AngularJS by Google

http://angularjs.org

Mark Volkmann

mark@ociweb.com

Overview ...

- Single-page application (SPA) focus
  - update parts of page rather than full page refresh
- Makes HTML more **declarative**
  - results in less code to write
  - adds many elements and attributes to HTML (called directives)
    - more can be added
  - associations between input fields and model properties are more clear
    - `<div ng-controller="PlayerCtrl">`
    - `  <input type="text" ng-model="player.name"/>`
  - associations between user actions and event handling are more clear
    - `<button ng-click="createPlayer()">Create</button>`

W3C “Web Components” also supports custom elements

- 2-way data binding
  - automatically synchronizes models and views, so less DOM manipulation is needed
  - doesn't require custom model classes and properties like EmberJS, Backbone.js and Knockout

jQuery makes DOM manipulation easier. AngularJS makes DOM manipulation unnecessary.

... Overview ...

- Encouraged application structure
  - templates, controllers, services, directives, filters

each of these terms is described later

- History support through routes
  - browser back and forward buttons work
- Dependency injection
  - for making services available to controllers, directives, filters, and other services
  - also for mock testing of services
- Form validation
  - based on HTML form validators
  - inputs can be required, have min/max lengths, have a type, and be compared to regular expressions
  - currently supported HTML5 input types include **`email`**, **`number`**, **`text`** and **`url`**
    - unsupported types include **`color`**, **`date`**, **`month`**, **`range`**, **`tel`** and **`time`**

... Overview

- Recommended testing tools
  - Karma (previous called Testacular) test runner
    - created by AngularJS team
    - runs tests across set of configured browsers
    - can perform headless testing using PhantomJS
    - can watch files for changes and automatically rerun tests
    - runs on NodeJS; install with **`npm install -g karma`**
  - supported test frameworks are Jasmine, Mocha and QUnit
- Supported browsers
  - desktop: Safari, Chrome, Firefox, Opera, IE8 and IE9
  - mobile: Android, Chrome Mobile and iOS Safari

History

- Original developed in 2009 by Misko Hevery and Adam Abrons
  - to support a commercial product for online storage of JSON data
  - at domain GetAngular.com
  - named after angle brackets used in HTML tags
- Introduction at Google
  - Misko worked on an internal Google project that was using GWT
  - he rewrote all the code developed over six months using GWT in three weeks using AngularJS
  - team agreed code was much shorter and clearer
  - later Google began funding further development of AngularJS
- Misko still works at Google where he continues AngularJS development
  - other Google employees now working on AngularJS include Ingor Minar and Vojta Jina

jqLite

- A light version of jQuery that has the same API
- Included with AngularJS
- Used by AngularJS if jQuery is not available
    - not pulled in by a **`script`** tag
- See http://docs.angularjs.org/api/angular.element
    - summarizes jQuery methods supported by jqLite
    - **`addClass`**, **`after`**, **`append`**, **`attr`**, **`bind`**, **`children`**, **`clone`**, **`contents`**, **`css`**, **`data`**, **`eq`**, **`find`**, **`hasClass`**, **`html`**, **`next`**, **`on`**, **`off`**, **`parent`**, **`prepend`**, **`prop`**, **`remove`**, **`removeAttr`**, **`removeClass`**, **`removeData`**, **`replaceWith`**, **`text`**, **`toggleClass`**, **`triggerHandler`**, **`unbind`**, **`val`**, **`wrap`**
    - many of these methods do not support selectors
- **`angular.element(`*`val`*`)`**
    - creates and returns a jQuery object
      (or jqLite object if jQuery wasn't included)
      that wraps a given DOM element or
      a new element created from an HTML string

