Contents

1 Welcome **3**

2 The short guide to service definitions **5**

3 A short guide to routing **11**

4 Preventing flickering **15**

5 Conditionally applying CSS classes **19**

6 Setting configs in angular **23**

7 The Power of Expression Binding **29**

8 Optimizing Angular: the view (part 1) **35**

9 Optimizing Angular: the view (part 2) **39**

10 Getting connected to data **43**

11 Real-time Presence Made Easy with AngularJS and Firebase **49**

12 Pushy Angular **55**

13 AngularJS SEO, mistakes to avoid. **61**

14 AngularJS with ngAnimate **65**
What directives support animations? 65
CSS3 Transitions & Keyframe Animations 66
JavaScript Animations 69
Triggering Animations inside of our own Directives 71
Learn more! Become a Pro 72

15 HTML5 api: geolocation **73**

16 HTML5 api: camera **79**

17 Good deals – Angular and Groupon **85**

18 Staggering animations with ngAnimate **89**
How can we put this to use? 89
What about CSS3 Keyframe Animations? 91
What directives support this? 93
But what about JS animations? 93
Where can I learn more? 93

19 Getting started unit-testing Angular **95**

20 Build a Real-Time, Collaborative Wishlist with GoAngular v2101
Sign up for GoInstant 103
Include GoInstant & GoAngular 103
Create and Join a Room 104
Fetch our wishes 104
Watch our wishes, so we know when they change 105

21 Immediately satisfying users with a splash page **107**

22 A Few of My Favorite Things: Isolated Expression Scope **111**

23 Conclusion **119**

Chapter 1

Welcome

Congrats on grabbing ng-newsletter.com's mini AngularJS cookbook. In this mini-cookbook, we'll walk through beginner and intermediate recipes to get up and running with Angular quickly without the fluff. As we walk through this book, we encourage you to try out the recipes. We've found that working through them helps both memory and understanding.

We had a lot of fun writing these mini-recipes and we hope you enjoy using them!

Chapter 2

The short guide to service definitions

One of the most misunderstood components of Angular that beginners often ask is about the differences between the service(), factory(), and provide() methods. This is where we'll start the twenty-five days of Angular calendar.

The Service

In Angular, services are singleton objects that are created when necessary and are never cleaned up until the end of the application life-cycle (when the browser is closed). Controllers are destroyed and cleaned up when they are no longer needed.

> This is why we can't dependably use controllers to share data across our application, especially when using routing.Services are designed to be the glue between controllers, the minions of data, the slaves of functionality, the worker-bees of our application.

Let's dive into creating service. Every method that we'll look at has the same method signature as it takes two arguments

- name - the name of the service we're defining
- function - the service definition function.

Each one also creates the *same* underlying object type. After they are instantiated, they all create a service and there is *no functional difference* between the object types.

factory()

Arguably the easiest way to create a service is by using the `factory()` method.

The factory() method allows us to define a service by returning an object that contains service functions and service data. The service definition function is where we place our *injectable* services, such as `$http` and `$q`.

```
angular.module('myApp.services')
.factory('User', function($http) { // injectables go here
  var backendUrl = "http://localhost:3000";
  var service = {
    // our factory definition
    user: {},
    setName: function(newName) {
      service.user['name'] = newName;
    },
    setEmail: function(newEmail) {
      service.user['email'] = newEmail;
    },
    save: function() {
      return $http.post(backendUrl + '/users', {
        user: service.user
      });
    }
  };
  return service;
});
```

Using the factory() in our app

It's easy to use the factory in our application as we can simply *inject* it where we need it at run-time.

```
angular.module('myApp')
.controller('MainController', function($scope, User) {
  $scope.saveUser = User.save;
});
```

When to use the `factory()` method

> The `factory()` method is a great choice to use to build a factory when we just need a collection of methods and data and don't need to do anything especially complex with our service.

We cannot use the `factory()` method when we need to configure our service from the `.config()` function.

service()

The service() method, on the other hand allows us to create a service by defining a *constructor* function. We can use a prototypical object to define our service, instead of a raw javascript object.

Similar to the `factory()` method, we'll also set the *injectables* in the function definition:

```
angular.module('myApp.services')
.service('User', function($http) { // injectables go here
  var self = this; // Save reference
  this.user = {};
  this.backendUrl = "http://localhost:3000";
  this.setName = function(newName) {
    self.user['name'] = newName;
  }
  this.setEmail = function(newEmail) {
    self.user['email'] = newEmail;
  }
  this.save = function() {
    return $http.post(self.backendUrl + '/users', {
      user: self.user
    })
  }
});
```

Functionally equivalent to using the factory() method, the service() method will hold on to the object created by the *constructor* object.

Using the service() in our app

It's easy to use the service in our application as we can simply *inject* it where we need it at run-time.

```
angular.module('myApp')
.controller('MainController', function($scope, User) {
  $scope.saveUser = User.save;
});
```

When to use the `service()` method

> The `service()` method is great for creating services where we need a bit more control over the functionality required by our service. It's also mostly guided by preference to use `this` instead of referring to the service.

We cannot use the `service()` method when we need to configure our service from the `.config()` function.

provide()

The lowest level way to create a service is by using the provide() method. This is the **only** way to create a service that we can configure using the `.config()` function.

Unlike the previous to methods, we'll set the *injectables* in a defined this.$get() function definition.

```
angular.module('myApp.services')
.provider('User', function() {
  this.backendUrl = "http://localhost:3000";
  this.setBackendUrl = function(newUrl) {
    if (url) this.backendUrl = newUrl;
  }
  this.$get = function($http) { // injectables go here
    var self = this;
    var service = {
      user: {},
      setName: function(newName) {
        service.user['name'] = newName;
      },
      setEmail: function(newEmail) {
        service.user['email'] = newEmail;
      },
      save: function() {
        return $http.post(self.backendUrl + '/users', {
          user: service.user
        })
      }
    };
    return service;
  }
});
```

Using the provider() in our app

In order to configure our service, we can *inject* the provider into our .config() function.

```
angular.module('myApp')
.config(function(UserProvider) {
  UserProvider.setBackendUrl("http://myApiBackend.com/api");
})
```

We can use the service in our app just like any other service now:

```
angular.module('myApp')
.controller('MainController', function($scope, User) {
  $scope.saveUser = User.save;
});
```

When to use the `provider()` method

The `provider()` method is required when we want to configure our service before the app runs. For instance, if we need to configure our services to use a different back-end based upon different deployment environments (development, staging, and production).

It's the preferred method for writing services that we intend on distributing open-source as well as it allows us to configure services without needing to hard-code configuration data.

The code for this entire snippet is available here.

Chapter 3

A short guide to routing

Almost all non-trivial, non-demo Single Page App (SPA) require multiple pages. A settings page is different from a dashboard view. The login page is different from an accounts page.

We can get this functionality with Angular incredibly simply and elegantly using the angular.route module.

Installation

In order to actually use routes, we'll need to install the routing module. This is very easy and can be accomplished in a few ways:

Using bower

```
$ bower install --save angular-route
```

Saving the raw source

Alternatively, we can save the source directly from angularjs.org by clicking on the big download button and then clicking on the *extras* link. From there, we can simply save the file to somewhere accessible by our application.

Usage

It's easy to define routes. In our main module, we'll need to *inject* the ngRoute module as a dependency of our app.

```
angular.module('myApp', ['ngRoute'])
  .config(function($routeProvider) {});
```

Now, we can define the routes of our application. The route module adds a $routeProvider that we can *inject* into our .config() function. It presents us two methods that we can use to define our routes.

when()

The when() method defines our routes. It takes two arguments, the *string* of the route that we want to match and a route definition object. The route string will match on the url in the browser. The main route of the app is usually the / route.

The route definition can also accept symbols that will be substituted by angular and inserted into a service called the $routeParams that we can access from our routes.

```
angular.module('myApp', ['ngRoute'])
.config(function($routeProvider) {
  $routeProvider
    .when('/', {
      templateUrl: 'views/main.html',
      controller: 'MainController'
    })
    .when('/day/:id', {
      templateUrl: 'views/day.html',
      controller: 'DayController'
    })
```

The route definition object is where we'll define all of our routes from a high-level. This is where we'll assign a controller to manages the section in the DOM, the templates that we can use, and other route-specific functionality.

Most often, we'll set these routes with a controller and a templateUrl to define the functionality of the route.

otherwise()

The otherwise() method defines the route that our application will use if a route is not found. It's the *default* route.

For example, the route definition for **this** calendar is:

```
angular.module('myApp', ['ngRoute'])
.config(function($routeProvider) {
  $routeProvider
    .when('/', {
      templateUrl: 'views/main.html',
      controller: 'MainController'
    })
    .when('/day/:id', {
      templateUrl: 'views/day.html',
      controller: 'DayController'
    })
    .otherwise({
      redirectTo: '/'
    });
})
```

Using in the view

Okay, so we've defined routes for our application, now how do we actually *use* them in our app?

As a route is simply a new view with new functionality for a portion of a page, we'll need to tell angular which portion of the page we want to switch. To do this, we'll use the ng-view directive:

```
<div class="header">My page</div>
<div ng-view></div>
<span class="footer">A footer</span>
```

Now, anytime that we switch a route, only the DOM element (`<div ng-view></div>`) will be updated and the header/footer will stay the same.

Chapter 4

Preventing flickering

When a particular page or component of our application requires data be available on the page when it loads up, there will be time between when the browser renders the page and angular does. This gap may be tiny so we don't even see the difference or it can be long such that our users see the un-rendered content.

This behavior is called `Flash` `Of` `Unrendered` `Content` (FOUC) and is **always** unwanted. In this snippet, we'll explore a few different ways to prevent this from happening for our users.

ng-cloak

The `ng-cloak` directive is a built-in angular directive that hides all of the elements on the page that contain the directive.

```
<div ng-cloak>
  <h1>Hello {{ name }}</h1>
</div>
```

After the browser is done loading and the *compile phase* of the template is rendering, angular will delete the `ngCloak` element attribute and the element will become visible.

The safest way to make this IE7 safe is to also add a class of `ng-cloak`:

```
<div ng-cloak class="ng-cloak">
  <h1>Hello {{ name }}</h1>
</div>
```

ng-bind

The `ng-bind` directive is another built-in Angular directive that handles data-binding in the view. We can use `ng-bind` instead of using the {{ }} form to bind elements to the page.

Using `ng-bind` instead of {{ }} will prevent the unrendered {{ }} from showing up instead of empty elements being rendered.

The example from above can be rewritten to the following which will prevent the page from flickering with {{ }}:

```
<div>
  <h1>Hello <span ng-bind="name"></span></h1>
</div>
```

resolve

When we're using routes with different pages, we have another option to prevent pages from rendering until some data has been loaded into the route.

We can specify data that we need loaded before the route is done loading by using the `resolve` key in our route options object.

When the data loads successfully, then the route is changed and the view will show up. If it's not loaded successfully, then the route will **not** change and the `$routeChangeError` event will get fired.

```
angular.module('myApp', ['ngRoute'])
.config(function($routeProvider) {
  $routeProvider
  .when('/account', {
    controller: 'AccountController',
    templateUrl: 'views/account.html',
    resolve: {
      // We specify a promise to be resolved
      account: function($q) {
        var d = $q.defer();
        $timeout(function() {
          d.resolve({
            id: 1,
            name: 'Ari Lerner'
          })
        }, 1000);
        return d.promise;
      }
```

```
    }
  })
});
```

The `resolve` option takes an object, by `key/value` where the `key` is the name of the resolve dependency and the `value` is either a string (as a service) or a function whose return value is taken as the dependency.

> `resolve` is very useful when the resolve `value` returns a promise that becomes resolved or rejected.

When the route loads, the `keys` from the resolve parameter are accessible as injectable dependencies:

```
angular.module('myApp')
.controller('AccountController',
  function($scope, account) {
    $scope.account = account;
});
```

We can use the `resolve` key to pass back results from `$http` methods as well, as `$http` returns promises from it's method calls:

```
angular.module('myApp', ['ngRoute'])
.config(function($routeProvider) {
  $routeProvider
  .when('/account', {
    controller: 'AccountController',
    templateUrl: 'views/account.html',
    resolve: {
      account: function($http) {
        return $http.get('http://example.com/account.json')
      }
    }
  })
});
```

The recommended usage of the `resolve` key is to define a service and let the service respond with the required data (plus, this makes testing much easier). To handle this, all we'll need to do is build the service.

First, the `accountService`:

```
angular.module('app')
.factory('accountService', function($http, $q) {
  return {
    getAccount: function() {
      var d = $q.defer();
      $http.get('/account')
      .then(function(response) {
        d.resolve(response.data)
      }, function err(reason) {
        d.reject(reason);
      });
      return d.promise;
    }
  }
})
```

We can use this service method to replace the `$http` call from above:

```
angular.module('myApp', ['ngRoute'])
.config(function($routeProvider) {
  $routeProvider
  .when('/account', {
    controller: 'AccountController',
    templateUrl: 'views/account.html',
    resolve: {
      // We specify a promise to be resolved
      account: function(accountService) {
        return accountService.getAccount()
      }
    }
  })
});
```

Chapter 5

Conditionally applying CSS classes

Switching content based upon state is trivial inside of Angular, but how about styles? For instance, let's say that we have a list of elements, each with a checkbox to indicate to our users that we're selecting an element to run some action on the element or to indicate that a date in a calendar is today (like on the 25 days of Angular calendar front-page).

Angular has a number of built-in ways that we can apply custom styling to DOM elements based upon different data states inside of our `$scope` objects.

ng-style

The `ng-style` directive is a built-in directive that allows us to set styles directly on a DOM element based upon angular expressions.

```
<div ng-controller='DemoController'>
  <h1 ng-style="{color:'blue'}">
    Hello {{ name }}!
  </h1>
</div>
```

The `ng-style` directive accepts an object or a function that returns an object whose keys are CSS style names (such as `color`, `font-size`, etc) and the values that correspond to the style key names.

In the above example, we're *hard-coding* the style color, but we can easily change that to either pass in a function that calculates and returns back the color or we can pass in an object that contains the specific values for the style.

