails', {
url: '/lists/:listId',
templateUrl: 'client/lists/views/list-
details.ng.html',
controller: 'DetailsController'
});
```

25. Details controller

The second controller in our app will get the list we want and a list of users. We're going to show the number of collaborators but you could extend it to allow the addition or removal of collaborators to the list. We name the DetailsController function (convention would be to keep it anonymous) for readable stack traces when we are debugging.

```
angular.module('lys').
controller('DetailsController', ['$scope',
'$meteor', '$stateParams',
function DetailsController ($scope, $meteor,
$stateParams) {
// next step
}
]);
```

26. Meteor reactivity

$meteor.object wraps a Meteor object to give it 'reactivity'. That is, the ability to update when other users' version of that object updates. Note also the second argument of 'false' to the Meteor collection. This toggles automatic client-side saving which we want to disable for 'users', by default this is true.

```
$scope.list = $meteor.object(Lists,
$stateParams.listId).subscribe('lists');
$scope.users = $meteor.collection(Meteor.
users, false).subscribe('users');
```

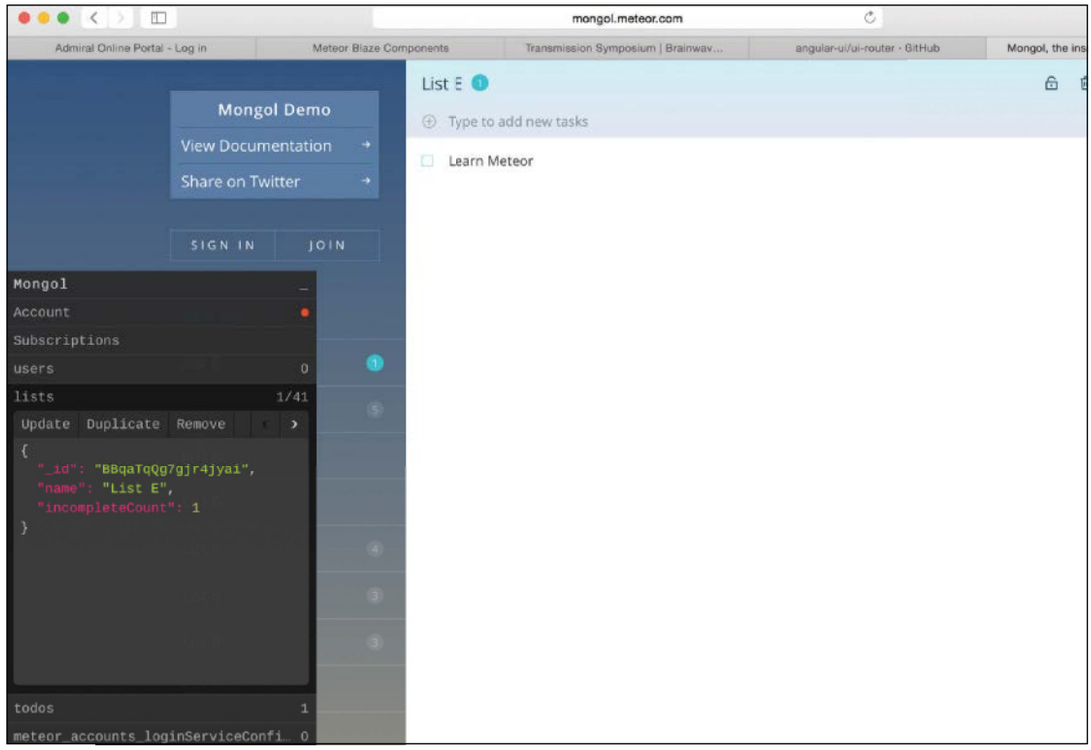
27. Render to page

Lastly we output the name of the list and the number of collaborators into the template. As an exercise for the discerning reader, see if you can get a list of email addresses to display for each collaborator. Harnessing Angular's principles with Meteor is a powerful way to architect your app.