Widgets

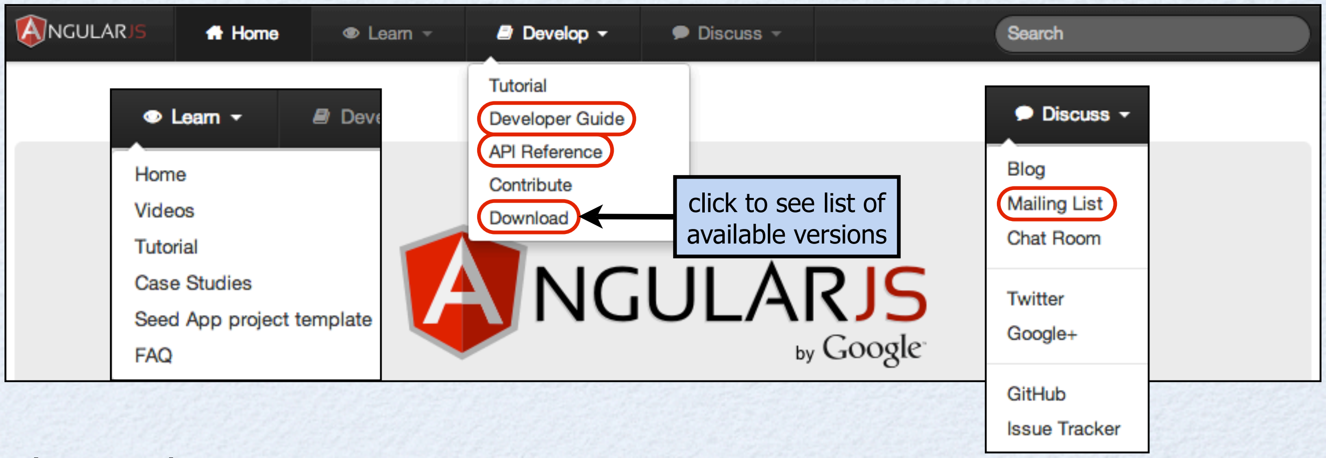
- AngularJS does not provide widgets
- There are third-party, open source widgets that integrate with AngularJS apps
  - for example, AngularUI - http://angular-ui.github.io
- Can use widgets from Twitter Bootstrap
  - and create custom directives for AngularJS integration
  - http://getbootstrap.com - see “Components” and “JavaScript” tabs
  - examples include dropdowns, navbars, breadcrumbs, alerts, progress bars, modal dialogs, tabs, tooltips and carousels

Styles

- AngularJS does not provide CSS styles
- Consider using Twitter Bootstrap
    - http://getbootstrap.com
- Does add CSS classes to some elements that can be customized
    - input elements get these classes
        - **`ng-pristine`** and **`ng-dirty`** - based on whether they ever had a value
        - **`ng-valid`** and **`ng-invalid`** - based on validation tests

Documentation / Help

- Home page
  - http://angularjs.org



click to see list of available versions

- Video tutorials from John Lindquest
  - http://www.egghead.io

- Mailing List
  - a Google Group at https://groups.google.com/group/angular

Terminology Overview

- **Controllers** - manage the "scope" of an element and those nested inside it
  by adding data and methods to it;
  the data and methods are used in directives and binding expressions
- **Services** - provide business logic and data access (ex. Ajax REST calls to access databases)
  to controllers and other services
- **Directives** - extend HTML syntax by adding elements and attributes that can manipulate the DOM;
  can provide a view Domain-Specific Language (DSL)
- **Binding Expressions** - evaluate to the value of scope expressions
  that can access scope data and call scope methods
- **Filters** - format, filter and sort data to be rendered;
  typically specified in directives and binding expressions
- **Routes** - map URL paths to templates and views
- **Templates** - snippets of HTML that can replace the content of a view
- **Views** - sections of a page whose content can be replaced

2-Way Data Binding Basics

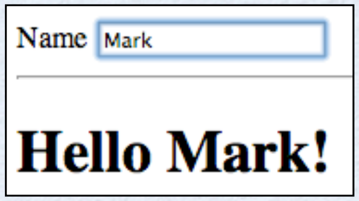
- Controllers add data and methods to `$scope` as properties
- HTML elements select a controller
  - with `ng-controller` attribute
  - in effect for the element and its descendants
- Form elements bind to `$scope` properties
  - with `ng-model` attribute
- Binding expressions access data and methods
  - on `$scope` of their controller
  - expressions in double-curly braces `{{ }}` that appear in HTML attribute values and element content
- When JavaScript code changes value of a `$scope` property ...
  - form elements that refer to them with `ng-model` are updated
  - binding expressions that refer to them are updated
- When user changes value of an form element ...
  - `$scope` property referred to by the `ng-model` attribute is updated