For example, we can use a dropdown to set the specific style on the element and update a model with the selected color on the dropdown:

```
<div ng-controller='DemoController'>
  <select ng-model="selectedColor"
    ng-options="color for color in allColors">
  </select>
  <h1 ng-style="{color:selectedColor}">
    Hello {{ name }}!
  </h1>
</div>
```

The `DemoController` controller in this case looks like:

```
angular.module('adventApp')
.controller('DemoController', function($scope) {
  $scope.allColors = ['blue', 'red', '#abc', '#bedded'];
  $scope.selectedColor = 'blue';
});
```

When we change the `selectedColor`, the `ng-style` directive updates and our HTML changes.

ng-class

The `ng-class` directive is available for us to apply different css classes to a DOM object, instead of styles. Allowing us to set classes is often more powerful than setting styles directly as we can define our CSS design separate from our business logic of the app.

```
<div ng-controller='DemoController'>
  <button ng-click="shouldHighlight=!shouldHighlight">
    Highlight text
  </button>
  <h1 ng-class="{highlight:shouldHighlight}">
    Hello {{ name }}!
  </h1>
</div>
```

Functionally, it is similar to the `ng-style` directive as it accepts an object or a function that returns an object to define the classes that are to be applied to the DOM object.

As we can pass an object, we can set multiple conditionals on the element. This flexibility enables us to define custom design for our elements.

```
<div ng-controller='DemoController'>
  <button ng-click="shouldHighlight=!shouldHighlight">
    Highlight text
  </button>
  <button ng-click="emphasize=!emphasize">
    Emphasize
  </button>
  <h1 ng-class="{
    important:emphasize,
    highlight:shouldHighlight
    }">
    Hello {{ name }}!
  </h1>
</div>
```

custom directive

Another solution we can use to create custom styles for our elements is by setting up a directive to generate a custom CSS stylesheet for us.

> When we need to set a large number of styles or we're setting a lot of styles for a particular custom page, we can use the custom directive is an efficient method. For instance, if we're building a site that we allow users to customize a site of theirs.

A custom directive might look something like:

```
angular.module('myApp')
.directive('myStylesheet', function() {
  return {
    restrict: 'A',
    require: 'ngModel',
    scope: { ngModel: '=', className: '@' },
    template: "<style " +
      "type='text/stylesheet'>" +
    ".{{ className }} {" +
    " font-size: {{ ngModel.fontsize }};" +
    " color: {{ ngModel.color }};" +
    "}" +
    "</style>"
  }
});
```

This directive will add a `<style>` tag to our page and allow us to pass in a single `ng-model` that can contain our custom styles.

We can use this directive simply by attaching our data model that contains our DOM element's style on the tag.

```
<div ng-controller='DemoController'>
  <div my-stylesheet
    ng-model="pageObject"
    class-name="customPage" >
  </div>
  <div class="customPage">
    <h1>Hello</h1>
  </div>
</div>
```

Chapter 6

Setting configs in angular

Like with any modern software development lifecycle that's destined for production, we're likely to have multiple deployment destinations. We might have a development server, a staging server, a production server, etc.

For each of these different scenarios, we're probably using different versions of the app, with the latest, bleeding edge version on development and staging and the stable version in production.

Each of these likely is talking to different back-end services, is using different individual settings, such as which back-end url we're talking to, if we're in testing mode or not, etc.

Here are a few of the multiple methods for how we can handle separating out our environments based upon deployment type. But first…

Configuration in Angular

The easiest way to use configuration inside of an Angular app is to set a constant with an object. For example:

```
angular.module('myApp.config')
.constant('myConfig', {
  'backend': 'http://localhost:3000/api',
  'version': 0.2
})
```

Our Angular objects now have access to this `myConfig` as a service and can *inject* it just like any other Angular service to get access to the configuration variables:

```
angular.module('myApp', ['myApp.config'])
.factory('AccountSrv',
function($http, myConfig) {
  return {
    getAccount: function() {
      return $http({
        method: 'GET',
        url: myConfig.backend + '/account'
      })
      // ...
    }
  }
});
```

> In this example, we separate out the main app module from the one that contains the configuration and then later set it as a dependency of our main app. We'll follow this pattern for all examples in this snippet.

We'll use this same pattern in the few different methods to get config data in our app.

Server-side rendering

If we're using a server-side rendering engine, such as Ruby on Rails or NodeJS to deliver our application files, we can render out different JavaScript in/for the view.

When we're using Rails, this process is pretty simple (we'll leave it as an exercise for the reader to implement in different back-end solutions). Presumably our configuration is set up using a hash object that looks similar to:

```
CONFIG = {
  development: {
    backend: "http://localhost:3000"
  },
  production: {
    backend: "http://my.apibackend.com"
  }
}
```

We can use this inside of our `index.html.haml` file to deliver rendered JavaScript to our browser.

```
<!-- Our app content -->
div{'ng-view' => ""}
:javascript
  angular.module('myApp.config', [])
  = .constant('myConfig', #{CONFIG[Rails.env]})
%script{:src => "scripts/app.js"}
```

As with any separate module in Angular, we'll add it as a dependency of our main app and we're good to go. We'll write our app code to include the `myApp.config` as a dependency and inject the configuration where we need it:

```
angular.module('myApp', ['myApp.config'])
.service('AccountService',
  function($http, myConfig) {
    this.getAccount = function() {
      return $http({
        method: 'GET',
        url: myConfig.backend+'/account.json'
      });
    }
});
```

Client-side

If we are not using a back-end to serve our app and cannot depend upon html interpolation to render us back a configuration, we must depend upon configuration at compile-time.

As we can see above, we can manually set our configuration files if we're not depending upon any complex configuration or deploying only to one or two environments.

We will generate a file for each environment that contains the relevant configuration for each environment. For instance, the development might look like:

```
angular.module('myApp.development', [])
.constant('myConfig', {
  backend: 'http://dev.myapp.com:3000'
})
```

And the production might look like:

```
angular.module('myApp.production', [])
.constant('myConfig', {
```

```
  backend: 'http://myapp.com'
})
```

Now, inside of our app, we'll only need to switch a single line to define which configuration it should use in production. As we said before, we'll need to add one of these to our dependencies of our app module:

```
angular.module('myApp', ['myApp.production'])
// ...
```

The tricky part of this is that we'll need to manually change the configuration file for each build. Although it's not a difficult task for a single developer, this can become a giant hassle for larger teams.

We can use our build tools instead of doing this configuration switch manually.

Using grunt to automate the process

Using Grunt, we can use the `grunt-template` task. This task allows us to specify variables once and output a template of our choosing.

> If you're using yeoman to build your angular apps (and you should be), this should look very familiar.

Installation

Installing `grunt-template` is easy to do with npm:

```
$ npm install --save-dev grunt-template
```

Since Gruntfiles are simply JavaScript, we can write our configuration in `.json` files in our path and load them from our `Gruntfile`. Since we're interested in using different environments, we'll set up our configuration files in a config directory.

For instance, the development config file in `config/development.json`:

```
{
  backend: 'http://dev.myapi.com:3000'
}
```

And the production version: `config/production.json`:

```
{
  backend: 'http://myapp.com'
}
```

With these set, we'll need to tackle two other tasks:

1. Loading the configuration for the current environment
2. Compiling the config into a template

To load the configuration for the current environment, we'll simply use a bit of JavaScript in our `Gruntfile` to take care of this process. At the top of the `Gruntfile` we'll load the following to load the *right* config into a variable we'll call `config`:

```
var env = process.env.ENV || 'development';

var config = require('./config/' + env + '.json');
```

This will set the configuration to load the object from `config/development.json` by default. We can pass the environment variable `ENV` to set it to production:

```
$ ENV=production grunt build
```

Now all we need to do is compile our template. To do this, we'll use the `grunt-template` task:

```
grunt.initConfig({
  // ...

  'template': {
    'config': {
      'options': {
        data: config
      }
    },
    'files': {
      // The .tmp destination is a yeoman
      // default location. Set this dest
      // to fit your own needs if not using
      // yeoman
      '.tmp/scripts/config.js':
        ['src/config.js.tpl']
    }
```

```
  }
  // ...
});

grunt.loadNpmTasks('grunt-template');
grunt.registerTask('default', ['template']);
```

We'll need to write our `src/config.js` template. This is the easy part as we'll simply generate the file we manually created above. In our `src/config.js.tpl`, add the following:

```
angular.module('myApp.config', [])
.constant('myConfig', function() {
  backend: '<%= backend %>'
});
```

With this automated process, we can eliminate the manual process of building our configuration entirely.

Chapter 7

The Power of Expression Binding

10 out of 10 developers agree that they love data-binding in Angular. This glowing endorsement is well earned as it has saved us from having to write truck loads of boilerplate code while increasing the testable surface area of our application exponentially.

In this recipe, we look at some *magic* we can do with Angular.

New developers to Angular tend to follow a learning curve when it comes to data-binding that usually goes in this order:

- primitives
- data-structures
- expressions

Binding to expressions is an incredibly powerful technique that opens the door to some really interesting possibilities.

In this article, we are walking through an example where we are going to move from simple data-binding to a fairly complex and interesting expression binding example.

> Disclaimer: Binding to expressions are really powerful but if not used delicately, we'll end up paying a performance cost. The example, we're talking a about a prototype-ready example that is not appropriate with 1000s of items or logic-heavy expressions.

Simple Bindings

We are going to kick things off with a simple controller called `StepOneController` that essentially has two properties and a method.

We will bind to `$scope.categories` array and then call `setCurrentCategory` to update the `currentCategory` property from the view.

```
angular.module('myApp')
.controller('StepOneController', ['$scope', function ($scope) {
  // Example categories
  $scope.categories = [
      {name: 'one', display: 'Category One'},
      {name: 'two', display: 'Category Two'},
      {name: 'three', display: 'Category Three'},
      {name: 'four', display: 'Category Four'}
  ];

  $scope.currentCategory = null;

  $scope.setCurrentCategory = function (category) {
    $scope.currentCategory = category;
  };
}])
```

In the view, we are looping over the `categories` array with `ng-repeat` and create a `category-item` div for each object in the `categories` array.

Within the div we are binding the `display` property on the `category` object.

```
<div class="container" ng-controller="StepOneController">
  <h2>Step One</h2>
  <div class="row">
    <div class="col-xs-3 category-item"
      ng-repeat="category in categories"
      ng-click="setCurrentCategory(category)">
      <strong>{{category.display}}</strong>
    </div>
  </div>
  <h4>Active Category: {{currentCategory.display}}</h4>
</div>
```

When our user clicks on the `category-item div`, the `ng-click` directive calls `setCurrentCategory()` with the `category` object along with it.

In the `h4` tag below, we're simply binding the `currentCategory.display` data above as we would in simply data-binding.

A Basic Expression Binding

Albeit simple, the above example is impressive in its own right with how much we can accomplish in a little amount of code with data-binding.

Now that the foundation has been laid and now we are going to extend it to allow us to bind to an expression with Angular.

In the demo code, we have create second controller called `StepTwoController` that is identical to `StepOneController` with an additional method `isCurrentCategory()` that returns a boolean if the current category matches the category argument.

```
angular.module('myApp')
.controller('StepTwoController', function ($scope) {
  $scope.categories = [
    {name: 'one', display: 'Category One'},
    {name: 'two', display: 'Category Two'},
    {name: 'three', display: 'Category Three'},
    {name: 'four', display: 'Category Four'}
  ];

  $scope.currentCategory = null;

  $scope.setCurrentCategory = function (category) {
    $scope.currentCategory = category;
  };

  $scope.isCurrentCategory = function (category) {
    return $scope.currentCategory === category;
  }
})
```

> Generally speaking this about as complicated as we like bound expressions to be with simple evaluations that incur very little overhead.

We are going to use this method to dynamically set a class on our `category-item` div using the `ng-class` directive.

Using the `ng-class` directive, we'll dynamically apply a class based upon the result of an expression. In this case, we're applying the `current-active` class to the DOM element if the result of the `isCurrentCategory()` method is truthy:

```
<div class="container" ng-controller="StepTwoController">
  <div class="col-xs-3 category-item"
```

```
    ng-repeat="category in categories"
    ng-click="setCurrentCategory(category)"
    ng-class="{'current-active':isCurrentCategory(category)}">
    <strong>{{category.display}}</strong>
  </div>
</div>
```

In our CSS, the `current-active` class applies a 10 pixel border to the element it is applied to. Practically speaking, when a user clicks a category, there will be a grey border around it indicating that it is the active element.

```
.current-active {
  border: 10px solid #666;
}
```

We have made just a few small additions to our code to dynamically update the UI based on the value of a simple expression.

Expressions All The Way Down

The final example where we get into some really powerful stuff using expressions within expressions.

> We are going to *dynamically* set a *dynamically-defined* class based on the value of a *dynamically-defined* variable.

The first thing we need to do to make this work is to create a class that corresponds to each category object we have in our categories array.

The idea is that *.current-one* applies to the *category* that has the name `one` and `.current-two` corresponds to the `category` that has the name `two`, etc.

```
.current-one {
  border: 10px solid #457b97;
}

.current-two {
  border: 10px solid #727372;
}

.current-three {
  border: 10px solid #a66a48;
}
```

```
.current-four {
  border: 10px solid #f60;
}
```

> The only difference between these four classes is the color of the border defined in the above CSS.

We have declared a `StepThreeController` that is the exact same as `StepTwoController` and the HTML between the two examples are the same except for one new addition of the class variable we are dynamically setting in the view.

```
<div ng-controller='StepThreeController'>
  <div
    class="col-xs-3 category-item btn btn-primary"
    ng-repeat="category in categories"
    ng-click="setCurrentCategory(category)"
    ng-class="
      {'current-{{category.name}}':
        isCurrentCategory(category)
    }">
    <strong>{{category.display}}</strong>
  </div>
  <div class="clear"></div>
</div>
```

This works is that Angular knows to evaluate the bindings before evaluating the `ng-class` expression.

Binding to expressions is an incredibly powerful technique that allows you to accomplish some pretty impressive things when done responsibly.

The code for this article is available **here**

Chapter 8

Optimizing Angular: the view (part 1)

It's incredibly easy for us to build prototypes of angular apps, but what about when we head to production?

Will our app hold up? What about when we start dealing with large amounts of data? How can we make sure our application is still performant?

In this part 1 of optimizing angular, we'll walk through a few ways to optimize our view and why they are important.