```
<p>{{list.name}} has {{list.collaborators.
length}} collaborators</p>
```

Monitor and debug with Mongol

An invaluable package while developing Meteor apps is msavin's Mongol (**mongol.meteor.com**). Add it to your project with '$ meteor add msavin:mongol' and wait for Meteor to restart. You shouldn't notice anything different until you hit Cmd/Ctrl+M in the browser and a black window pops up in the bottom left. Mongol gives you real-time visibility of all the collections that are currently published to the page. This includes users! Here you can add, edit, duplicate and remove documents (analogous to records). It's useful for working out if something has made it to the database or not and can help with debugging. Mongol's smart enough to remove itself when built or deployed unless you're in debug mode.



GET THE CODE
http://www.filesilo.co.uk/10k5-0009

ADVANCED ANGULAR

Get your hands on an unmissable collection of tips and techniques to take your Angular code to the next level

BE AN ANGULAR EXPERT

AngularJS is a client-side MVW framework written in JavaScript, aimed at writing single-page applications (SPAs). Initially released in 2009, Angular has surfaced as one of the most popular choices for developers. It's maintained by Google as an open source project and boasts a lot of functionality built with modern web standards in mind. MVW stands for Model-View-Whatever, which gives us flexibility over design patterns when developing applications. We might choose an MVC (Model-View-Controller) or MVVM (Model-View-ViewModel) approach with it.

Angular focuses on our data, enabling our HTML to become the dynamic language that it was certainly not built for. It brings powerful concepts and ideas from various programming languages and has server-side influences that make it a top choice for web application development at present. Angular parses our declared HTML, binds it to our models that consist of plain-old JavaScript objects and keeps everything in sync with the DOM to spring our application to life. This synchronisation of DOM and JavaScript is called two-way data-binding – it is one of Angular's most powerful features amongst others such as automated DOM creation or destruction, powerful conditional logic, automated event listeners, template encapsulation, class-like components, dependency injection (DI) and promises.

Angular focuses on a fully data-driven approach to developing applications without needing to refresh models, update the Document Object Model (DOM) and other time-consuming tasks such as ironing out browser quirks and structuring our code consistently. model data can be set manually or fetched via XMLHttpRequest (XHR), which is usually a JSON resource, making it extremely easy to integrate within Controllers.

In this feature, we've packed tons of tips that originate from expert industry experience with AngularJS in the last few years, from simple time-saving to high-end performance enhancing. We're also going to create an custom Angular filter for enabling us to filter arrays of content at speed and reflect them in the UI. directives are a huge part of Angular. We'll also be looking at creating our own simple directive, focusing on structure and good practices, built-in features and more, to take away and apply to your development techniques.

REASONS FOR DIRECTIVES AND ENCAPSULATION

Modern web standards are rapidly approaching, more specifically the Web Components standard. Web Components are an exciting new dimension of how we'll build web documents in the future, with the intention to bring a component-based approach to developing web applications. These components will comprise of many small, reuseable and possibly interlinking parts.

Encapsulate in a Web Components fashion from day one and your applications will have the ability to scale with ease, be well tested, and offer flexibility and reusability. Angular lets us pretty much do all this already (and more) through its directive API, opening up this new paradigm to developers right now.

Understanding Web Components and knowing how to implement Web Components concepts in Angular today will help you succeed in writing better components overall.

So let's see how each of these specifications is defined in the Web Components standard and how Angular lets you benefit from similar features today.

Custom Elements:

What Web Components standard says:
Lets you define your own HTML elements.

What Angular lets you do already:
Angular offers the concept of directives to let you define your own custom HTML elements. You can create new HTML elements or augment existing ones. You can also add behaviour to HTML attributes.

Shadow DOM:

What Web Components standard says:
Gives you the ability to scope markup and styles in a separate DOM tree.

What Angular lets you do already:
Shadow DOM requires browser support so it's technically not possible for Angular to implement it if your browser does not support it. However Angular already provides similar benefits from a developer standpoint. Angular enables you to encapsulate markup using a HTML template, logic using a directive controller and content using an isolated scope. It also offers transclusion, which can be thought of as a single insertion point from the Shadow DOM specification.

Used for encapsulating and scoping DOM, Angular mimics this behaviour by compiling HTML and injecting it into our DOM.

The idea behind Shadow DOM is to separate content from presentation, creating a clear distinction between where our templates contain placeholders require content, and the content itself to be parsed and injected. Shadow DOM is technically a 'DOM within DOM', it's entirely scoped and you can create as many instances of each component as you need.

HTML Imports:

What Web Components standard says:
Provides a way to include and reuse HTML documents in other HTML documents.

What Angular lets you do already:
Our components need dependencies, so we need to import them using HTML Imports. Angular actually does all this for us, and we tie our imports much closer to the component definition itself with Angular, as the HTML Imports standard actually requires us to import in the <head> of our document.

HTML Templates:

What Web Components standard says:
Enables you to define blocks of markup with the ability to inject dynamic content into.

What Angular lets you do already:
Angular enables us to use string templates or reference HTML files, enabling us to use a method best for our team, us or the project. Our templates contain no content, Angular uses {{ handlebar }} syntax for adding placeholders for where our content will be injected, compiled and finally inserted in the DOM.

BUILD A REUSEABLE FILTER

Filters are really powerful ways to manipulate data. A filter could parse data into a different output, such as converting millisecond dates into a human readable format, or filter content (like lists) by returning items that match certain criteria. Let's look at the Angular .filter() API, create a dummy email list and make an ng-repeat filter.

1. Change the drawing radius

Let's hook into the Angular module that we've made called 'app' and extend it further. Inside the filter's callback, return another function – this function is called with filter arguments and will return a filtered response.

```
angular
.module('app', [])
.filter('important', function important() {
return function () {
//
};
});
```

2. Hook up the DOM

Create ng-repeat with filter intentions in place. Email in vm.emails and a pipe | declares the 'important' filter name.

```
<ul>
<li ng-repeat="email in vm.emails |
important" ng-class="{important: email.
important}">
{{ email.label }}
</li>
</ul>
```

Ng-class here adds 'important' class for emails with an 'important: true' property to help style them differently.

3. Pass in the array to be filtered

Next allow for some arguments, and name the first one 'items'. Let's 'return' the items to let the filter work, as we don't return anything, Angular won't get an array back to rerender the DOM with our updated filter.