Hello Example

```
<!DOCTYPE html>
<html ng-app>
  <head>
    <script
      src="https://ajax.googleapis.com/ajax/libs/angularjs/1.0.7/angular.min.js">
    </script>
  </head>
  <body>
    <div>
      <label>Name</label>
      <input type="text" ng-model="yourName" placeholder="Enter a name here">
      <hr>
      <h1>Hello {{yourName}}!</h1>
    </div>
  </body>
</html>
```

CDN versions are here

no custom JavaScript code required in this example; no app module, controllers or services


Name Mark

Hello Mark!

Modules ...

- Divide application code into logical groupings
  - services, controllers, directives and filters are scoped to a module
- Can test each module separately
- Can have one module for entire application
  - common for small applications
- Consider creating at least one module for each of the following
  - service definitions
  - directive definitions
  - filter definitions
  - controller definitions and other application logic
    - this module will depend on the previous modules

... Modules

- To create a module
  - `var module = angular.module('name', [module-dependencies]);`
  - if the module doesn't depend on other modules,
    pass empty array for second argument
- To retrieve a module
  - `var module = angular.module('name');`
  - note that only one argument is passed
  - useful for defining the
    services, controllers, directives and filters
    of a module in multiple source files

Apps

- An app is an instance of a module
- To create an app, create a module whose name is the app name
    - app names are typically camel-case with first letter uppercase
- Specify the app name in the **`ng-app`** attribute on an element in the main HTML file
    - typically on **`html`** tag in **`index.html`**
    - can only have one per HTML document
    - typically only one per application
        - only the first is automatically processed

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

```
<html ng-app="Todo">
  ...
</html>
```

Controllers ...

- Contain business logic for a single view (portion of a page)
  - not DOM manipulation, view data formatting, data filtering, or sharing data across controllers

use services to share
data between controllers

- Add data and methods to **`$scope`**
  - for use in model bindings (**`ng-model`** attribute),
    binding expressions **`{{`*`expression`*`}}`**
    and event handling directives (ex. **`ng-click`**)
- Can depend on services and invoke their methods
- Controllers are not singletons
  - controller functions are invoked each time the associated view is rendered
  - local variables in controller functions are lost, so state cannot be maintained in them

... Controllers

- To create a controller
  - `app.controller('name', function (services) { ... });`
  - controller names are typically camel-case with first letter uppercase, ending with "`Ctrl`"
  - ***`services`*** is a list of service names this controller uses
    as separate parameters, not an array
  - services are provided via dependency injection
- To use a controller
  - add **`ng-controller`** attribute to an HTML element

```
app.controller('TodoCtrl', function ($scope) {
  ...
});
```

```
<div ng-controller="TodoCtrl">
  ...
</div>
```

Directives

- Custom HTML elements, attributes and CSS classes that provide AngularJS functionality
- Many are provided
- Can add more

```
<input type="text" ng-model="todoText"/>

<button ng-click="addTodo()" ng-disabled="!todoText">Add</button>

<ul>
  <li ng-repeat="todo in todos"/>
    <input type="checkbox" ng-model="todo.done"/>
    <span class="done-{{todo.done}}">{{todo.text}}</span>
    <button ng-click="deleteTodo(todo)">Delete</button>
  </li>
</ul>
```

Provided Directives ...

only listing those that are commonly used

- **`ng-app`** - typically on **`html`** tag of main page
- **`ng-controller`** - allows binding expressions and other directives
  within this element and descendants
  to access the **`$scope`** properties of this controller
- **`ng-model`** - binds the value of a form element to a **`$scope`** property
- **`ng-repeat`** - creates multiple elements (ex. **`li`** or **`tr`** elements)
  from each element in an array or each property in an object
- **`ng-options`** - similar to **`ng-repeat`**, but creates **`option`** elements in a **`select`** element
- **`ng-show`**, **`ng-hide`** - conditionally shows or hides an element based on a scope expression
- **`ng-pattern`** - validates text input against a regular expression
- **`ng-disabled`** - conditionally disables a form element based on a scope expression
- **`ng-include`** - includes an HTML template (a.k.a. partial)