Limiting the filter

Filters are incredibly powerful ways to sort and manage our way through data in the view. They can help us format data, live-search data, etc.

```
<table ng-controller="FilterController">
  <thead>
    <tr>
      <th>Filter</th>
      <th>Input</th>
      <th>Output</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>uppercase</td>
      <td ng-non-bindable>{{ msg | uppercase }}</td>
      <td>{{ msg | uppercase }}</td>
```

```
    </tr>
    <tr>
      <td>currency</td>
      <td ng-non-bindable>{{ 123.45 | currency }}</td>
      <td>{{ 123.45 | currency }}</td>
    </tr>
    <tr>
      <td>date</td>
      <td ng-non-bindable>{{ today | date:'longDate' }}</td>
      <td>{{ today | date:'longDate' }}</td>
    </tr>
    <tr>
      <td>custom pig-latin filter</td>
      <td ng-non-bindable>{{ msg | piglatin }}</td>
      <td>{{ msg | piglatin }}</td>
    </tr>
  </tbody>
</table>
```

The `piglatin` filter is a custom filter that looks like this:

```
angular.module('myApp')
.filter('piglatin', function() {
  return function(text) {
    text = text || '';
    text = text
      .replace(/\b([aeiou][a-z]*)\b/gi, "$1way");
    text = text
      .replace(/\b([bcdfghjklmnpqrstvwxy]+)([a-z]*)\b/gi, "$2$1ay");
    return text;
  }
});
```

(jsbin example)

Although this is really incredibly useful to be able to do when we're prototyping, it's a cause for a lot of slow-down in the view. Every time that there is a `$digest` event, these filters will run for **every** value. That's exponentially worse when we're using `ng-repeat` and setting a filter on those values.

How do we get away with moving this functionality out of the view so it runs once, rather than every single time? Use the controller!

We can change the view above to hand us the filtered value so we don't need to do any parsing in the view ourselves.

Using filters in the controller

To get access to a filter inside of a controller, we'll need to *inject* it into the controller. For instance:

```
angular.module('myApp')
.controller('HomeController', function($scope, $filter) {
  // We now have access to the $filter service
});
```

With the `$filter` service, we can fetch the filter we're interested in and apply it to our data.

The `date` example from above is a good candidate to provide a formatted date in the controller. We might hand back a `formatted_date` instead of using the raw date in the view:

```
angular.module('myApp')
.controller('HomeController', function($scope, $filter) {
  $scope.today = new Date();
  var dateFilter = $filter('date');
  $scope.formatted_today = dateFilter($scope.today, 'mediumDate');
})
```

Instead of showing the `today` scope property in our view, we can simply show the `formatted_today` value and have the filter run in the background.

Chapter 9

Optimizing Angular: the view (part 2)

In this recipe, we're going to continue optimizing the view with Angular.

It's relatively well-known that the Angular event system can handle approximately 2000 watches on any given page for a desktop browser. Although this number has been somewhat arbitrarily cited by the Angular team and others, it holds true with our experience as well.

What's a watch

Before we can talk about how to optimize the numbers of watches in the view, it's a good idea to define what we mean when we say *watch*.

Angular sets up these constructs called *watches* which monitor the changes to interesting data in an angular application. This *interesting data* is specified by our app by showing elements on-screen using directives or the `$scope` from inside of a controller using the `{{ }}` template tags.

The following creates two watches, one for the input and one for the output:

```
<div>
  Enter your name: <input type="text" ng-model="name" />
  <h3>Hello {{ name }}</h3>
</div>
```

For instance, anytime that we set up a list that uses `ng-repeat` with a bunch of components that may or may not change. If we set up a binding between the front-end and the backend, we create at minimum 1 watch. If we loop over a

list while we create that, that's at least `n` watches. When we start setting up multiple bindings for each iteration, that's n*n watches.

Limiting the watches

There are a few strategies that we can take to limit the number of watches on a page. For this snippet, we'll talk about the `bindonce` library.

Whenever we're using static information, information that we don't expect to change, it's inefficient to keep a watcher around to watch data that will never change.

For instance, when we get a list of our user's Facebook friends we don't expect their names to change. We'll look at this data as *static* information.

Rather than creating a watch for every one of our facebook friend's names, we can tell Angular to only create a single watch to watch the list of friends to repeat over using the `bindonce` library.

This code creates a 3 watchers for every friend of ours. In a list of 20 friends, that's 60 watchers in this code alone:

```
<ul>
  <li ng-repeat="friend in friends">
    <a ng-href="friend.link">
      <h1 ng-bind="friend.name">
      <h4 ng-bind="friend.birthday">
      </h4>
    </a>
  </li>
</ul>
```

Installation

To install the `bindonce` library, we can use `bower` to download it:

```
$ bower install angular-bindonce --save
```

Or we can grab it from the repo at https://github.com/Pasvaz/bindonce. Either way, we'll need to include the file in our HTML and reference it as a module in our app's dependency module array.

```
angular.module('app', [
  'pasvaz.bindonce'
])
```

Using `bindonce`, we can tell angular to watch data while it's unresolved and as soon as it resolves we can set it to remove the watcher. The following code only resolves 1 watcher for the entire loop:

```
<ul>
  <li bindonce
    ng-repeat="friend in friends">
    <a bo-href="friend.link">
      <h1 bo-text="friend.name">
      <h4 bo-text="friend.birthday">
      </h4>
    </a>
  </li>
</ul>
```

Use simple watches

Another approach to optimize the watches on our view is to keep them as simple as possible. The `$digest` cycle will run watches at a minimum of 2 times per cycle, so the simpler the `$watch` function is, the faster it will run.

For deeper look into other optimizations, check out the optimization chapter in ng-book.com.

Chapter 10

Getting connected to data

Webapps are only as interesting as the functionality and data that they provide us. Other than isolated apps such as calculators and solitaire games, data powers most functionality.

In this snippet, we're going to look at the *Angular Way* of connecting to data sources. Specifically, we're going to work with the flickr API to get public photos. Let's get started!

Naively getting data

Angular provides us the `$http` service by default. The `$http` service is an interface to the native `XMLHttpRequestObject` provided by the browser that works directly with our angular apps.

To use the `$http` object, we'll first need to *inject* it into our angular objects, like any other service we'll want to use:

```
angular.module('myApp')
.controller('HomeController', function($scope, $http) {
  // We now have access to the $http object
});
```

We're going to use the `$http` object to fetch photos from the public flickr API:

```
<small>First random photo in the public data stream</small>
<div ng-controller="FlickrApiController">
  <a href="{{ photo.link }}">
    <img ng-src="{{ photo.media.m }}"
        alt="{{ photo.title }}" />
```

```
  </a>
</div>
```

We are fetching this photo from the flickr API using the `$http` service in our controller using:

```
angular.module('myApp')
.controller('FlickrApiController', function($scope, $http) {
  $http({
    method: 'JSONP',
    url: 'http://api.flickr.com/services/feeds/photos_public.gne',
    params: {
      'format': 'json',
      'jsoncallback': 'JSON_CALLBACK'
    }
  }).success(function(photos) {
    $scope.photo = photos.items[0];
  })
})
```

Now, although this *works*, it's **not** the Angular Way of fetching photos. Using the `$http` service in our controller makes it incredibly difficult to test controllers and our functional logic for fetching from our external services.

The angular way

Rather than using our controllers to get our photo, we'll create a *service* that will house our *flickr* interactions.

A service is a singleton object that we can both bundle our common services together as well as use to share data around our application. In this case, we'll create a `Flickr` service that bundles together our flickr functionality.

As the *Flickr* API requires an API key for any non-public requests, we'll likely want to create a `provider` as we'll want to configure our api key on a per-app basis.

```
angular.module('adventApp')
.provider('Flickr', function() {
  var base = 'http://api.flickr.com/services',
      api_key = '';

  // Set our API key from the .config section
  // of our app
  this.setApiKey = function(key) {
```

```
    api_key = key || api_key;
  }

  // Service interface
  this.$get = function($http) {
    var service = {
      // Define our service API here
    };

    return service;
  }
})
```

This gives us the ability to set our `api_key` from Flickr in our `.config()` function on our app so we can use it later in our calls to the flickr api:

```
angular.module('myApp')
.config(function(FlickrProvider) {
  FlickrProvider.setApiKey('xxxxxxxxxxxxxxxxxxxxxxxx')
})
```

Instead of using the `$http` service in our controller, we can take the entire functionality from above and copy+paste it into our service and return the promise.

```
// ...
// Service interface
this.$get = function($http) {
  var service = {
    // Define our service API here
    getPublicFeed: function() {
      return $http({
        method: 'JSONP',
        url: base + '/feeds/photos_public.gne?format=json',
        params: {
          'api_key': api_key,
          'jsoncallback': 'JSON_CALLBACK'
        }
      });
    }
  };

  return service;
}
// ...
```

Our `Flickr` service now has a single method: `getPublicFeed()` that we can use in our controller. This changes our entire call to the flickr api to look like:

```
angular.module('myApp')
.controller('FlickrApiController', function($scope, Flickr) {
  Flickr.getPublicFeed()
  .then(function(data) {
    $scope.newPhoto = data.data.items[0];
  });
})
```

One final component that we like to change here is how the `data.data.items[0]` looks in the usage of our `getPublicFeed()` function. This is particularly unappealing and requires users of our service to know exactly what's going on behind the scenes.

We like to clean this up in our service by creating a *custom* promise instead of returning back the `$http` promise. To do that, we'll use the `$q` service directly to create a promise and resolve that directly. We'll change our `getPublicFeed()` api method to look like:

```
// ...
this.$get = function($q, $http) {
  var service = {
    getPublicFeed: function() {
      var d = $q.defer();
      $http({
        method: 'JSONP',
        url: base + '/feeds/photos_public.gne?format=json',
        params: {
          'api_key': api_key,
          'jsoncallback': 'JSON_CALLBACK'
        }
      }).success(function(data) {
        d.resolve(data);
      }).error(function(reason) {
        d.reject(reason);
      })
      return d.promise;
    }
  };

  return service;
}
// ...
```

Now we can call the `getPublicFeed()` without needing to use the extra `.data`:

```
Flickr.getPublicFeed()
.then(function(photos) {
  $scope.newPhoto = photos.items[0];
});
```

The full source of this example can be found on jsbin.

Chapter 11

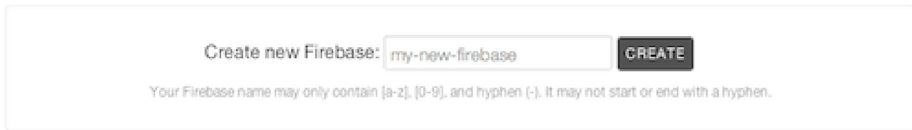
Real-time Presence Made Easy with AngularJS and Firebase

It is mind-boggling what can be accomplished these days within a modern web stack. Looking back only a few years ago, implementing a real-time tracker for counting the number of users on our pages in less than 50 lines of code seemed impossible. Today, we can easily do this in a matter of minutes in only a few lines of code.

In this snippet, we are going to build this **real-time presence system** using AngularJS and Firebase and we'll do it in only a matter of minutes. Let's buckle up and get started!

Step One: Create a Firebase

The first thing we need to do is go to Firebase and create a free account or login if we already have an account. Head to firebase.com



Once logged in, we can head to account overview and create a Firebase. Here, we'll create a unique name that's associated with our account and click **create**.

The name that we create our Firebase with indicates how we will fetch it later. For instance, if we named it `ng-advent-realtime`, then the URL to retrieve

it from is `https://ng-advent-realtime.firebaseio.com/`. We'll hold on to this for a minute…

Step Two: Setting up our app

Now that we have a Firebase set up, we need need to add the Firebase library to our project.

We just need to add the script tag below to our `index.html` page.

```
<script src="https://cdn.firebase.com/v0/firebase.js"></script>
<script
  src='https://cdn.firebase.com/libs/angularfire/0.6.0/angularfire.min.js'>
</script>
```

We are also kicking things off by bootstrapping our Angular app that we'll name *myRealtimeAdvent* by setting the `ng-app` directive in our DOM:

```
<html ng-app="myRealtimeAdvent">
```

We'll also set the `MainController` (that we'll create shortly) to drive the view:

```
<body ng-controller="MainController">
```

Now, in the `script.js` file, we'll create the barebone version of our app, so that our `index.html` page can actually load:

```
angular.module('myRealtimeAdvent', [])
  .factory('PresenceService', function() {
    // Define our presence service here
  })
  .controller('MainController', ['$scope',
    function($scope, PresenceService) {
      // Controller goes here
    }
  ])
```

Now we have an AngularJS application that has real-time Firebase capabilities waiting in the wings to be utilized.

Step Three: The Real-time Service Minus Real-time

We want to isolate the real-time presence functionality into its own service and so we are going to create the `PresenceService` for the rest of the application to use as we did above.

For the time being, we are going to declare the `PresenceService` and give it just enough functionality to have the appearance of being useful.

We have created a property `onlineUsers` and we are exposing this property via the `getOnlineUserCount()` method.

```
app.factory('PresenceService', ['$rootScope',
  function($rootScope) {
    var onlineUsers = 0;

    var getOnlineUserCount = function() {
      return onlineUsers;
    }

    return {
      getOnlineUserCount: getOnlineUserCount
    }
  }
]);
```

The `PresenceService` API has only the one method of `getOnlineUserCount()`, but that's all we'll need for now.

Step Four: A Real Real-time Service

We are going to be doing three distinct things with our Firebase and they are as follows:

1. Create a main Firebase reference keeps track of how many connected users are on the page. We are also going to create a child Firebase reference that uniquely identifies us as a user.
2. Create a presence reference that points to a special Firebase location (at .info/connected) that monitors a client connection state. We will use this reference to modify the state of our other references which will propagate to all of the connected clients.
3. Listen to the main Firebase reference for changes in the user count so that we can update the rest of the AngularJS application.

   We are not going to delve into every part of the Firebase API as it exists in this code but we strongly encourage checking out the docs for further reading.

Create Our References

Create a Firebase reference is a matter of creating a new Firebase object and sending in the location that you want to target.

We are creating a variable we'll call `listRef` that will serve as our main Firebase reference. We will also creating child reference called `userRef` that we will use to uniquely identify each user.

Additionally, we will create a `presenceRef` that will handle all the legwork for letting Firebase know when users come and go:

```
var baseUrl = 'https://ng-advent-realtime.firebaseio.com'
var listRef = new Firebase(baseUrl + '/presence/');
// This creates a unique reference for each user
var userRef = listRef.push();
var presenceRef = new Firebase(baseUrl + '/.info/connected');
```

Announcing new users

Once the `presenceRef` is connected, we set the `userRef` to `true` on the page to add ourselves to the `listRef`.