```
angular
.module('app', [])
.filter('important', function important() {
return function (items) {
return items;
};
});
```

4. Define important properties

Assuming that we now have data coming from our backend, we can make it available in our Controller, and pass it to our filter. Before it hits the filter, we need to make sure we have an 'important' property containing a Boolean value.

```
angular
.module('app', [])
.controller('MainCtrl', function MainCtrl()
{
this.emails = [{
label: 'Item dispatched',
important: true
},{
label: 'Reset your password',
important: false
},{
label: 'Renew your home insurance',
important: true
},{
label: 'Welcome to Amazon!',
important: false
},{
label: 'Angels & Airwaves ticket
confirmation #402881',
important: true
},{
label: 'TravisCI test passing',
important: false
}];
});
```

5. Return the important objects

Our filter function now only looks at returning the important emails. Create a new empty array called 'filtered', loop through the 'items' (which Angular will pass in the bound ng-repeat for us) and check if the important property is 'true'. If 'true', we use array.prototype.push to add that object into the filtered array, and finally return it once iteration is complete, giving us all important objects.

```
.filter('important', function important() {
return function (items, important) {
var filtered = [];
for (var i = 0; i < items.length; i++) {
if (items[i].important) {
filtered.push(items[i]);
}
}
return filtered;
};
})
```

6. Toggle capability

We could make the filter better by adding in an optional parameter to enable us to toggle between 'All' emails and 'Important' emails in the same list. We can pass in a second argument to the DOM filter using a colon to separate arguments. Let's add a checkbox to enable us to reflect the showImportant model Boolean value. Once checked, showImportant becomes 'true', and our filter will run again with the second argument available as 'true', enabling us to decipher between whether the UI is requesting 'all' or 'important' emails.

DEPENDENCY INJECTION ANNOTATION SYNTAX

Dependency Injection (DI) in Angular is fairly simple, we just pass the dependency names that we want to the function as arguments, and Angular injects them in for us to use. Behind the scenes Angular converts our functions to a string and then it reads any dependencies that we're requesting. Note that at this point the arguments cannot be renamed.

When we minify our code (and we should), the argument names will be minified and Angular won't be able to interpret them correctly, so we need to explicitly tell Angular what dependencies we're requesting, and in what order we need them.

There are several ways to inject our dependencies to keep them safe when minified, for example by making use of the inline $inject syntax or array syntax. With the array syntax, we will need to pass the dependencies into our callback as an array, which will then hold the function as a final argument.

```
angular
.module('app', [])
.controller('MainCtrl', ['$scope',
'$rootScope', function MainCtrl($scope,
$rootScope) {
//
}]);
```

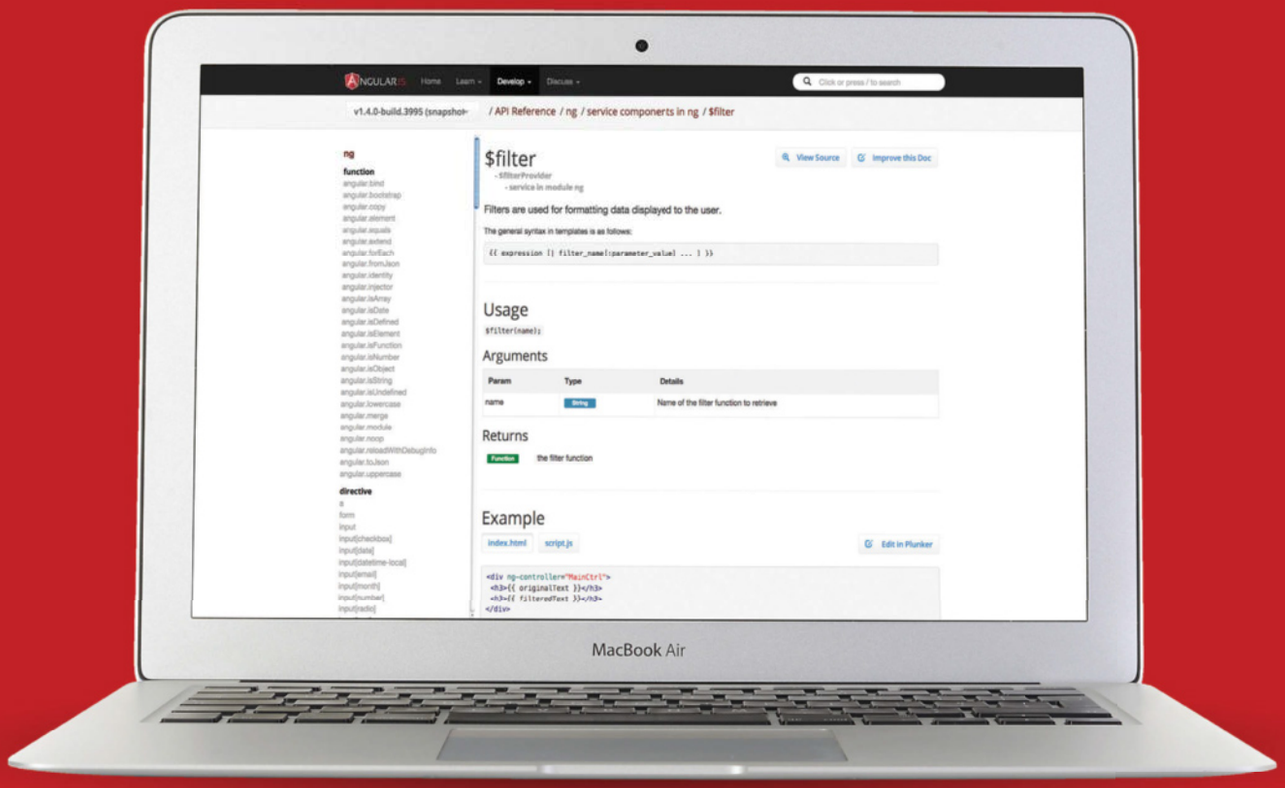
With the $inject syntax, we can create a property on the function itself, which will offer us more flexibility if we want to make the code slightly less callback-looking and enhance readability.