... Provided Directives

- **`ng-switch`** - conditionally changes the content of a given element based on a scope expression
  - also see **`ng-switch-when`** and **`ng-switch-default`** directives
- **`ng-view`** - marks a place where templates can be inserted by **`$routeProvider`** service
- **`ng-click`**, **`ng-change`** - general event handling
- **`ng-keydown`**, **`ng-keypress`**, **`ng-keyup`** - keyboard event handling
- **`ng-mousedown`**, **`ng-mouseenter`**, **`ng-mouseleaave`**,
  `ng-mousemove`, **`ng-mouseover`**, **`ng-mouseup`** - mouse event handling

Binding Expressions

- Specified in HTML using `{{...}}`
- Provide an expression that references properties and calls methods on scopes of current controllers
- Renders expression value

some binding expressions appeared on the "Directives" slide ... here are two more

```
<div>
  {{getUncompletedCount()}} of {{todos.length}} remaining
</div>
```

Scopes

- Provide access to data and methods within the scope of a controller
- Exist in a hierarchy
    - lookup starts at scope of controller and proceeds upward through scope hierarchy
- Each has a unique id stored in its **`$id`** property
- **`$scope`**
    - a provided service that can be injected into controllers to access current scope
    - to add data, `$scope.name = value;`
    - to add a method, `$scope.fnName = function (...) { ... };`
    - to get parent scope, `$scope.parent()`
- **`$rootScope`**
    - a provided service that can be injected into controllers to access root scope
    - topmost common ancestor of all scopes
    - putting data here is a discouraged way of sharing data across controllers; better to use a service

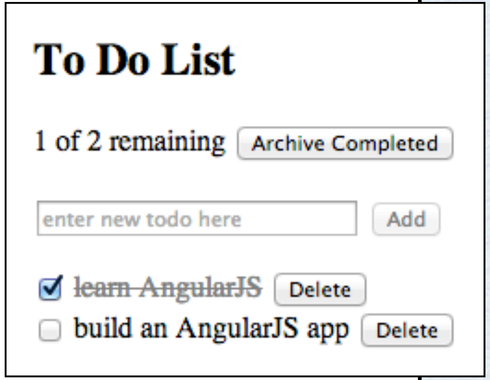
Todo List App ...

index.html

```
<!DOCTYPE html>
<html ng-app="Todo">
  <head>
    <link rel="stylesheet" href="todo.css"/>
    <script src=".../angular.min.js"></script>
    <script src="todo.js"></script>
  </head>
  <body>
    <h2>To Do List</h2>
    <div ng-controller="TodoCtrl">
      <div>
        {{getUncompletedCount()}} of {{todos.length}} remaining
        <button ng-click="archiveCompleted()">Archive Completed</button>
      </div>
      <br/>
      <form>
        <input type="text" ng-model="todoText" size="30"
          placeholder="enter new todo here"/>
        <button ng-click="addTodo()" ng-disabled="!todoText">Add</button>
      </form>

      <ul class="unstyled">
        <li ng-repeat="todo in todos"/>
          <input type="checkbox" ng-model="todo.done"/>
          <span class="done-{{todo.done}}">{{todo.text}}</span>
          <button ng-click="deleteTodo(todo)">Delete</button>
        </li>
      </ul>
    </div>
  </body>
</html>
```

Wrapping this in a **form** causes the button to be activated when the input has focus and the return key is pressed.



todo.css

```
body {
  padding-left: 10px;
}

ul.unstyled {
  list-style: none;
  margin-left: 0;
  padding-left: 0;
}

.done-true {
  text-decoration: line-through;
  color: gray;
}
```

... Todo List App

todo.js

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

app.controller('TodoCtrl', function ($scope) {
  $scope.todos = [
    {text: 'learn AngularJS', done: true},
    {text: 'build an AngularJS app', done: false}
  ];

  $scope.addTodo = function () {
    $scope.todos.push({text: $scope.todoText, done: false});
    $scope.todoText = ''; // clears input
  };