We will also tell Firebase to remove us when we disconnect using the function on UserRef `userRef.onDisconnect().remove()`:

```
// Add ourselves to presence list when online.
presenceRef.on('value', function(snap) {
  if (snap.val()) {
    userRef.set(true);
    // Remove ourselves when we disconnect.
    userRef.onDisconnect().remove();
  }
});
```

Announce Everyone Else

Now that we have added ourselves to the list, we need to listen for when other users are added to the same location.

We can do this by listening for the `value` and then updating `onlineUsers` to `snap.numChildren` and then broadcasting an `onOnlineUser` event to all interested parties.

```
// Get the user count and notify the application
listRef.on('value', function(snap) {
```

```
  onlineUsers = snap.numChildren();
  $rootScope.$broadcast('onOnlineUser');
});
```

The complete `PresenceService` looks like:

```
.factory('PresenceService', ['$rootScope',
  function($rootScope) {
    var onlineUsers = 0;

    // Create our references
    var baseUrl = 'https://ng-advent-realtime.firebaseio.com'
    var listRef = new Firebase(baseUrl + '/presence/');
    // This creates a unique reference for each user
    var userRef = listRef.push();
    var presenceRef = new Firebase(baseUrl + '/.info/connected');

    // Add ourselves to presence list when online.
    presenceRef.on('value', function(snap) {
      if (snap.val()) {
        userRef.set(true);
        // Remove ourselves when we disconnect.
        userRef.onDisconnect().remove();
      }
    });

    // Get the user count and notify the application
    listRef.on('value', function(snap) {
      onlineUsers = snap.numChildren();
      $rootScope.$broadcast('onOnlineUser');
    });

    var getOnlineUserCount = function() {
      return onlineUsers;
    }

    return {
      getOnlineUserCount: getOnlineUserCount
    }
  }
]);
```

Our new `PresenceService` is complete and ready to be used within our application.

Step Four: It's Alive!

And at this point, it we can simply listen for the `onOnlineUser` event on `$scope` and update `$scope.totalViewers` to the value returned by the function on the PresenceService called: `PresenceService.getOnlineUserCount()`.

```
.controller('MainController', ['$scope', 'PresenceService',
  function($scope, PresenceService) {
    $scope.totalViewers = 0;

    $scope.$on('onOnlineUser', function() {
      $scope.$apply(function() {
        $scope.totalViewers =
          PresenceService.getOnlineUserCount();
      });
    });
  }
])
```

It is important to point out that because `onOnlineUser` is being broadcasted as a result of an event outside of the AngularJS universe, we need to use `$scope.$apply` to kick of a digest cycle within the `MainController` otherwise we won't see any updates in our app.

> Pro Tip: If you are binding to something that is not updating, it is possible that AngularJS is not able to detect that the value has changed. The first thing that we do is wrap the expression in question in an `$apply` method call. It's also preferable to use the built-in AngularJS services where possible i.e. `$timeout` vs timeout.

And now we can bind to `totalViewers` in our HTML and complete the circuit.

```
<h1>{{totalViewers}} viewers are viewing</h1>
```

The entire code for this article is available **here**.

This article was written by Lukas Ruebbelke and Ari Lerner.

Chapter 12

Pushy Angular

Real-time with Angular is a topic that's growing to be an increasingly important in today's fast-moving pace. We have already looked at how to handle real-time presence with Firebase.

Pusher is especially good for generating real-time data that don't necessarily need custom storage. In this snippet, we're going to build a small dashboard for a server running a tiny stats collection process that runs every 10 seconds.

Get pushy

In order to work with the Pusher service, we'll need to sign up for it (obviously). Head to Pusher and sign up for the service. We'll be working with the free account.

Once we've signed up, we'll need to include loading the library in our HTML. Now, we can do this in the *usual* way by placing a script tag on the page:

```
<script src="http://js.pusher.com/2.1/pusher.min.js"
  type="text/javascript"></script>
```

Or we can create a provider to load the library for us. This has many advantages, the most of which is that it allows us to use Angular's dependency injection with externally loaded scripts.

Although we won't walk through the source here line-by-line, we've included comments at the relevant parts. The following snippet is simply dynamically loading the library on the page for us.

```
angular.module('alPusher', [])
.provider('PusherService', function() {
```

```
var _scriptUrl = '//js.pusher.com/2.1/pusher.min.js'
, _scriptId = 'pusher-sdk'
, _token = ''
, _initOptions = {};

this.setOptions = function(opts) {
  _initOptions = opts || _initOptions;
  return this;
}

this.setToken = function(token) {
  _token = token || _token;
  return this;
}

// Create a script tag with moment as the source
// and call our onScriptLoad callback when it
// has been loaded
function createScript($document, callback, success) {
  var scriptTag = $document.createElement('script');
  scriptTag.type = 'text/javascript';
  scriptTag.async = true;
  scriptTag.id = _scriptId;
  scriptTag.src = _scriptUrl;
  scriptTag.onreadystatechange = function () {
    if (this.readyState == 'complete') {
      callback();
    }
  }
  // Set the callback to be run
  // after the scriptTag has loaded
  scriptTag.onload = callback;
  // Attach the script tag to the document body
  var s = $document
    .getElementsByTagName('body')[0];
  s.appendChild(scriptTag);
}

// Define the service part of our provider
this.$get = ['$document', '$timeout', '$q', '$rootScope', '$window',
  function($document, $timeout, $q, $rootScope, $window) {
  var  deferred = $q.defer(),
      socket,
      _pusher;

  function onSuccess() {
```

```
      // Executed when the SDK is loaded
      _pusher = new $window.Pusher(_token, _initOptions);
    }

      // Load client in the browser
      // which will get called after the script
      // tag has been loaded
    var onScriptLoad = function(callback) {
      onSuccess();
      $timeout(function() {
        // Resolve the deferred promise
        // as the FB object on the window
        deferred.resolve(_pusher);
      });
    };

    // Kick it off and get Pushing
    createScript($document[0], onScriptLoad);
    return deferred.promise;
   }]
})
```

> This is our preferred method of injecting external libraries as it also makes it incredibly simple to test our external library interactions.

With the `PusherService` above, we can create a secondary service that will actually handle subscribing to the Pusher events.

We'll create a single method API for the `Pusher` service that will subscribe us to the Pusher channel.

```
angular.module('myApp', ['alPusher'])
.factory('Pusher', function($rootScope, PusherService) {
  return {
    subscribe: function(channel, eventName, cb) {
      PusherService.then(function(pusher) {
        pusher.subscribe(channel)
        .bind(eventName, function(data) {
          if (cb) cb(data);
          $rootScope
            .$broadcast(channel + ':' + eventName, data);
          $rootScope.$digest();
        })
      })
    }
```

```
  }
})
```

This `Pusher` service allows us to subscribe to a channel and listen for an event. When it receives one, it will `$broadcast` the event from the `$rootScope`. If we pass in a callback function, then we'll run the callback function.

For example:

```
Pusher.subscribe('stats', 'stats', function(data) {
  // from the stats channel with a stats event
});
```

Triggering events

Although it is possible to trigger events from the client, pusher discourages this usage as it's insecure and we must be careful to accept all events as client-side events cannot always be trusted. To allow the client application to trigger events, make sure to enable it in the applications settings in the browser.

We can then create a `trigger` function in our factory above:

```
angular.module('myApp', ['alPusher'])
.factory('Pusher', function($rootScope, PusherService) {
  return {
    trigger: function(channel, eventName, data) {
      channel.trigger(eventName, data);
    }
  }
})
```

Tracking nodes

We'll need to keep track of different nodes with our dashboard. Since we're good angular developers and we write tests, we'll store our nodes and their active details in a factory.

There is nothing magical about the `NodeFactory` and it's pretty simple. It's entire responsibility is to hold on to a list of nodes and their current stats:

```
angular.module('myApp')
.factory('NodeFactory', function($rootScope) {
  var service = {
    // Keep track of the current nodes
```

```
    nodes: {},
    // Add a node with some default data
    // in it if it needs to be added
    addNode: function(node) {
      if (!service.nodes[node]) {
        service.nodes[node] = {
          load5: 0,
          freemem: 0
        };
        // Broadcast the node:added event
        $rootScope.$broadcast('node:added');
      }
    },
    // Add a stat for a specific node
    // on a specific stat
    addStat: function(node, statName, stat) {
      service.addNode(node);
      service.nodes[node][statName] = stat;
    }
  }
  return service;
})
```

Tracking

We're almost ready to track our server stats now. All we have left to do is configure our `Pusher` service with our API key and set up our controller to manage the stats.

We need to configure the `PusherService` in our `.config()` function, like normal:

```
angular.module('myApp')
.config(function(PusherServiceProvider) {
  PusherServiceProvider
    .setToken('xxxxxxxxxxxxxxxxxxxx')
    .setOptions({});
})
```

Now we can simply use our `Pusher` factory to keep real-time track of our nodes. We'll create a `StatsController` to keep track of the current stats:

```
angular.module('myApp')
.controller('StatsController', function($scope, Pusher, NodeFactory) {
  Pusher.subscribe('stats', 'stats', function(data) {
```

```
    NodeFactory.addStat(data.node, 'load5', data.load5);
    NodeFactory.addStat(data.node, 'freemem', data.precentfreemem);
  });
  $scope.$on('node:added', function() {
    $scope.nodes = NodeFactory.nodes;
  });
});
```

Lastly, our HTML is going to be pretty simple. The only tricky part is looping over the current nodes as we iterate over the collection. Angular makes this pretty easy with the `ng-repeat` directive:

```
<table>
  <tr ng-repeat="(name, stats) in nodes track by $id(name)">
    <td>{{ name }}</td>
    <td>{{ stats.load5 }}</td>
    <td>{{ stats.freemem }}</td>
  </tr>
</table>
```

The source for this recipe can be found on jsbin.

Chapter 13

AngularJS SEO, mistakes to avoid.

Google does not run (most) JavaScript when it crawls the web, so it can't index our fat JavaScript client websites, including AngularJS pages.

We can have a headless browser – like PhantomJS – open our pages, run the JavaScript, and save the loaded DOM as an HTML snapshot.

When the Googlebot visits your site, we can serve them that snapshot instead of our normal page so that Google will see the rendered page same thing your users see without needing to run any JavaScript.

Read our post for more detailed explanation of this process.

However, a naive implementation of this solution isn't good enough for most sites.

In this recipe, we're going to look at a few key things to watch out for.

Caching snapshots

First and foremost, we'll need to cache our snapshots in advance. If we open and run our pages in a headless browser as Googlebot requests them, we're going going to see a ton of drag in our requests and the number of problems we're opening ourselves up for is going to cause us lots issues.

At best, it will make our site appear really slow to Google, hurting our rankings. More likely, Google will request a ton of pages at once and our webserver won't be able to launch headless browsers fast enough to keep up them. When Google comes calling, we definitely don't want to respond with 500 errors.

Now that we have a cache, we'll want to need a way to keep it up to date. When the content on a page changes, we'll need to design a way for our system to know how to update the snapshot in the cache. The specific implementation depends heavily on our architecture and business needs, but it is a question that we'll need to answer.

Do we want to make it time-based? Do we want to manually clear the cache? How do we know what needs to be cleared?

Also, what happens when we make a site wide change to our site? We'll need to be able to regenerate our entire cache. This can take a long time, depending upon how many pages our app is, how long our system will take to generate snapshots, etc. When we're adding a few new pages to the cache everyday, this might not take that much power, but when we're regenerating everything, this process can take a LONG time. It can also introduce the complexities of scaling past a single machine.

Remove JavaScript from snapshots

Google does execute some JavaScript. Sometimes things like javascript redirects or javascript content clearing can have unexpected results when left in the snapshot. Since our HTML snapshot already contains exactly what we want Google to see, we don't need to leave any javascript in the page to potentially cause problems for the search engine bot.

Design to fail

It's important that we are ready for crashes in our system. PhantomJS is great, but it has a habit of just crashing anytime it gets confused. Our solution needs to handle this gracefully, restarting PhantomJS as needed and reporting problems so we can track them down later. By designing a system that we expect to fail, we can start from the get-go to build a system designed for stability.

Special characters!!!!

Special characters are the bane of web programming and working with your headless browser is no exception. If we have any urls with characters that have to be encoded (with those ugly percent signs), we'll need to set up a test suite against them such that the routes actually resolve. It's important that we test any two byte characters, spaces, or percent signs.

Finally, if we don't want to have to deal with all of these issues ourselves, then Brombone might be a good option to check out. Reminder, they are offering calendar readers 25% off a year of SEO services. We can focus on our core products and leave the SEO hoola-hooping to them.

For our in-depth post on Angular SEO, head to What you need to know about Angular SEO

Chapter 14

AngularJS with ngAnimate

AngularJS 1.2 comes jam packed with animation support for common `ng` directives via ngAnimate module.

To enable animations within our application, we'll need to link the **angular-animate.js** JavaScript file and include **ngAnimate** as a dependency within our application. The angular-animate.js file can be downloaded from the angularjs.org homepage or from the code.angularjs.org CDN.

> It is best to stick to using **AngularJS 1.2.4 or higher** to make use of the **best animation features**.

What directives support animations?

The following directives are "animation aware" in AngularJS 1.2:

- ngRepeat (**enter**, **leave** and **move** animations)
- ngInclude (**enter** and **leave** animations)
- ngView (**enter** and **leave** animations)
- ngIf (**enter** and **leave** animations)
- ngSwitch (**enter** and **leave** animations)
- ngClass (**addClass** and **removeClass** animations)
- ngShow and ngHide (**addClass** and **removeClass** animations for `.ng-hide`).

We can also use animations within our own directives by using the `$animate` service within our link functions.

What's nice about ngAnimate is that is fully isolates animations from our directive and controller code. Animations can be easily reused across other applications by just including the CSS and/or JS animation code.

CSS3 Transitions & Keyframe Animations

`ngAnimate` supports native CSS3 transitions and keyframe animations right out of the box. With AngularJS 1.2, all we need to do is create a few compound CSS classes in our CSS code which are tied to a single base CSS class and then reference that CSS class inside of our HTML template code.

The steps we recommend are:

- think of an animation name (something like "slide")
- make a CSS class from that (something like ".slide-animation")
- setup the styles for each of the events that we wish to animate (ngInclude for example will contain `enter` and `leave` events, therefore we'll need to provide animation styles for the `.slide-animation.ng-enter` and `.slide-animation.ng-leave` CSS classes)

Pretty simple!

Transitions

`ngAnimate` works very nicely with CSS3 Transitions. All we need to do is provide **starting** and **ending** CSS classes. The starting CSS class (known as the setup class) is where we place the starting CSS styles that we wish to have when the animation starts. The ending CSS class (known as the active class) is where we'll place the styles that we expect our animation to animate towards.