```
function MainCtrl($scope, $rootScope) {
//
}
MainCtrl.$inject = ['$scope', '$rootScope'];
angular
.module('app', [])
.controller('MainCtrl', MainCtrl);//
}]);
```

We can alternatively use a mix of both if preferred (using the array syntax but also passing a function in).

```
function MainCtrl($scope, $rootScope) {
//
}
angular
.module('app', [])
.controller('MainCtrl', ['$scope',
'$rootScope', MainCtrl]);
```

Use ng-annotate to automate both of these syntaxes so we can save a lot of time typing, it's clever enough to know whether to use $inject or the array syntax. You can install ng-annotate via NPM, point it to your project files, let it compile and automatically add the dependency injection annotations for you.




```
<input type="checkbox" ng-
model="showImportant">
<ul>
<li ng-repeat="email in vm.emails |
important:showImportant" ng-
class="{important: email.important}">
{{ email.label }}
</li>
</ul>
```

7. Augment arguments

As we've declared another argument in the DOM, we can get another argument available in the JavaScript filter code, aliased as 'important'. Add an if statement to check if 'important' is passed in, otherwise return all array items.

```
.filter('important', function important() {
return function (items, important) {
if (important) {
// return only important emails
}
// if important is false, we'll return all
items
return items;
};
})
```

8. Refactor our filtering

Now we have an 'important' flag being passed in, we can move our earlier code into the 'if' statement, and run our filter when we want all our important Objects back, otherwise we'll just assume the user requests 'all' and send the entire collection back.

```
.filter('important', function important() {
return function (items, important) {
if (important) {
var filtered = [];
for (var i = 0; i < items.length; i++) {
if (items[i].important) {
filtered.push(items[i]);
}
}
return filtered;
}
return items;
}; })
```

ONE-TIME BIND WHERE IT IS NEEDED

Angular 1.3 introduced a brilliant feature called one-time binding. Using one-time binding is good as it removes $$watchers from the $digest cycle after they've been parsed and populated. If the data is a one-time static render then it can be bound once and removed from further $digest loops, this will help to keep future loops much lighter and therefore faster for the JavaScript to look up any change. This feature will also enable us to remove the value from Angular $digest cycle (which contains all our data to check states when any model values change) as soon as it becomes anything other than 'undefined'.

The $digest cycle can become heavy and slow down our app, which isn't very helpful if we only need to render our data once, for example with a dynamically populated navigation, static list content or view titles. To specify that an expression only needs to be bound once, we can prefix the expression with '::', like '{{ ::name }}'.

MATT GIFFORD
monkehworks.com
Consultant developer, Monkeh Works Ltd.
@coldfumonkeh

"AngularJS is not just for single-page web applications or browser-based sites. The implementation of AngularJS within development stack of frameworks, for example, as Ionic enhances the effectiveness of PhoneGap or Cordova mobile development."

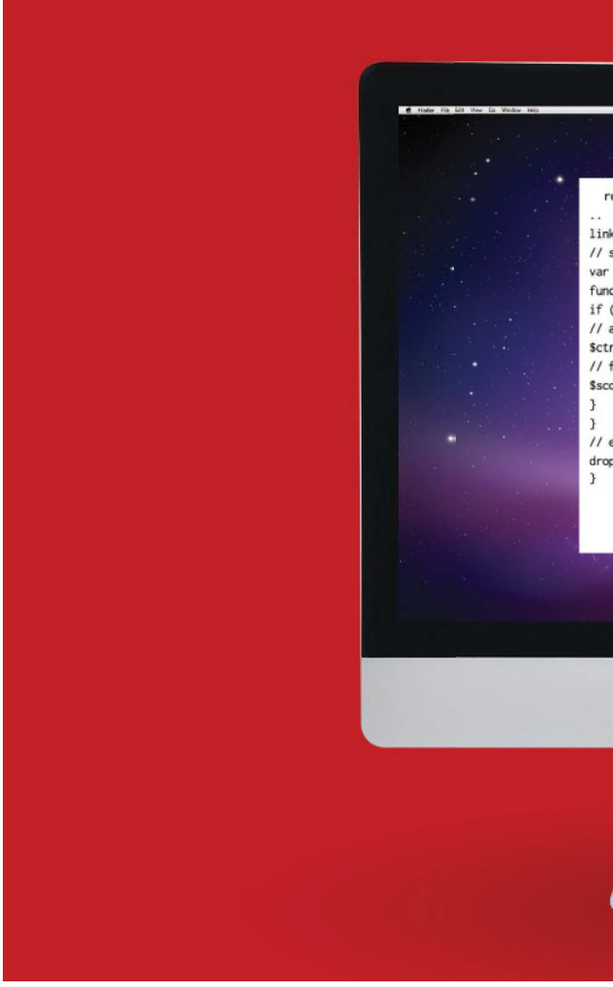
LINK FUNCTIONS FOR DOM

It may be extremely tempting to litter your Controllers with DOM manipulation, especially when integrating things such as plugins or third-party scripts and you need to set or get values. Directives are a perfect way to encapsulate any necessary presentational logic, as they include a Controller, and offer a gateway to the DOM through the 'link' function. This 'link' function gives us the root element for the directive, so that we can actually access any element inside it and bind things such as ontouchstart listeners for example, which currently aren't part of the Angular core.

When using a Controller alongside a directive, Angular passes it to the 'link' function as a fourth argument, so we can run any presentational logic from callbacks to raw DOM manipulation.

Remember that inside raw DOM listeners we will need to run $scope.$apply to tell Angular to look at any new values. As an example (excluding most directive properties) it is possible to pass in a Controller, aliased $ctrl to access presentational methods inside our raw DOM event listeners.