  $scope.archiveCompleted = function () {
    // Not saving completed todos in this version.
    $scope.todos = $scope.todos.filter(function (t) { return !t.done; });
  };

  $scope.deleteTodo = function (todo) {
    $scope.todos = $scope.todos.filter(function (t) { return t !== todo; });
  };

  $scope.getUncompletedCount = function () {
    var count = 0;
    angular.forEach($scope.todos, function (todo) {
      if (!todo.done) count++;
    });
    return count;
  };
});
```

no custom services
are needed
in this example

Services

- Provide functions to controllers, directives, filters, and other services via dependency injection
- Services are singletons
  - only a single instance of each is created
  - can maintain state in them
- There are many provided services and custom services can be created
  - custom service names are typically camel-case with first letter lowercase, ending with "`Svc`"

Some Provided Services

- **`$http`** - sends HTTP requests; details on next slide
- **`$routeProvider`** - maps URL paths to templates and where they should be inserted (in **`ng-view`** elements)
- **`$location`** - **`path`** method is used to change to a path specified with **`$routeProvider`**
- **`$log`** - writes log messages to browser console for debugging
  - has **`log`**, **`info`**, **`warn`**, and **`error`** methods (**`debug`** will be added in 1.2)
  - a no-op when browser doesn't have **`console`** object (ex. IE8 when Developer Tools is not open)
- **`$q`** - promise/deferred implementation based on "Q" promise library
  - used internally and can be used in your code
- **`$rootScope`** and **`$scope`** - described earlier
- **`$timeout`** - wrapper around **`window.setTimeout`**
  - returns a promise that is resolved after the timeout is reached
- **`$exceptionHandler`** - handles uncaught exceptions

`$q` Service

- AngularJS promise service based on popular “q” library
  - see https://github.com/kriskowal/q
  - AngularJS implementation supports a subset of “q” methods
- An alternative to passing callback functions to asynchronous functions
  - like Ajax requests

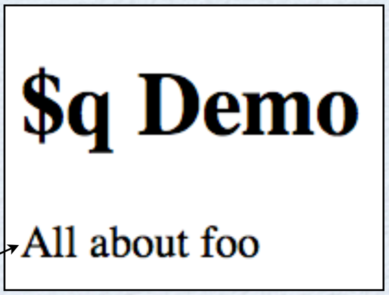
Using **`$q`**

- In async function
  - inject **`$q`** into service that defines function
  - create a deferred object - **`var dfr = $q.defer();`**
  - call async function
    - on success - **`dfr.resolve(result);`**
    - on error - **`dfr.reject(err);`**
    - optionally give progress notifications - **`dfr.notify(msg);`**
      - can call any number of times
  - return promise - **`return dfr.promise;`**
- In caller
  - call async function to get promise
  - call **`then`** on promise,
    passing it callback functions for success, error and notify
    in that order
    - success callback is passed value that was passed to **`dfr.resolve`**
    - error callback is passed value that was passed to **`dfr.reject`**

$q Example ...

index.html

```
<!DOCTYPE html>
<html ng-app="MyApp">
  <head>
    <script src=".../angular.min.js"></script>
    <script src="promise.js"></script>
  </head>
  <body>
    <h1>$q Demo</h1>
    <div ng-controller="MyCtrl" ng-bind="title"></div>
  </body>
</html>
```



$q Demo

All about foo

foo.html

```
<html>
  <head>
    <title>All about foo</title>
  </head>
  <body>
    Foo is a made up word.
    Related words include bar and baz.
  </body>
</html>
```

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

app.controller('MyCtrl', function ($scope, mySvc) {
  // Can't use a cross-domain URL unless
  // server includes a header to allow it.
  var url = 'foo.html';

  mySvc.getTitle(url).then(
    function (title) { $scope.title = title; },
    function (err) { alert(err); });
});
```

... $q Example

```
app.factory('mySvc', function ($http, $q) {
  var svc = {};

  // Gets the title from the HTML document at the given URL.
  svc.getTitle = function (url) {
    var dfr = $q.defer();

    $http.get(url).then(
      function (res) {
        // Get title text from response HTML using DOM XML parser and XPath.
        var parser = new DOMParser();
        // Chrome and Safari do not yet support 'text/html'.
        var doc = parser.parseFromString(res.data, 'text/xml');
        var title = doc.evaluate(
          '/html/head/title/text()', doc, null, XPathResult.STRING_TYPE);
        dfr.resolve(title.stringValue);
      },
      function (err) {
        dfr.reject(err.data);
      });

    return dfr.promise;
  };

  return svc;
});
```