For the example below, let's setup animations with `ngRepeat`.

Click here to view this demo live!

What does our code look like?

```
<input placeholder="Filter Repeat Items..." ng-model="f" />
<div ng-repeat="item in items | filter:f" class="repeat-animation">
  {{ item }}
</div>
```

```
/*
 * ngRepeat triggers three animation events: enter, leave and move.
```

```
 */
.repeat-animation.ng-enter,
.repeat-animation.ng-leave,
.repeat-animation.ng-move {
  -webkit-transition:0.5s linear all;
  transition:0.5s linear all;
}

/* ending enter and move styling
   (this is what the element will animate from */
.repeat-animation.ng-enter,
.repeat-animation.ng-move { opacity:0; }

/* ending enter and move styling
   (this is what the element will animate towards */
.repeat-animation.ng-enter.ng-enter-active,
.repeat-animation.ng-move.ng-move-active { opacity:1; }

/* starting leave animation */
.repeat-animation.ng-leave { opacity:1; }

/* ending leave animation */
.repeat-animation.ng-leave.ng-leave-active { opacity:0; }
```

(click here to and or edit the demo code in a live editor on plnkr.co).

Keyframe Animations

Keyframe animations triggered by ngAnimate work just about the same as with transitions, however, **only the setup class is used**. To make this work, we'll just reference the keyframe animation in our starting CSS class (the setup class).

For the example below, let's setup animations with `ngView`.

Click here to view this demo live!

What does our code look like?

```
<div class="view-container">
  <div ng-view class="view-animation"></div>
</div>
```

```
/*
 * ngView triggers three animation events: enter, leave and move.
 */
```

```
.view-container {
  height:500px;
  position:relative;
}

.view-animation.ng-enter {
  -webkit-animation: enter_animation 1s;
  animation: enter_animation 1s;

  /*
   * ng-animate has a slight starting delay for optimization purposes
   * so if we see a flicker effect then we'll need to put some extra
   * styles to "shim" the animation.
   */
  left:100%;
}

.view-animation.ng-leave {
  -webkit-animation: leave_animation 1s;
  animation: leave_animation 1s;
}

.view-animation.ng-leave,
.view-animation.ng-enter {
  position:absolute;
  top:0;
  width:100%;
}

/*
 * the animation below will move enter in the view from
 * the right side of the screen
 * and move the current (expired) view from the center
 * of the screen to the left edge
 */
@keyframes enter_animation {
  from { left:100%; }
  to { left:0; }
}

@-webkit-keyframes enter_animation {
  from { left:100%; }
  to { left:0; }
}

@keyframes leave_animation {
```

```
  from { left:0; }
  to { left:-100%; }
}

@-webkit-keyframes leave_animation {
  from { left:0; }
  to { left:-100%; }
}
```

(click here to and or edit the demo code in a live editor on plnkr.co).

Both CSS3 Transitions and Keyframe Animations with ngAnimate are automatically canceled and cleaned up if a new animation commences when an existing animation is midway in its own animation.

JavaScript Animations

JavaScript animations are also supported and they also work by referencing a CSS class within the HTML template.

To define our own JS animations, just use the `.animation()` factory method within our ngModule code.

For the example below, let's setup animations with `ngShow` using the amazing animation library http://www.greensock.com/.

Click here to view this demo live!

What does our code look like?

```
//be sure to link jquery and greensock.js to our application
<script
  src="//ajax.googleapis.com/ajax/libs/jquery/1.10.2/jquery.min.js">
</script>
<script
  src="http://cdnjs.cloudflare.com/ajax/libs/gsap/1.11.2/TweenMax.min.js">
</script>

//and here's our ng-show animation
<button ng-click="showMe = !showMe">Toggle Show Hide</button>
<div class="big-box show-hide-animation" ng-show="showMe">
  I am visible
</div>

var myModule = angular.module('myApp', ['ngAnimate'])
```

```
myModule.animation('.show-hide-animation', function() {
  /*
   * the reason why we're using beforeAddClass and
   * removeClass is because we're working
   * around the .ng-hide class (which is added when ng-show
   * evaluates to false). The
   * .ng-hide class sets display:none!important and we want
   * to apply the animation only
   * when the class is removed (removeClass) or before
   * it's added (beforeAddClass).
   */
  return {

    /*
     * make sure to call the done() function when the animation is complete.
     */
    beforeAddClass : function(element, className, done) {
      if(className == 'ng-hide') {
        TweenMax.to(element, 1, { height: 0, onComplete: done });

        //this function is called when the animation ends or is cancelled
        return function() {
          //remove the style so that the CSS inheritance kicks in
          element[0].style.height = '';
        }
      } else {
        done();
      }
    },

    /*
     * make sure to call the done() function when the animation is complete.
     */
    removeClass : function(element, className, done) {
      if(className == 'ng-hide') {
        //set the height back to zero to make the animation work properly
        var height = element.height();
        element.css('height', 0);

        TweenMax.to(element, 1, { height: height, onComplete: done });

        //this function is called when the animation ends or is cancelled
        return function() {
          //remove the style so that the CSS inheritance kicks in
          element[0].style.height = '';
        }
```

```
      } else {
        done();
      }
    }
  }
});
```

(click here to and or edit the demo code in a live editor on plnkr.co).

In addition to `beforeAddClass` and `removeClass`, we can also use `beforeRemoveClass` and `addClass` as well. For structural animations, such as ngRepeat, ngView, ngIf, ngSwitch or ngInclude use, we can use `enter`, `leave` and `move`.

It's up to us to determine how to perform animations within JS animations. Just be sure to call done() to close off the animation (even if we're skipping an animation based on the className … Otherwise the animation would never close).

Triggering Animations inside of our own Directives

We can easily trigger animations within our own directives by calling the animation member functions on the `$animate` service.

For the example below, let's setup animations to trigger based of a click event.

Click here to view this demo live!

What does our code look like?

```
myModule.directive('expandingZone', ['$animate', function($animate) {
  return function(scope, element, attrs) {
    var clickClassName = 'on';
    element.on('click', function(event) {
      event.preventDefault();
      element.hasClass(clickClassName) ?
        $animate.removeClass(element, clickClassName) :
        $animate.addClass(element, clickClassName);
    });
  };
}]);
```

And our HTML code looks like so:

```
<div expanding-zone class="horizontal-animation">Click me!</div>
```

Our CSS3 Transition code can be created easily by referencing the `.horizontal-animation` CSS class. Notice that our CSS code below can work even without ngAnimate being included since it follows a traditional CSS class transition. The animation will snap back and forth when the `.on` class is added or removed.

```
.horizontal-animation {
  -webkit-transition: 0.5s linear all;
  transition: 0.5s linear all;
  width:100px;
}
.horizontal-animation.on {
  width:500px;
}
```

(click here to and or edit the demo code in a live editor on plnkr.co).

Learn more! Become a Pro

To fully master ngAnimate and to learn more tricks with other libraries, be sure to visit www.yearofmoo.com. Yearofmoo is a popular programming blog which has a ton of useful AngularJS material ranging from animation to testing to SEO.

For more information, check out http://www.yearofmoo.com and ng-newsletter's animation post.

This article was written by Matias Niemelä (aka @yearofmoo) of www.yearofmoo.com and edited by Ari Lerner (aka @auser).

Chapter 15

HTML5 api: geolocation

With Angular, it's trivial to use HTML5 APIS given to us by the browser. In this snippet, we're going to walk through how to get a popular one from the browser: the `geolocation`.

When using an HTML5 API, we'll always need to handle the case when someone is visiting our page from a non-html browser that doesn't implement the same HTML5 apis. We'll deal with this shortly. First, let's look look at how to grab the geolocation on an HTML5 compliant browser (if you're not, you should be using Chrome to visit this page for the best experience).

Are we compatible?

In order to determine if a browser has the ability to fetch a geolocation, we'll need to test for the existence of the geolocation object on the `navigator` object on the `window` global object:

```
if (window.navigator && window.navigator.geolocation) {
  // Awesome, HTML5 geolocation is supported
} else {
  // HTML5 geolocation is not supported for this browser yet
}
```

With this simple check, we're ready to get the geolocation for our browser. The geolocation.getCurrentPosition() method takes a single argument: a callback function to run with the position. Because the `getCurrentPosition()` is asynchronous, we have to pass in a callback function to actually fetch the position.

```
window.navigator.geolocation.getCurrentPosition(function(position) {
  // position is a JSON object that holds the lat and long
```

```
  // if the call to getCurrentPosition() is successful.
});
```

We can wrap this into a controller function, like so such that we can attach the results to the view (as we've done in the example below):

```
$scope.getLocation = function() {
  $window.navigator.geolocation.getCurrentPosition(function(pos) {
    $scope.pos = pos;
    $scope.$digest();
  });
}
```

We can use this like so:

```
<div ng-controller='GeoLocationController'>
  <div ng-if="geoSupport()">
    <a ng-click="getLocation()" class="btn btn-primary">Get location</a>
    <span ng-bind-html="pos"></span>
  </div>
  <div ng-if="!geoSupport()"></div>
</div>
```

> Note that we had to use `$scope.$digest()` or `$scope.$apply()`. Due to the fact that the `getCurrentPosition()` function is asynchronous and outside of the scope of our angular app, we'll need to let our app *know* that something changed and we'll have to invoke a digest loop. We use `$digest()` above because we don't need to trigger a full digest loop, just one on the current scope.

Handling errors

Sometimes the geolocation api does not come back successfully. It may be that the user chose not to allow our app access to their geolocation or that perhaps the GPS unit did not return in time. We need to be able to handle these failures as they will happen. In order to handle them, the `getCurrentPosition()` allows us to pass a second error handling function that will be called with the error:

```
$scope.getLocation = function() {
  $window.navigator.geolocation.getCurrentPosition(function(pos) {
    $scope.pos = pos;
    $scope.$digest();
  }, function(err) {
```

```
    // An error has occurred with an error code
    // If the err.code is:
    //    * 0 - the error is unknown
    //    * 1 - permission was denied to the location api
    //    * 2 - position is not available
    //    * 3 - timeout
  });
}
```

Tracking location

With the `getCurrentPosition()` function, it will only ever get executed once. If we want to continuously track the user's position, we'll need to use a different function on the `geolocation` object: `watchPosition()`. The `watchPosition()` method will call the callback function **anytime the position changes**.

```
$scope.watchPosition = function(cb) {
  $window.navigator.geolocation.watchPosition(cb);
}
$scope.positionChanged = function(pos) {
  console.log("called positionChanged");
  $scope.$digest();
}
```

Where we called the `watchPosition()` method with the `positionChanged` function defined on our scope like:

```
<a ng-click="watchPosition(positionChanged)">Watch position</a>
```

The Angular way

Okay, this is all well and great, but it's not very angular. Instead of placing all of our functions on top of our `$scope`, it's better to nest them on a service that can be called from the `$scope`. We can basically copy and paste the code we already have written above into our new service with only a few minor modifications to account for using promises and events to communicate state. We'll include comments inline to explain the code:

```
.factory('Geolocation', function($q, $window, $rootScope) {
  // Default to false
  var geoCapable = false;
  // Set geoCapable to true if geolocation is supported
  if ($window.navigator && $window.navigator.geolocation) geoCapable = true;
```

```
// If we're not geoCapable, send a message to the rest of
// the angular app so we can catch it from the rest of
// the application.
if (!geoCapable) {
  $rootScope.broadcast('geo:error', 'geolocation not supported');
}
var service =  {
  // Building our service API with the two functions:
  //  * getPosition()
  //  * watchPosition()
  getPosition: function() {
    // We'll return a promise for this function
    var d = $q.defer();
    // if we're not able to handle geolocation, immediately
    // reject the promise and return
    if (!geoCapable) return d.reject();
    // Call the getCurrentPosition function on the
    // geolocation object
    $window.navigator.geolocation.getCurrentPosition(function(pos) {
      // Resolve the promise as we did above
      d.resolve(pos);
      $rootScope.$apply();
    }, function(err) {
      // If there was an error, send an event back to the
      // rest of the app and reject the promise
      $rootScope.$broadcast('geo:error', err);
      d.reject(pos);
      $rootScope.$apply();
    })
    // return the promise
    return d.promise;
  },
  watchPosition: function(cb) {
    // First, if we aren't geoCapable then immediately return
    // and potentially call a potential callback if it's given
    if (!geoCapable) {
      $rootScope.$broadcast('geo:error', 'Geo not supported');
      if (cb) cb('Geo not supported');
      return;
    }
    // Call the watchPosition() function on the geolocation
    $window.navigator.geolocation.watchPosition(function(pos) {
      // Anytime the position changes, invoke an event so
      // we can use the event to communicate across the app.
      $rootScope.$broadcast('geo:positionChanged', pos);
```

```
        // If a callback is given, then run the callback
        if (cb) {
          cb(null, pos);
          $rootScope.$apply();
        }
      });
    }
  }

  return service;
})
```

We can use this factory by calling the API on our scope. Notice that we provide multiple ways to use the results on our scope. This way we can pass in a function OR we can simply listen for events on our scopes.

```
angular.module('myApp')
.controller('GeoController', function($scope, Geolocation) {
  $scope.getPosition = function() {
    Geolocation.getPosition()
    .then(function(pos) {
      $scope.position = pos;
    })
  }
});
```

Chapter 16

HTML5 api: camera

In the previous recipe, we looked at how to grab the geolocation using the HTML5 geolocation API. In this snippet, we're going to check out how to grab camera controls through the HTML5 api with Angular.

There is quite a lot of history surrounding the camera API. Although we won't go through it here as it's outside of the scope of this snippet, but it's important to note that because of the not-yet-solidified camera API and we'll need to work around it. We'll also then dive head-first into using it with Angular. Let's get started.

Are we compatible?

In order to check if our user's browser implements the camera API we'll need to check to see if it implements any of the *known* camera API functions. We can iterate through all of the different options and check if the browser implements them.

```
var getUserMedia = function() {
  if (navigator.getUserMedia) return navigator.getUserMedia;
  else if (navigator.webkitGetUserMedia) return navigator.webkitGetUserMedia;
  else if (navigator.mozGetUserMedia) return navigator.mozGetUserMedia;
  else if (navigator.msGetUserMedia) return navigator.msGetUserMedia;
  else return undefined;
}
```

It's always a good idea to check for compatibility and show the result to the user. We can do it with the following snippet:

"'javascript

Your browser supports a version of the camera api, success!