```
return {
..
link: function postLink($scope, $element,
$attrs, $ctrl) {
// some code to fetch files from a made up
drag/drop uploader
var drop = $element.find('.drop-zone')[0];
function onDrop(e) {
if (e.dataTransfer && e.dataTransfer.files)
{
// assumes "uploadFiles" method handles the
upload!
$ctrl.uploadFiles(e.dataTransfer.files);
// force a $digest cycle
$scope.$apply();
}
}
// events
drop.addEventListener('drop', onDrop,
false);
} .. }
```



UNBIND $ROOTSCOPE.$ON LISTENERS

You'll likely find the need at some stage during your Angular career to use the internal events system, $emit, $broadcast and $on. These event methods are available in both the $rootScope and $scope.

If we change views in our application, Angular will destroy our Controller and thus it will destroy the $scope object associated with it. This is great, however, all $scope objects are child objects that will inherit from the $rootScope, and this will then persist throughout the entire application.

We will need to manually unbind $rootScope listeners by calling the returned closure function when the $scope is destroyed by listening to the $destroy event, otherwise when we revisit a view where the same Controller is used, the $rootScope.$on will be bound again. This can then cause duplicated events and also some issues with the data syncing.

```
var unbind = $rootScope.$on('fooEvent',
function () {});
$scope.$on('$destroy', unbind);
```

With multiple events, we tend to use an array and loop through to automatically unbind.

```
[
$rootScope.$on('fooEvent' function () {}),
$rootScope.$on('barEvent, function () {})
].forEach(function (unbind) {
$scope.$on('$destroy', unbind);
});
```

ABSTRACT BUSINESS LOGIC INTO SERVICES

If you've ever been tempted to use things like $http (Angular's XHR wrapper) inside a Controller, then it's time to understand the patterns you're using. A Controller is a glorified ViewModel, a $scope object, which consists of maintaining and presenting (you guessed it) the presentational layer. Controllers should only withhold presentational logic.

Angular provides us with service, factory and provider APIs, to delegate business logic into. This makes sure our presentational layer is separated clearly from our business logic. Business logic would usually hit an API endpoint, fetch data and a Controller will make a copy of that data, pass it to the $scope for Angular to express what our model currently looks like.

AVOID DOM FILTERING

Using a filter in the HTML of Angular is great, but can impact performance. Filters can be easily added using a pipe inside an expression like so:

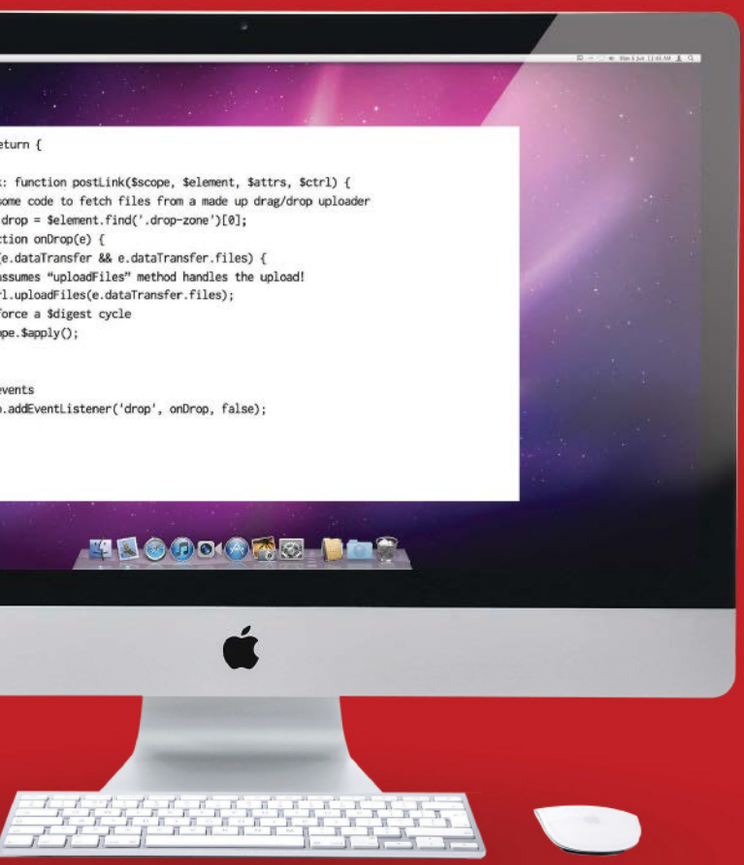
```
{{ someDateValue | filter:date }}
```

That's an example of using Angular's built-in date filter, we can however write our own filters. Filters are fairly common on ng-repeat declarations however, and can be huge performance killers.

Filters run on every $digest loop and will create a new array every time that it runs. Watch for changes to a specific model (such as a user typing) or a user interaction (such as a user clicking a button) and then run a filter inside the Controller's JavaScript using the $filter method, which will manually run your own filter upon detection of any change.

TIM STONE
fetimo.com
Lead front-end developer, Redweb.
@timofetimo

"Updating your UI to react to changes can be a messy, hard to maintain mess. Angular abstracts this by dealing with the DOM for you. You then start to think of your applications state, which is a fundamental shift."



$ROOTSCOPE.$APPLYASYNC FOR BATCHING XHR

The Angular 1.3 release saw another quietly brilliant feature, $rootScope.$applyAsync for intelligent XHR batching. Angular uses 'dirty-checking' to make sure all values are up to date, it does this by running what it calls a $digest cycle. The $digest can have a lot of data in, making UI response quite slow at times due to high watcher count. If we have multiple XHR calls happening around the same time, Angular is going to run a $digest after each of these to make sure all of its values are up to date. That's a lot of potential $digest loops happening around the same time. Using $applyAsync mitigates this issue and can result in significant performance improvements for apps making many concurrent HTTP requests. It's easy to use, so why not just drop it in to your config to see how it works?