Chaining Promises

- The **then** method returns a promise
  which allows chaining promises to be resolved sequentially
- If any promise in the chain is rejected,
  the remainder of the chain will not be evaluated
  and the final error callback will be invoked
- For example, add this to **mySvc**

```
svc.addOne = function (value) {
  var dfr = $q.defer();
  $timeout(function () {
    dfr.resolve(value + 1);
  }, 100);
  return dfr.promise;
};
```

$timeout must be injected into **mySvc**

- Call it three times like this

```
mySvc.addOne(10)
  // The function passed to then on the next line returns a promise.
   .then(function (result) { return mySvc.addOne(result); })
  // This ia another way to write the previous line.
  .then(mySvc.addOne.bind(null))
  // Process the final result which will be 13.
  .then(function (result) {
    console.log('result =', result);
  });
```

in this example, the same service method is called multiple times, but they could be calls to different methods

`$http` Service

- Sends HTTP requests
  - `$http(`*`options`*`)` returns a `$q` promise with two additional HTTP-specific methods, `success` and `error`
  - example: `$http({method: `*`method`*`, url: `*`url`*`}).success(`*`successCb`*`).error(`*`errorCb`*`);`
  - for `post` and `put`, can set `data` option; objects are automatically serialized to JSON
  - JSON response bodies are automatically deserialized to JavaScript values
  - other options include `params`, `headers`, `cache` and `timeout`
  - *`successCb`* and *`errorCb`* functions are passed
    `data`, `status`, `headers` and `config` as separate parameters in that order
- Shorthand methods on `$http` also return a `$q` promise
  - `get`, `head`, `post`, `put`, `delete`, `jsonp`
  - example: `$http.get(`*`path`*`, `*`options`*`).success(`*`successCb`*`).error(`*`errorCb`*`);`
  - example: `$http.post(`*`path`*`, `*`data`*`, `*`options`*`).success(`*`successCb`*`).error(`*`errorCb`*`);`
- Can enable caching of HTTP requests
  - `$http.get(`*`path`*`, {cache: true}).success(`*`successCb`*`).error(`*`errorCb`*`);`
  - see http://pseudobry.com/power-up-%24http.html for more detail

see examples of using this service later

`$http` Promises

- Service methods that use **`$http`** can simply return the result which is a promise

```
svc.getContent = function (url) {
  return $http.get(url);
};
```

- Callers get results from the promise in two ways
- 1) **`success`** and **`error`** methods

```
mySvc.getContent(url)
  .success(function (data) { console.log('content =', data); },
  .error(function (err) { alert(err); });
```

- 2) **`then`** method

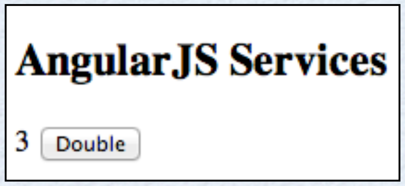
```
mySvc.getContent(url).then(
  function (res)  console.log('content =', res.data); },
  function (res) { alert(res.data); });
```

`res` is a response object that contains the properties **`status`**, **`headers`**, **`data`** and **`config`**

Custom Services

- **Five ways to define a service**
- 1) **Constant** - provide primitive value or object that only contains data
  - `module.constant(name, constant);`
- 2) **Value** - provide primitive value, function, or object that may contain functions
  - `module.value(name, value);`
- 3) **Factory** - provide function that returns a value,
  typically an object with functions or a single function
  - `module.factory(name, function (dependencies) { ... });`
- 4) **Constructor** - provide constructor function
  - `module.service(name, ctorFn);`
  - an instance will be created by calling `new ctorFn()`
- 5) **Provider** - provide function that supports service configuration and
  defines `$get` method that returns a configured object with function properties or a single function
  - `module.provider(name, providerFn);`

Custom Services Example ...



index.html

```
<!DOCTYPE html>
<html ng-app="Services">
  <head>
    <script src=".../angular.min.js"></script>
    <script src="services.js"></script>
  </head>
  <body>
    <h2>AngularJS Services</h2>
    <div ng-controller="MyCtrl">
      {{number}}
      <button ng-click="double()">Double</button>
    </div>
  </body>
</html>
```

... Custom Services Example ...

services.js

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

// This example shows five ways to define a service.