Your browser does not support the camera API :(Try a different browser (like Chrome)

> We can also use a tool like Modernizr to gain access to the `getUserMedia` api.

Once we've determined if the camera API is available, we can request access to it from the browser. To request access, we'll need to request which type of access we would like. There are two types of available input we can take:

- audio
- video

The first argument of the `userMedia` object is the JSON object of which media input we are interested. The second two are the success callback function and the error callback function that get called with their respective place when the method succeeds or fails.

```
getUserMedia({
  audio: false,
  video: true
}, function success(stream) {
  // We now have access to the stream
}, function failure(err) {
  // It failed for some reason
});
```

Once one of these callbacks return, we'll have access to our media api.

Wrap it into a service

Before we get into actually using the API, let's wrap this component in a service. This will make it incredibly easy to test later on (not covered in *this* snippet). It will also abstract away any dealings we'll need to do with the `userMedia` object and is quite simple:

```
angular.module('myApp')
.factory('CameraService', function($window) {
  var hasUserMedia = function() {
    return !!getUserMedia();
  }
```

```
  var getUserMedia = function() {
    navigator.getUserMedia = ($window.navigator.getUserMedia ||
                              $window.navigator.webkitGetUserMedia ||
                              $window.navigator.mozGetUserMedia ||
                              $window.navigator.msGetUserMedia);
    return navigator.getUserMedia;
  }

  return {
    hasUserMedia: hasUserMedia(),
    getUserMedia: getUserMedia
  }
})
```

With this service, we can simply request access to the user media inside of our other angular objects. For instance:

```
angular.module('myApp')
.controller('CameraController', function($scope, CameraService) {
  $scope.hasUserMedia = CameraService.hasUserMedia;
})
```

Setting up the directive

Now that we have a hold of the `userMedia` object, we can request access to it. Since we're going to place this element on the page, it's a good idea to write it as a directive. The non-camera-specific directive code:

```
angular.module('myApp')
.directive('camera', function(CameraService) {
  return {
    restrict: 'EA',
    replace: true,
    transclude: true,
    scope: {},
    template: '<div class="camera"><video class="camera" autoplay="" />\
              <div ng-transclude></div></div>',
    link: function(scope, ele, attrs) {
      var w = attrs.width || 320,
          h = attrs.height || 200;

      if (!CameraService.hasUserMedia) return;
      var userMedia = CameraService.getUserMedia(),
```

```
          videoElement = document.querySelector('video');
      // We'll be placing our interaction inside of here
    }
  }
});
```

Using the camera API

We can now request access to the camera inside of our link function in the directive. This simply means that we'll set up the two callback functions that will be called from the `getUserMedia()` method call and then make the request for permission:

```
// Inside the link function above
// If the stream works
var onSuccess = function(stream) {
  if (navigator.mozGetUserMedia) {
    videoElement.mozSrcObject = stream;
  } else {
    var vendorURL = window.URL || window.webkitURL;
    videoElement.src = window.URL.createObjectURL(stream);
  }
  // Just to make sure it autoplays
  videoElement.play();
}
// If there is an error
var onFailure = function(err) {
  console.error(err);
}
// Make the request for the media
navigator.getUserMedia({
  video: {
    mandatory: {
      maxHeight: h,
      maxWidth: w
    }
  },
  audio: true
}, onSuccess, onFailure);

scope.w = w;
scope.h = h;
```

Once we've asked permission, the video element will get a src url and start playing. For instance:

```
<div ng-controller="CameraController">
  <div ng-if="hasUserMedia">
    <a class="btn btn-info" ng-click="enabled=!enabled">Demo</a>
    <camera ng-if="enabled"></camera>
  </div>
  <div ng-if="!hasUserMedia" class="sad">
    Your browser does not support the camera API
  </div>
</div>
```

Extending the directive

This directive is pretty simple and reasonably so. However, what if we want to make a call to say, create a snapshot of the current frame. For instance, to allow our users to take a photo of themselves for a profile photo. This is pretty easy to do as well. We'll simply add a function on our `scope` that will handle taking the photo for us.

Note, that we will create a way for us to extend the directive with new controls, each we'll create in the similar fashion. First, to implement this pattern, we'll need to add a controller to our parent directive (`camera`):

```
// scope: {},
controller: function($scope, $q, $timeout) {
  this.takeSnapshot = function() {
    var canvas  = document.querySelector('canvas'),
        ctx     = canvas.getContext('2d'),
        videoElement = document.querySelector('video'),
        d       = $q.defer();

    canvas.width = $scope.w;
    canvas.height = $scope.h;

    $timeout(function() {
      ctx.fillRect(0, 0, $scope.w, $scope.h);
      ctx.drawImage(videoElement, 0, 0, $scope.w, $scope.h);
      d.resolve(canvas.toDataURL());
    }, 0);
    return d.promise;
  }
},
// ...
```

Now we can safely create camera controls with this parent DOM element. For instance, the snapshot controller:

```
.directive('cameraControlSnapshot', function() {
  return {
    restrict: 'EA',
    require: '^camera',
    scope: true,
    template: '<a ng-click="takeSnapshot()">Take snapshot</a>',
    link: function(scope, ele, attrs, cameraController) {
      scope.takeSnapshot = function() {
        cameraController.takeSnapshot()
        .then(function(image) {
          // data image here
        });
      }
    }
  }
})
```

This will need to placed inside the camera, like so:

```
<camera>
  <camera-control-snapshot></camera-control-snapshot>
</camera>
```

Now that you know how to handle the camera inside Angular, put on your best face and start taking some photos!

The source for the article can be found at the jsbin example.

Chapter 17

Good deals – Angular and Groupon

In this recipe, we're looking at how to connect our application to the great deals offered by the Groupon API.

The Groupon API requires a an `api_key` to authenticate the requesting user. Luckily, getting a Groupon `api_key` is really easy to do.

Head over to the Groupon API docs page at http://www.groupon.com/pages/api and click on the giant *Get My Api Key* button.

After we sign up, we get our API key. Let's hold on to this key for the time being.

Providing the API

Anytime that we are building a service that we'll want to configure, for example with an API key we'll want to use the `.provider()` function to create a service. This will allow us to craft a service that holds on to our key for future requests.

We'll write the service to be able to be configured with an api key. This can be accomplished like so:

```
angular.module('alGroupon', [])
.provider('Groupon', function() {
  // hold on to our generic variables
  var apiKey = '',
      baseUrl = '//api.groupon.com/v2';

  // A function to set the api key
```

```
  this.setApiKey = function(key) {
    apiKey = key || apiKey;
  }

  this.$get = [function() {
    // Service definition
  }]
});
```

In our app, we'll use this module by setting it as a dependency and then configuring our service inside of a `.config()` function. For example:

```
angular.module('myApp', ['alGroupon'])
.config(function(GrouponProvider) {
  GrouponProvider
    .setApiKey('XXXXXXXXXXXXXXXXXXX');
})
```

Defining the service

Now we haven't actually created any functionality for the Groupon service. Let's go ahead and create a function that will find some of the latest deals.

The API method to get the latest deals (can be found in the docs page) is at the route `/deals.json`. Luckily for us, the Groupon API supports JSONP fetching which allows us to fetch the deals without needing to concern ourselves with CORS.

Inside of our requests, we'll need to pass in two parameters for each one of our requests:

- callback
- client_id

As we have these already available to us in our service, we can create a small helper function that will create the request parameters we'll eventually use to make the service request:

```
// The service definition
this.$get = ['$q', '$http',
  function($q, $http) {
    var prepareRequest = function(conf) {
      // Ensure we have a config option
      conf = conf || {}
```

```
      // Set the callback and the client_id
      // in the config object
      conf['callback'] = 'JSON_CALLBACK';
      conf['client_id'] = api_key
      return conf;
    }
    // ...
}]
```

With that set, let's create the function that gets some deals. This simply is a function that we can set up with the standard `$http` request:

```
this.$get = ['$q', '$http',
  function($q, $http) {
    // ...
    var service = {
      getDeals: function(conf) {
        var d = $q.defer();
        conf = prepareRequest(conf);
        // Execute the request in the background
        $http({
          method: 'JSONP',
          url: baseUrl + '/deals.json',
          params: conf
        }).success(function(data) {
          d.resolve(data.deals);
        }).error(function(reason) {
          d.reject(reason);
        })
        return d.promise;
      }
    }
    return service;
}]
```

> We're creating a custom promise in this example. Although this isn't necessary, it allows us to pass back custom data. In here, we're passing back not just data, but `data.deals`.

Now, we can simply inject this into our controller and call `getDeals` on the service as we would any other service.

```
angular.module('myApp', ['alGroupon'])
.controller('GrouponController',
```

```
  function($scope, Groupon) {
    $scope.getDeals = function() {
      Groupon.getDeals()
      .then(function(deals) {
        $scope.deals = deals;
      });
    }
});
```

We can use this like so:

```
<div ng-controller='GrouponExampleController'>
  <a ng-click="getDeals()" class="btn btn-warning">Get deals</a>
  <div ng-show="showing">
    <a ng-click="showing=!showing" class="btn btn-primary">Hide</a>
    <span ng-bind-html="deals"></span>
  </div>
</div>
```

Notice in our function we allow the user pass in configuration for the requests. We can use this configuration object to customize our request to the Groupon API.

That's it. Connecting with Groupon is that easy with our configurable service.

> The Groupon API documentation can be found here.

The source for this recipe can be found live at jsbin.

Chapter 18

Staggering animations with ngAnimate

After getting a glimpse at how to perform animations with ngAnimate in last week's ng-newsletter entry: AngularJS with ngAnimate, let's expand our knowledge and learn how to supercharge our animations even further. What are we going to learn? Staggering Animations!

AngularJS 1.2 introduces a hidden, but powerful, CSS trick that informs `$animate` to stagger all successive CSS animations into a curtain-like effect.

> It is best to stick to using AngularJS 1.2.4 or higher to make use of the best animation features.

How can we put this to use?

Let's say we have a ngRepeat animation and, instead of everything animating at the same time, we would like there to be a 50 millisecond delay between each enter operation. This would be quite difficult to do on our own using JavaScript with timeout delays, but luckily it is very easy to do using ngAnimate by adding some additional CSS code that the '$animate service understands.

Let's **continue** off our our **code from last week's article** and apply some stagger animations to it. What did we have? Just a few simple transition animations for the enter, leave and move events on a list of items rendered by ngRepeat.

```
/*
 * ngRepeat triggers three animation events: enter, leave and move.
```

```
 */
.repeat-animation.ng-enter,
.repeat-animation.ng-leave,
.repeat-animation.ng-move {
  -webkit-transition:0.5s linear all;
  transition:0.5s linear all;
}

/* ending enter and move styling
   (this is what the element will animate from */
.repeat-animation.ng-enter,
.repeat-animation.ng-move { opacity:0; }

/* ending enter and move styling
   (this is what the element will animate towards */
.repeat-animation.ng-enter.ng-enter-active,
.repeat-animation.ng-move.ng-move-active { opacity:1; }

/* starting leave animation */
.repeat-animation.ng-leave { opacity:1; }

/* ending leave animation */
.repeat-animation.ng-leave.ng-leave-active { opacity:0; }
```

Click here to see the full HTML + CSS code.

Now let's enhance this by adding in a **stagger effect** to the **enter**, **leave** and **move** animations.

```
.repeat-animation.ng-enter,
.repeat-animation.ng-leave,
.repeat-animation.ng-move {
  -webkit-transition:0.5s linear all;
  transition:0.5s linear all;
}

.repeat-animation.ng-enter-stagger,
.repeat-animation.ng-leave-stagger,
.repeat-animation.ng-move-stagger {
  /* 50ms between each item being animated after the other */
  -webkit-transition-delay:50ms;
  transition-delay:50ms;

  /* this is required here to prevent any CSS inheritance issues */
  -webkit-transition-duration:0;
  transition-duration:0;
```

```
}

/* the rest of the CSS code... */
```

And voila! We have a nice delay gap between each item being entered. To see this demo in action, click on the link below.

http://d.pr/l4ln

What about CSS3 Keyframe Animations?

The code above was crafted together using CSS transitions, but CSS keyframe animations can also be used to trigger the animation. Let's expand our code from above, change the CSS styling to use keyframe animations instead of transitions.

The keyframe animation code is very similar to our transition code, but instead of using `transition` we use `animation`.

```
.repeat-animation.ng-enter {
  -webkit-animation: enter_animation 0.5s;
  animation: enter_animation 0.5s;
}
.repeat-animation.ng-leave {
  -webkit-animation: leave_animation 0.5s;
  animation: leave_animation 0.5s;
}

@-webkit-keyframes enter_animation {
  from { opacity:0; }
  to { opacity:1; }
}

@keyframes enter_animation {
  from { opacity:0; }
  to { opacity:1; }
}

@-webkit-keyframes leave_animation {
  from { opacity:1; }
  to { opacity:0; }
}

@keyframes leave_animation {
  from { opacity:1; }
```

```
  to { opacity:0; }
}
```

(click here to get learn the basics of ngAnimate with keyframe animations).

So far so good. Now let's enhance that yet again using some stagger CSS code

```
.repeat-animation.ng-enter-stagger,
.repeat-animation.ng-leave-stagger,
.repeat-animation.ng-move-stagger {
  /* notice how we're using animation instead of transition here */
  -webkit-animation-delay:50ms;
  animation-delay:50ms;

  /* yes we still need to do this too */
  -webkit-animation-duration:0;
  animation-duration:0;
}

/* the rest of the CSS code... */
```

Keep in mind, however, that CSS Keyframe animations are triggered only after the "reflow" operation has run within an animation triggered by ngAnimate. This means that, depending on how many animations are being run in parallel, the element may appear in a pre-animated state for a fraction of a second.

Why? Well this is a performance boost that the `$animate` service uses internally to queue up animations together to combine everything to render under one single reflow operation. So if our animation code starts off with the element being hidden (say opacity:0) then the element may appear as visible before the animation starts. To inform `$animate` to style the element beforehand, we can just place some extra CSS code into our setup CSS class (the same CSS class where we reference the actual keyframe animation).

```
.repeat-animation.ng-enter {
  /* pre-reflow animation styling */
  opacity:0;

  /* the animation code itself */
  -webkit-animation: enter_animation 0.5s;
  animation: enter_animation 0.5s;
}
```

And finally, here's a link to our amazing, stagger-enhanced, CSS keyframe animation code: http://d.pr/dmQk.