```
angular
.module('app')
.config(function ($httpProvider) {
$httpProvider.useApplyAsync(true);
});
```

UNIT TEST ANGULAR MOCKS

Karma can run your unit tests whether in a physical browser (such as Chrome) or headless (PhantomJS). Angular mocks (from the Angular core team) extends Angular with extra functionality that you may need inside your unit tests. For example, Angular mocks makes it easy to test Controllers, directives and services (and more) by letting you mock HTTP requests, test async methods and simulate responses.

WHEN TO USE NG-BIND OVER HANDLEBAR EXPRESSIONS

Angular provides us two ways to insert text into our HTML, using the ng-bind or expression {{ }} syntax. An issue you will face if you have HTML content already on the page whilst the browser is parsing the HTML, before Angular has even loaded, is you'll see these curly expression braces, which then populate with data as Angular springs into life. To get around this we can use ng-bind, which will inject the value once Angular has loaded, and we'll just see an empty element which is a better experience than expression tags everywhere.

```
<!-- expressions -->
<p>{{ foo }}</p>
<!-- ng-bind -->
<p ng-bind="foo"></p>
```

Any views that are loaded with Angular routers, such as ui-router or ngRouter will compile all HTML and data before injecting the view, so you only have to cater for this issue on initial load.

FAVOUR NG-HREF ATTRIBUTES OVER HREF

Avoid 404 errors whilst the browser downloads content with expressions in the HTML. The browser will begin downloading <img src="images/{{ profile }}.jpg"> rather than the parsed value (ie app/images/niels.jpg). Using ng-href will overcome this, as Angular will populate and set href attributes for us after the expressions have been evaluated.

STANDARDISE HTML BINDING SYNTAX

Angular promotes the use of their directives like ng-click. However ng-click can also be written as x-ng-*, ng:*, ng_* and data-ng-*. If you need to be standards-compliant with your attributes, use data-ng-*, though this is most commonly seen as ng-* prefixed.

NAME ANONYMOUS FUNCTIONS

If you're passing in anonymous functions into Angular's API, such as .controller() then name the function you are passing in. This allows '.controller('MainCtrl', function() {...});' to become '.controller('MainCtrl', function MainCtrl() {...});', which really aids in stack-trace debugging in the dev tools.

BE RESPONSIBLE AND AVOID LONG API CHAINS

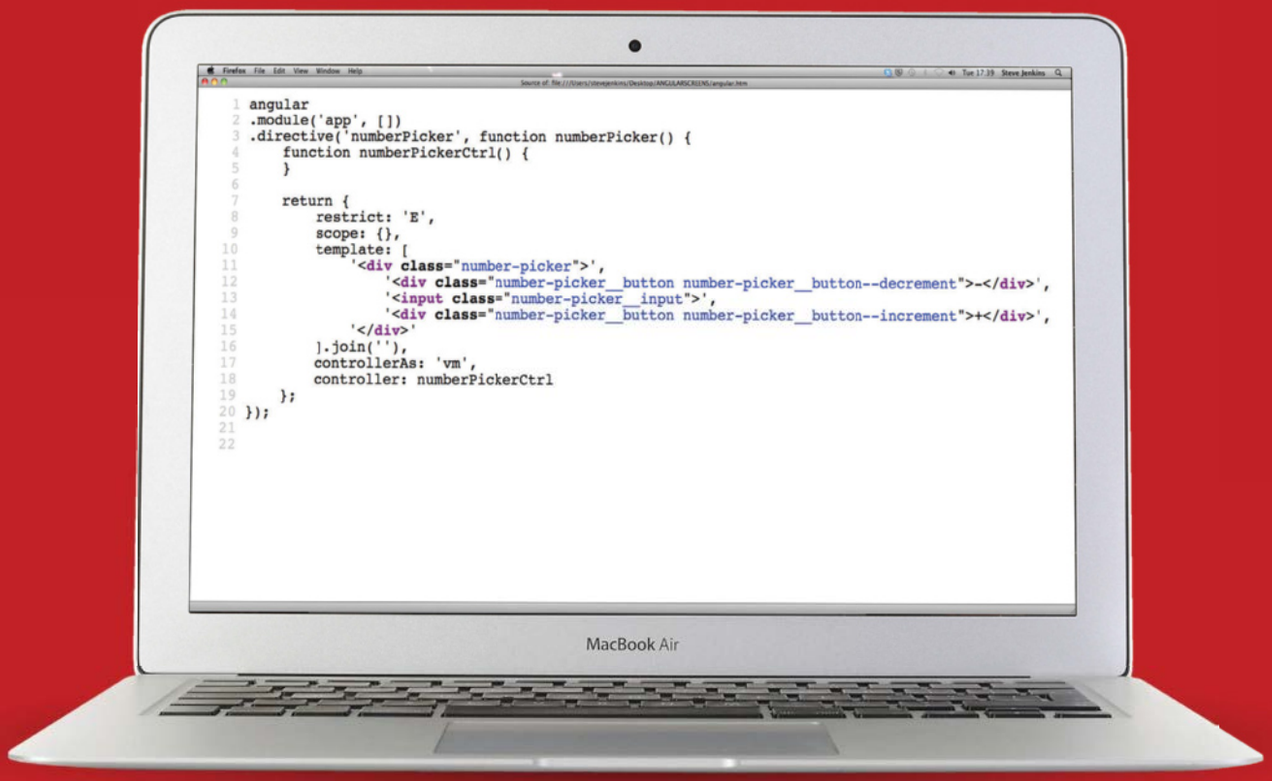
Aim to keep one component per file, a single responsibility. Only chain what is necessary, for instance a MainCtrl.js file will contain:

```
angular.module('app')
.controller('MainCtrl', function MainCtrl()
{...});
```

DOWNLOAD AND USE BATARANG

Brian Ford (Angular core team member) launched Batarang for AngularJS developers. Batarang is very useful as it "extends the Chrome Developer Tools, adding tools for debugging and profiling AngularJS applications."

"The $digest can have a lot of data in, making UI response quite slow at times due to high watcher count. If we have multiple XHR calls happening around the same time, Angular is going to run a $digest after each of these to make sure all its values are up to date. That's a lot of potential $digest loops happening"



REUSEABLE DIRECTIVE FOR INCREMENTING OR DECREMENTING NUMBERS

This directive will use two-way data binding to take care of updating a parent model from any $scope, through the use of isolate scope. Encapsulating logic enables us to reuse it, and isolating $scope enables us to use the same directive in many different places.

1. Custom elements

Before we begin, let's create a custom element named 'number-picker', which will have an attribute called 'value' (this can be named something different). The 'expression' is just a placeholder for a model value we're passing later.

```
<number-picker value="expression"></
number-picker>
```

2. Directive declaration and HTML

Next we'll create a basic directive definition, and add a template property. Directives can use templates from a string or from an external file using templateUrl (pointing to a relative URL). Let's set up the base directive properties and add our HTML template in.

```
angular
.module('app', [])
.directive('numberPicker', function
numberPicker() {
function numberPickerCtrl() {
}
return {
restrict: 'E',
scope: {},
template: [
'<div class="number-picker">',
'<div class="number-picker__button number-
picker__button--decrement">-</div>',
'<input class="number-picker__input">',
'<div class="number-picker__button number-
picker__button--increment">+</div>',
'</div>'
].join(''),
controllerAs: 'vm',
controller: numberPickerCtrl
};
});
```

3. Isolate scope binding

We need to pass the updated value in the directive back to where it was initialised after changes have taken place, this lets us keep other data models in the parent Controller updated. Pass the value in via the scope property, creating a new object to initiate 'isolate scope', and creating a property name called 'value'. We can use = as the value, which tells Angular to bind the value two ways, back up the parent Controller.

```
.directive('numberPicker', function
numberPicker() {
 ...
return {
...
scope: {
value: '='
},
...
};
});
```

4. Controller methods

We need to implement methods to increase or decrease our number inside the Controller. As we're using the controllerAs syntax, we need to attach these to the function instance using the 'this' keyword rather than injecting $scope.

```
angular
.module('app', [])
.directive('numberPicker', function
numberPicker() {
function numberPickerCtrl($scope) {
this.increment = function () {
};
this.decrement = function () {
};
}
...
});
});
```

5. Model manipulation

As our value is passed through $scope, we access it using $scope.value to pass back up to the parent model and not with this.value.