Ahh this is sweet isn't it?

What directives support this?

Both **ngRepeat** and **ngClass** support this, however, any angular directive can support it as well. It isn't actually the directive that triggers this animation, but instead, **a stagger animation will be triggered** if **two or more animations** of the same animation event residing within the same element container (the parent element) **are issued at the same time**. And with ngRepeat this happens automatically.

To get this to work with a directive like ngClass then it is just a matter of triggering a series of CSS class changes on a series of elements all under the same parent class. Please visit the yearofmoo article at the end of this article to view some awesome examples to issue animations using ngClass.

Understanding how this works can help us make stagger effects in our own directive code by using the helpful DOM operations available on the `$animate` method.

Click here to get a full understanding of how stagger animations are triggered by the `$animate` service.

But what about JS animations?

Right now staggering animations are not provided within JavaScript animations. This is because the feature itself is very new and young so it's best to see how popular and well this feature plays out on the web before cramming more features into AngularJS.

However, all is not lost, the helpful yearofmoo article below goes into detail about how to piece together a stagger-like animation using JavaScript animations in ngAnimate.

Where can I learn more?

An in-depth article, examples and a demo repo is available on www.yearofmoo.com in the article entitled: Staggering Animations in AngularJS. This article goes into detail on how to make use of staggering animations with ngAnimate, how to create custom directives to hook into staggering animations, and also how to wire them together with the amazing animate.css CSS animation library.

This article was written by Matias Niemela (aka @yearofmoo) of www.yearofmoo.com and edited by Ari Lerner (aka @auser) of ng-newsletter.

Chapter 19

Getting started unit-testing Angular

One of the fundamental reasons for choosing Angular is cited as that it is built with testing in mind. We can build complex web applications using various popular frameworks and libraries with features as tall as the sky and as equally complex. As with anything of increasing complexity and density, this can quickly grow out of hand.

Testing is a good approach to keep code maintainable, understandable, debug-able, and bug-free. A good test suite can help us find problems before they rise up in production and at scale. It can make our software more reliable, more fun, and help us sleep better at night.

There are several schools of thought about how to test and when to test. As this snippet is more about getting us to the tests, we'll only briefly look at the different options. In testing, we can either:

- Write tests first (Test-Driven Development | TDD) where we write a test to match the functionality and API we expect out of our element
- Write tests last where we confirm the functionality works as expected (WBT | Write-behind testing)
- Write tests to black-box test the functionality of the overall system

For the application that we're working on, it's really up to us to determine what style makes sense for our application. In some cases, we operate using TDD style, while in others we operate with WBT.

> We generally follow the following pattern when choosing a testing style: while we're at prototyping phase (or just after), we generally

> work with WBT testing as we don't always have a solidified API we're working with. Additionally, our team is generally pretty small. Otherwise, when our application starts to grow, we switch over to drive our functionality through testing.

Getting started

Enough chit-chat, let's test!

There are several libraries that we can use right out of the box when testing angular apps. The currently, most popularly supported framework released and sponsored by the Google team is called `karma`.

If we're not using the `yeoman.io` generator, we'll need to install karma for development puposes. Installing `karma` can by done using the `npm` tool, which is a package manager for node and comes built-in:

```
$ npm install --save-dev karma
```

> If we're using the `yeoman` generator with our apps, this is already set up for us.

`Karma` works by launching a browser, loading our app or a derivative of our source files and running tests that we write against our source code. In order to *use* karma, we'll need to tell the framework about our files and all the various requirements.

To kick it off, we'll use the `karma init` command which will generate the initial template that we'll use to build our tests:

```
$ karma init karma.conf.js
```

It will ask us a series of questions and when it's done, it will create a configuration file. Personally, we usually go through and answer yes to as many which are asked (except without RequireJS). We like to fill in the `files:` section manually (see below).

When it's done, it will create a file in the same directory where we ran the generator that looks similar to the following:

```
// Karma configuration
module.exports = function(config) {
  config.set({
    // base path, that will be used to resolve files and exclude
    basePath: '',
```

```
    // testing framework to use (jasmine/mocha/qunit/...)
    frameworks: ['jasmine'],

    // list of files / patterns to load in the browser
    files: [
      'app/components/angular/angular.js',
      'app/components/angular-mocks/angular-mocks.js',
      'app/scripts/**/*.js',
      'test/spec/**/*.js'
    ],

    // list of files / patterns to exclude
    exclude: [],

    // web server port
    port: 8080,

    // level of logging
    // possible values:
    // LOG_DISABLE || LOG_ERROR || LOG_WARN || LOG_INFO || LOG_DEBUG
    logLevel: config.LOG_INFO,

    // enable / disable watching file and executing tests
    // whenever any file changes
    autoWatch: false,

    // Start these browsers, currently available:
    // - Chrome
    // - ChromeCanary
    // - Firefox
    // - Opera
    // - Safari (only Mac)
    // - PhantomJS
    // - IE (only Windows)
    browsers: ['Chrome'],


    // Continuous Integration mode
    // if true, it capture browsers, run tests and exit
    singleRun: false
  });
};
```

This `karma.conf.js` file describes a simple `unit` test that karma will load when we start writing tests. We can also tell it to build an e2e, or end-to-end test

that is specifically intended for building black-box style testing, but that's for another snippet/article.

> Note that we **need** to have `angular-mocks.js` and our angular code available to reference inside the `karma.conf.js` file.

Now that we have our `karma.conf.js` file generated, we can kick off karma by issuing the following command:

```
$ karma start karma.conf.js
```

If we haven't run it before, it's likely going to **fail** or at least report errors of files not being found. Let's start writing our first test.

In Write-Behind development, we're going to test the following controller:

```
angular.module('myApp', [])
.controller('MainController', function($scope) {
  $scope.name = "Ari";
  $scope.sayHello = function() {
    $scope.greeting = "Hello " + $scope.name;
  }
})
```

First, let's create the actual test file in `test/spec/controllers/main.js`. Since `karma` works well with Jasmine, we'll be using the Jasmine framework as the basis for our tests.

> For information on Jasmine, check out their fantastic documentation at http://jasmine.github.io/2.0/introduction.html.

Inside our freshly created file, let's add the test block:

```
describe('Unit: MainController', function() {
  // Our tests will go here
})
```

Great! Now we need to do a few things to tell our tests what we are testing. We can do this by using the `beforeEach()` function to load the angular module that contains our `MainController` (in this case, it's just `myApp`):

```
describe('Unit: MainController', function() {
  // Load the module with MainController
  beforeEach(module('myApp'));
})
```

Next, we're going to *pretend* that we're angular loading the controller when it needs to be instantiated on the page. We can do this by *manually* instantiating the controller and handing it a `$scope` object. Creating it manually will also allow us to interact with the scope throughout the tests.

```
describe('Unit: MainController', function() {
  // Load the module with MainController
  beforeEach(module('myApp'));

  var ctrl, scope;
  // inject the $controller and $rootScope services
  // in the beforeEach block
  beforeEach(inject(function($controller, $rootScope) {
    // Create a new scope that's a child of the $rootScope
    scope = $rootScope.$new();
    // Create the controller
    ctrl = $controller('MainController', {
      $scope: scope
    });
  }));
})
```

Now, we have access to both the controller as well as the scope of the controller.

Writing a test

Now that everything is all set up and ready for testing, let's write one. It's *always* a good idea to test functionality of code that we write. Anytime that variables can be manipulated by the user or we're running any custom actions, it's usually a good idea to write a test for it.

In this case, we won't need to test setting the name to "Ari" as we *know* that will work (it's JavaScript). What we would like to know, however is that the `sayHello()` function works as-expected.

The `sayHello()` method simply preprends the `$scope.name` to a variable called `$scope.greeting`. We can write a test that verifies that `$scope.greeting` is `undefined` before running and then filled with our expected message after we run the function:

```
describe('Unit: MainController', function() {
  // Load the module with MainController
  beforeEach(module('myApp'));

  var ctrl, scope;
  // inject the $controller and $rootScope services
  // in the beforeEach block
  beforeEach(inject(function($controller, $rootScope) {
    // Create a new scope that's a child of the $rootScope
    scope = $rootScope.$new();
    // Create the controller
    ctrl = $controller('MainController', {
      $scope: scope
    });
  }));

  it('should create $scope.greeting when calling sayHello',
    function() {
      expect(scope.greeting).toBeUndefined();
      scope.sayHello();
      expect(scope.greeting).toEqual("Hello Ari");
  });
})
```

We have access to all different parts of the controller through the `scope` of it now. Feel the power of testing? This is only the beginning. For a deeper dive into testing, check out the expansive testing chapter in ng-book.

Chapter 20

Build a Real-Time, Collaborative Wishlist with GoAngular v2

GoInstant is a platform for building real-time, collaborative web and mobile apps. We don't need to worry about creating a separate back-end or server infrastructure, GoInstant will handle that part for us.

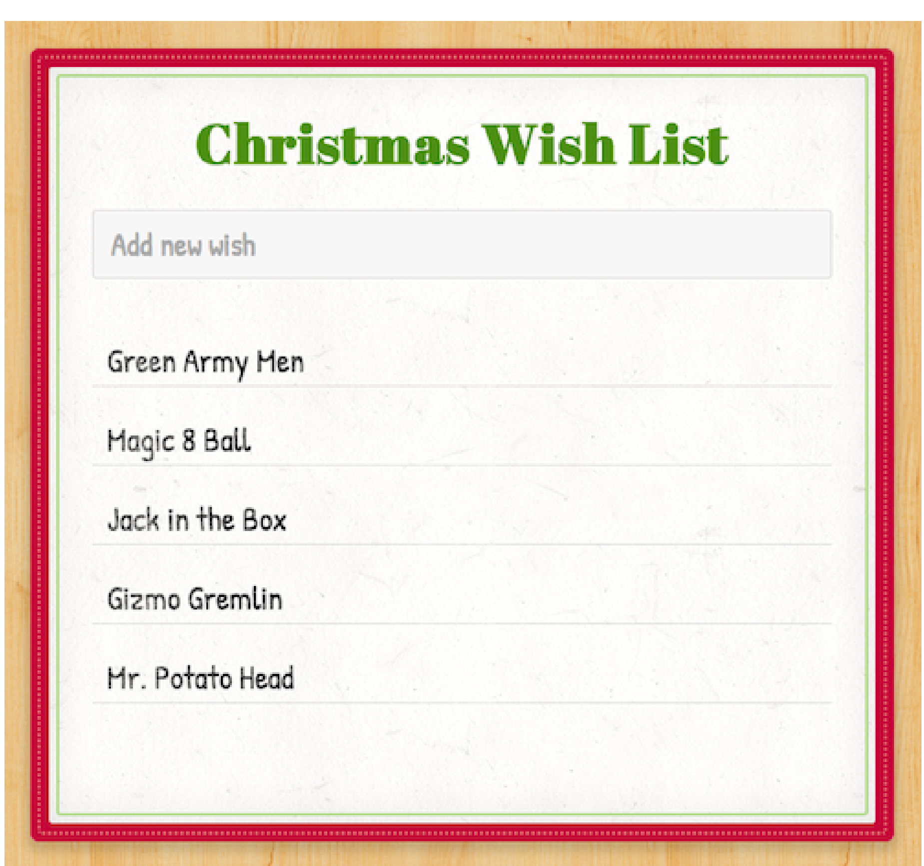
GoAngular, the GoInstant integration with AngularJS, makes it incredibly easy to use Angular and GoInstant together. The GoInstant team just released their V2 which is WAY more powerful than the V1 and made some big improvements including:

- Access to all of GoInstant's most powerful features! Including: user-management, presence, and a real-time data store.
- Promise-based API

In this snippet, we're going to look at how to use GoInstant and GoAngular to create the collaborative wishlist you see below. Check out the final version of the app here.

> We only need NodeJS and npm installed to get our wishlist app up and running. For more information on installing NodeJS, check out the nodejs.org page.

If you want to follow along, the source is available at https://github.com/mattcreager/goangular-wishlist-example and includes directions to help you get up and running.


Christmas Wish List
Add new wish
Green Army Men
Magic 8 Ball
Jack in the Box
Gizmo Gremlin
Mr. Potato Head

Sign up for GoInstant

Before we can get started, we'll need to sign up for GoInstant and create an app. Let's call it *wishlist*.

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

Include GoInstant & GoAngular

Next we'll add our external dependencies: GoInstant and GoAngular are available on bower or via their CDN.

In our `index.html`, we'll need to make sure the following lines are added:

```
<script
src="https://cdn.goinstant.net/v1/platform.min.js">
</script>
<script
src="https://cdn.goinstant.net/integrations/goangular/v2.0.0/goangular.min.js">
</script>
```

Add GoAngular as a dependency

Let's hit two birds with one stone: We'll declare dependencies for our app module and controller, and configure our GoInstant connection, with the URL we retrieved in step one.

```
angular.module('WishList', ['goangular'])
   .config(function(goConnectionProvider) {
     // Connect to our GoInstant backend url
     goConnectionProvider.set(CONFIG.connectUrl);
  })
```

After we've included `goangular` as a dependency for our app, we can *inject* the `goConnection` into our angular objects, which we'll use to connect to our GoInstant back-end:

```
.controller('ListController', function($scope, goConnection) {
  // goConnection is available in here
});
```

Create and Join a Room

Before we can access a key, we need to be in a room, which is the GoInstant idiom for channels. The `goConnection` service provides easy access to our pre-configured connection; we'll use that to create and join a room:

```
angular.module('WishList', ['goangular'])
.controller('ListController', function($scope, goConnection) {
  var itemsKey;

  goConnection.ready()
  .then(function(goinstant) {
    // create a room, join it and return a promise
    return goinstant.room('a-list').join().get('room');
  });
});
```

Notice that we're returning a promise inside of the `goConnection` connection. This tells the GoInstant library to resolve the promise and then pass it along the promise chain. Specifically, this allows us to interact with the promised object in the promise chain.

Using the promise chain guarantees that the interaction with GoInstant will be asynchronous and fits naturally into the async flow of javascript.

Fetch our wishes

Now we can use the room from the previous promise inside the *next* promise. In this case, we're trickling information down from the room, which in turn gets used to fetch the list of items inside the room, which finally gets resolved onto the `$scope` object as `$scope.items`.

```
goConnection.ready().then(function(goinstant) {
  return goinstant.room('a-list').join().get('room');
}).then(function(room) {
  return room.key('items').get();
}).then(function(result) {
  $scope.items = result.value || {};
}).finally($scope.$apply);
```

This interaction is actually quite performant and also makes it incredibly easy to test the features of our app.