```
angular
.module('app', [])
.directive('numberPicker', function
numberPicker() {
function numberPickerCtrl($scope) {
this.increment = function () {
++$scope.value;
};
this.decrement = function () {
--$scope.value;
};
}
...
});
```

6. Controller methods

Now we've got Controller methods to modify our value, we need to make sure our HTML template can talk to those methods through Angular's built-in directives, we need ng-click. As we're using controllerAs, the Controller is instantiated as a reference object in the DOM as vm (ViewModel) so we can access our public increment or decrement methods through the vm object.

```
angular
.module('app', [])
.directive('numberPicker', function
numberPicker() {
...
return {
...
template: [
'<div class="number-picker">',
'<div class="number-picker__button number-
picker__button--decrement" ng-click="vm.
decrement()">-</div>',
'<input class="number-picker__input"
ng-model="value">',
'<div class="number-picker__button number-
picker__button--increment" ng-click="vm.
increment()">+</div>',
'</div>'
].join(''),
...
};
});
```

7. Add $watch to validate input

As we enable the user to type into the input, we need to ensure that they only enter a number. We can use $watch to watch changes to the value, and reset it back to its old value if it's anything other than a number. We can use a simple Regular Expression to test if anything isn't a number, and revert it back to the previous value. This is just basic validation, a more comprehensive solution may fit your needs in a production application. We could use ng-pattern, but it still enables users to type anything, and Angular will refuse to set the model unless the element matches the pattern.

```
angular
.module('app', [])
.directive('numberPicker', function
numberPicker() {
function numberPickerCtrl($scope) {
$scope.$watch(function () {
return $scope.value;
}, function (newVal, oldVal) {
if (!/^[0-9]+$/.test(newVal)) {
$scope.value = oldVal;
}
});
...
}
...
});
```

8. Link function keyboard input

We're also going to let the user press up or down to increment or decrement the number. We can do this via the 'link' property on the directive, which gives us access to the root node of the directives instance, as well as the $scope and any attributes it has. Set up placeholders for handling up and down arrow keys through the keyCode property on the events passed to us.

```
angular
.module('app', [])
.directive('numberPicker', function
numberPicker() {
function numberPickerCtrl($scope) {
$scope.$watch(function () {
return $scope.value;
}, function (newVal, oldVal) {
if (!/^[0-9]+$/.test(newVal)) {
$scope.value = oldVal;
}
});
this.increment = function () {
++$scope.value;
};
this.decrement = function () {
--$scope.value;
};
}
function numberPickerLink($scope, $element,
$attrs) {
function handleKeyUp(e) {
if (e.keyCode === 38) {
}
if (e.keyCode === 40) {
}
}
$element.find('input').on('keyup',
handleKeyUp);
}
});
```

9. Pass in Controller

You can also pass through the Controller into the 'link' function as the fourth function argument (aliased as $ctrl) to make a clear distinction between where we hold presentational logic and DOM logic. We should ideally only use 'link' for raw DOM communication. This means we call our existing methods to manipulate the value, without polluting our Controller with DOM logic – a big antipattern in Angular. We need to run $scope.$apply() after raw DOM APIs to inform Angular of changes and to update any values that we updated ourselves.

10. Final touches

Let's see what everything looks like together and check FileSilo for the full code. Using the expression "{{ vm.value }}" we can get updates as the user types which are instantly reflected back up to the parent Controller, as it's the property delegated to the directive, manipulated and passed back up.

TODD MOTTO
@toddmotto
The Google developer expert and director of front-end engineering at Mozio gives us five Angular tips.

USE DIRECTIVECTRL STRING METHODS

Instead of assigning a Controller and using an anonymous/assigned function, use a string-based approach. This references a Controller created using the .controller() method and makes the Controller reuseable elsewhere in the application and testing much easier.

APPLY NG-CLOAK CLASS NAME

Elements with conditional statements such as ng-if, ng-show or ng-hide are sometimes visible until Angular has evaluated the expression. We can use ng-cloak attribute, which is essentially styled as display: none;. After Angular has evaluated expressions, it removes all ng-cloak attributes leaving the element in its correct conditional state.

ROUTING 'RESOLVE' PROPERTY

Ensure asynchronous operations are complete by the time Angular switches views, this ensures all data is fetched prior to rendering the view. The resolve property sits on both the Angular router (ngRouter) and ui-router. We can pass in services that resolve a promise, these resolved promises are then given to us in the Controller through dependency injection.

$SCOPE IS JUST A VIEWMODEL

Treat your $scope as a ViewModel at all times, never use DOM logic or even think to include jQuery! All a Controller should do is make a copy of data that's provided to it from a service, and express that model through Angular's ng-* directives. Angular Controllers are our presentational layer, and shouldn't be mixed with DOM logic or business logic.

TESTABLE DIRECTIVES THROUGH ABSTRACTION

Lessen the impact of testing directives by abstracting as much as you can into Controllers and services (providing the architectural pattern is correct for your code). This enables us to bullet-proof our workflow by writing tests for the Controller and service independently, and it also enables our directive test to be smaller and less data focused.