Watch our wishes, so we know when they change

When a wish is added, we can just add it to our model: `$scope.items`.

```
var itemsKey;

goConnection.ready().then(function(goinstant) {
  return goinstant.room('a-list').join().get('room');
}).then(function(room) {
  itemsKey = room.key('items');

  return itemsKey.get();
}).then(function(result) {
  $scope.items = result.value || {};

  $scope.addWish = function() {
    itemsKey.add($scope.wish);
  };

  itemsKey.on('add', {
    local: true,
    listener: function(value, context) {
      $scope.$apply(function() {
        var itemKey = context.addedKey.split('/')
        $scope.items[itemKey[itemKey.length -1 ]] = value;
      });
    }
  });
}).finally($scope.$apply);
```

There we have it, a functional real-time, collaborative wishlist! Dig in further with GoAngular's documentation, join the conversation in IRC.

From our experience, the team is incredibly responsive and will accept feedback to support@goinstant.com.

The GoAngular team recorded a screencast that walks through this process! Check it out at https://www.youtube.com/watch?v=6KspFPDOaMY&hd=1.

Chapter 21

Immediately satisfying users with a splash page

One of the common questions we get as teachers of Angular is how can we prevent the Flash Of Unrendered Content (FOUC). We covered a few methods to handle hiding this unrendered content on Day 3, but these simply hide the content from the user.

Hiding content, especially on mobile can make our app appear to not be working, slow, or broken. Rather than let our users *think* that our app is broken, we can address it specifically by showing them *something* while they wait for our app to load.

> We spoke about this in our talk at Google (view it here)

One incredibly short introduction how the browser works

When the browser, any browser gets HTML it will start to parse the HTML into a tree. We refer to this tree as the Document Object Model, or the DOM for short. When the browser has parsed the entire HTML, it will start walking down the tree and begin laying out the elements.

If it encounters an element that requires it to fetch data from a remote site, such as a `<link>` tag or a `<script>` tag with a `src` attribute, the browser will immediately try to fetch the resource before continuing on down the tree.

Additionally, any element in the `<head>` will be fetched before the rest of the document has been parsed and presented on-screen.

The browser takes care of placing elements on the screen as it encounters elements. This process is usually quite fast, but slow connections and slow devices allow us to see it in full-blown action.

So what can we do?

Use this process to our advantage

We can use the fact that the browser starts to render elements as soon as it encounters them to show a splash screen while the rest of the page is loading. A splash screen is simply a full-screen block of HTML that we will show when the page is loading.

To implement a splash screen, we'll need to strip all of the loading of any external files OUTSIDE of the `<head>` element. This includes any external CSS and JavaScript. The only elements we want in the `<head>` element are meta tags, the `<title>` element, and any in-line styles that our splash screen will use (as we'll see shortly).

Secondly, the first element that we want in our `<body>` tag is the splash screen head element. This way, once the browser starts parsing the `<body>` tag it will lay out the splash screen elements and style them before it starts to load any other files.

```
<html>
  <head>
    <title>My Angular App</title>
    <style type="text/css">
      .splash {}
    </style>
  </head>
  <body ng-app="myApp">
    <!-- The splash screen must be first -->
    <div id="splash" ng-cloak>
      <h2>Loading</h2>
    </div>
    <!-- The rest of our app -->
    <script
      src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.11/angular.min.js">
    </script>
    <div ng-controller="MainController">
    </div>
    <script src="js/app.js"></script>
  </body>
</html>
```

Laying out our page like this will give us the most optimal appearing loading of our app. What goes inside our splash screen element is really up to us. The more we can *inline* the elements that appear in the splash screen, the faster it will run.

Using base64 encoding for our images or SVG elements are easy methods of making our splash screen appear to be part of the normal, non-static flow of a splash screen. The more external elements we place inside of the top-most element, the slower it will appear.

Now, once we have our HTML setup like what we have above, we'll need to style our `<div id="splash">` element. We can do this by placing the style definitions in the sole `<style>` element in the `<head>` tag of our page.

```
<head>
  <title>My Angular App</title>
  <style type="text/css">
    /* some naive styles for a splash page */
    .splash {
      background: blue;
      color: white;
    }
    .splash h2 {
      font-size: 2.1em;
      font-weight: 500;
    }
  </style>
</head>
```

Hiding our splash screen when the app is ready

Lastly, we'll need to make sure that our splash screen goes away at the end of the loading process and let our app reclaim the page.

We can do this using the `ngCloak` directive that is built into Angular itself. The `ngCloak` directive's entire responsibility is to hide uncompiled content from the browser before the angular app has been loaded.

The `ngCloak` directive works by adding a CSS class to the element that sets a `display: none !important;` to the element. Once the page has been loaded and angular is bootstrapping our app, it will remove the `ng-cloak` class from the element which will remove the `display: none` and show the element.

We can hijack this `ngCloak` directive for the splash page and invert the style for the `splash` element only. For example, in the CSS in our `<head>` element, we can place the following CSS definitions:

```
[ng-cloak].splash {
  display: block !important;
}
[ng-cloak] {display: none;}
```

```
/* some naive styles for a splash page */
.splash {
  background: blue;
  color: white;
}
.splash h2 {
  font-size: 2.1em;
  font-weight: 500;
}
```

With this in place, our splash screen will load and be shown in our app before anything else is shown and we made **NO** changes to the functionality of our app.

> For more information on mobile, stay tuned to ng-newsletter for updates on our upcoming in-depth screencast on developing mobile apps with Angular.

Chapter 22

A Few of My Favorite Things: Isolated Expression Scope

What do we do when we need to create a component that needs to be reusable and want to give other developers the ability to use that component however they wish?

In AngularJS, isolated expression scope is a really powerful tool that allows us to delegate units of work to the parent controller of a directive instead of handling it internally. This flexibility allows us to use the same directive over and over but have it do totally different things depending on how we define it in the HTML.

In this article, we are going to create a single directive that we will use seven times to do completely different things. When we are finished, we will have seen the power and flexibility of isolated expression scope applied in an approachable manner that you can start to using immediately. Let's get started!

The code for this article is **here**

Isolated Expression Scope

We are going to create a `vote` directive that has an `vote-up` and `vote-down` method. Isolated expression scope comes into play the moment we decide that we want to be able to increment and decrement the vote count by varying denominations. In our case, we want one vote directive instance to increment and decrement the vote count by 100, another to increment and decrement the vote count by 10 and so on.

We will use the HTML as a starting point and walk through how the vote directive actually comes to together. We have a `MainController` that has a property of `votes` that we are going to use to keep track of our total vote count. We also have three vote components on the page and in each one we are defining `label`, `vote-up` and `vote-down` differently.

```
<div class="container" ng-controller="MainController">
  <h1>Current Votes: {{votes}}</h1>
  <hr>
  <h3>Isolated Expression Scope</h3>
  <div class="vote-container">
    <vote label="100" vote-up="incrementHundred()"
      vote-down="decrementHundred()"></vote>
    <vote label="10" vote-up="incrementTen()"
      vote-down="decrementTen()"></vote>
    <vote label="1" vote-up="incrementOne()"
      vote-down="decrementOne()"></vote>
  </div>
</div>
```

Notice that the label corresponds with the denomination that the vote directive increments or decrements the vote count. The directive with the label `100` has a method of `incrementHundred` and `decrementHundred` which does exactly that to the `votes` property.

Now that we have created some AngularJS requirements by defining them in our HTML, let us fulfill those in our JavaScript. First up, we have created the `MainController` with the `votes` property defined on scope as well as all of the increment and decrement properties we declared.

Just below that we are going to declare the `vote` directive.

```
angular.module('website', [])
  .controller('MainController', ['$scope',
    function($scope) {
      $scope.votes = 0;

      $scope.incrementOne = function() {
        $scope.votes += 1;
      };

      $scope.decrementOne = function() {
        $scope.votes -= 1;
      };

      $scope.incrementTen = function() {
```

```
        $scope.votes += 10;
      };

      $scope.decrementTen = function() {
        $scope.votes -= 10;
      };

      $scope.incrementHundred = function() {
        $scope.votes += 100;
      };

      $scope.decrementHundred = function() {
        $scope.votes -= 100;
      };
    }
  ])
  .directive('vote', function() {
    return {
      restrict: 'E',
      templateUrl: 'vote.html',
      scope: {
        voteUp: '&',
        voteDown: '&',
        label: '@'
      }
    };
  });
```

The vote directive is fairly simple in that we have restricted it to an element node with `restrict: 'E'` and declared the template with `templateUrl: 'Vote.html'`.

We are creating isolated scope by defining an object i.e.: `{}` for the `scope` property on the directive definition object.

An attribute isolated scope is being created for the label property with `label: '@'` which means we are creating a uni-directional binding from the parent scope to the directive to tell us what the `label` attribute evaluates to. This is appropriate because we only want to read the property and we never need to set it.

Expression isolated scope is being created for `voteUp` and `voteDown` with `voteUp: '&'` and `voteDown: '&'`. What this means is that when we call `voteUp` or `voteDown` in our directive, it will call whatever method we defined in that attribute on our directive.

For instance, because we have defined `incrementHundred()` in our `vote-up` attribute in the code below, when we call `voteUp` on our directive it will call

`incrementHundred()` on the parent scope.

> AngularJS converts camel-case to snake-case when moving from JavaScript to HTML. That is why we are using `voteUp` in our JavaScript and `vote-up` in the HTML.

And for our directive markup, we have an anchor tag that calls `voteUp` on `ng-click`, a div that displays the `label` value and another anchor tag to call `voteDown`.

```
<div class="vote-set">
  <a href="#" class="icon-arrow-up" ng-click="voteUp()"></a>
  <div class="number">{{label}}</div>
  <a href="#" class="icon-arrow-down" ng-click="voteDown()"></a>
</div>
```

And now we have created a vote directive that we are using in three places to increment the vote count with different denominations.

Isolated Expression Scope with Variables

Occasionally we need to pass a value back to the parent controller as a parameter on the expression we have bound to and so this is what we are going to tackle next.

In our previous example we have defined 6 methods to handle the increment and decrement operations but that seems a little verbose. Wouldn't it be nice if we could consolidate those methods and just send in the value that we want to increment or decrement the vote count by?

In the HTML below, we have set things into motion by defining an `incrementVote` and `decrementVote` method that accepts `unit` as a parameter.

```
<div class="container" ng-controller="MainController">
  <h1>Current Votes: {{votes}}</h1>
  <hr>
  <h3>Isolated Expression Scope with Variables</h3>
  <div class="vote-container">
    <vote label="9" vote-up="incrementVote(unit)"
      vote-down="decrementVote(unit)"></vote>
    <vote label="6" vote-up="incrementVote(unit)"
      vote-down="decrementVote(unit)"></vote>
    <vote label="3" vote-up="incrementVote(unit)"
```

```
      vote-down="decrementVote(unit)"></vote>
  </div>
</div>
```

And in the template markup below, we are going to call `voteUp` and `voteDown` and with a value for the parent controller.

```
<div class="vote-set">
  <a href="#" class="icon-arrow-up"
    ng-click="voteUp({unit:label})"></a>
  <div class="number">{{label}}</div>
  <a href="#" class="icon-arrow-down"
    ng-click="voteDown({unit:label})"></a>
</div>
```

> The most important thing to keep in mind when passing variables with isolated expressions is that we do not pass them one-by-one but with a parameter object.

We are going to use the value we defined for the `label` property as our `unit` value. And in the `MainController`, we will take the the `unit` value and cast it to an integer with `parseInt(unit,10)` and add or subtract it to `$scope.votes`.

```
angular.module('website', [])
  .controller('MainController', ['$scope',
    function($scope) {
      $scope.votes = 0;

      // ...

      $scope.incrementVote = function(unit) {
        $scope.votes += parseInt(unit,10);
      };

      $scope.decrementVote = function(unit) {
        $scope.votes -= parseInt(unit,10);
      };
    }
  ])
```

Isolated Expression Scope with Multiple Methods

We are going to wrap things up with one more neat thing that you can do with isolated expression scope. We can actually bind more than one method to a single expression on the directive.

For the sake of illustration, we are going to add a vote directive that increments and decrements the vote count by 1000. When this happens, we want to call a second method called `alertAuthorities` because clearly something is amiss!

```
<div class="container" ng-controller="MainController">
  <h1>Current Votes: {{votes}}</h1>
  <hr>
  <h3>Isolated Expression Scope with Multiple Methods</h3>
  <div class="vote-container">
    <vote label="1000"
      vote-up="incrementVote(1000);alertAuthorities();"
      vote-down="decrementVote(1000);alertAuthorities();">
      </vote>
  </div>
  <hr>
  <div ng-if="alert" class="alert alert-danger">{{alert}}</div>
</div>
```

We are showing an alert `div` if there is an `alert` defined on `$scope` which we set in the code below.

```
angular.module('website', [])
  .controller('MainController', ['$scope',
    function($scope) {
      $scope.votes = 0;

      // ...

      $scope.incrementVote = function(unit) {
        $scope.votes += parseInt(unit,10);
      };

      $scope.decrementVote = function(unit) {
        $scope.votes -= parseInt(unit,10);
      };

      $scope.alertAuthorities = function() {
        $scope.alert = 'The authorities have been alerted!';
      };
    }
  ])
```

By calling `voteUp` or `voteDown` on this particular `vote` directive, we are not only calling `incrementVote` or `decrementVote` but `alertAuthorities` as well.

In Conclusion

We started out with a simple isolated expression scope example and then extended it to pass values from the directive to the parent controller. And then to wrap things up, we learned how we could actually call more than one method on a parent controller using expression isolated scope.

This snippet was written by Lukas Ruebbelke (aka @simpulton) and edited by Ari Lerner (aka @auser).

Chapter 23

Conclusion

We hope you enjoyed our mini Angular cookbook and can't wait to see what you create with these snippets!

For more in-depth Angular coverage, check out our 600+ page ng-book. It covers these type of snippets and much much more, including authentication, optimizations, validations, building chrome apps, security, SEO, the **most** detailed testing coverage available, and much much more!

Stay tuned to ng-newsletter for our new screencast series where we'll feature highly detailed angular recipes with both back-end support and detailed walkthroughs of complete, professional applications.

Finally, enjoy our weekly newsletter: the best, hand-curated Angular information delivered to your inbox once a week.

Cheers!