// #1: Using a constant
// This is for primitives and objects that only contain data, not functions.
app.constant('startingValue', 3);

// #2: Using a primitive, function or object (may contain functions) value
// Services can't be injected into these functions.
app.value('doubleIt', function (n) { return n * 2; });

// #3: Using a "factory" function
// Can inject services when defined this way (ex. $log).
app.factory('factorySvc', function ($log) {
  $log.log('factorySvc entered');
  var svc = {};
  svc.double = function (number) { return number * 2; };
  return svc;
});

// #4: Using a constructor function
// Can inject services when defined this way (ex. $log).
function Doubler($log) {
  $log.log('constructor entered');
  this.double = function (number) { return number * 2; };
}
app.service('ctorSvc', Doubler);
```

could just return the `double` function

... Custom Services Example ...

services.js

```
// #5: Using a "provider" function
// Can inject services when defined this way (ex. $log).
app.provider('configurableSvc', function () {
  this.multiplier = 2;
  this.setMultiplier = function (multiplier) {
    this.multiplier = multiplier;
  };
  this.$get = function ($log) {
    $log.log('$get entered');
    var m = this.multiplier;
    return {
      double: function (n) { return n * m; }
    };
  };
});

// The service "configurableSvcProvider"
// is created automatically by app.provider.
app.config(function (configurableSvcProvider) {
  configurableSvcProvider.setMultiplier(3);
});
```

becomes a misleading name

... Custom Services Example

services.js

```
// Each of the five types of services defined above
// is injected into this controller.
app.controller('MyCtrl',
  function ($scope, startingValue,
    doubleIt, factorySvc, ctorSvc, configurableSvc) {

  $scope.number = startingValue;

  $scope.double = function () {
    //$scope.number = doubleIt($scope.number);
    //$scope.number = factorySvc.double($scope.number);
    //$scope.number = ctorSvc.double($scope.number);
    $scope.number = configurableSvc.double($scope.number);
  };
});
```

Dependency Injection (DI)

- Each application has an "injector" that manages DI
  - automatically created during bootstraping
- DI is a mechanism for making services available to controllers, directives, filters, and other services
- Many of these are defined by a "factory function"
  - passed to these `Module` methods: `config`, `factory`, `directive`, `filter` and `run`
- Can identify service dependencies by name as parameters to these factory functions
  - see other ways on next slide

`run` method registers code to run after injector loads all modules

DI For Controllers

- Three ways to specify
- 1) match names of parameters to function passed to ***`module`*****`.controller`**
  - `app.controller('MyCtrl', function ($scope, playerSvc, gameSvc) { ... });`
  - most common approach
  - **doesn’t work with minimizers!**
- 2) set **`$inject`** property on controller

  -
    ```
    function MyCtrl($scope, pSvc, gSvc) { ... };
    MyCtrl.$inject = ['$scope', 'playerSvc', 'gameSvc'];
    ```
  - order of names in array matches order of parameters in function passed to **`app.controller`**
- 3) pass array to ***`module`*****`.controller`**

  -
    ```
    app.controller('MyCtrl', [
      '$scope', 'playerSvc', 'gameSvc',
      function ($s, pSvc, gSvc) { ... }
    ]);
    ```

Templates

- Snippets of HTML (partials) that are inserted in DOM via **`$routeProvider`** service
- Allows a page to be constructed from HTML that comes from multiple files
  - **`ng-include`** directive inserts content into its element
  - see next slide
- Allows a section of the page to replaced by different content
  - **`ng-view`** directive defines where content can be inserted
  - **`$routeProvider`** service maps URLs to content and a view
  - **`$location`** service **`path`** method changes the URL, triggering a route
  - see slides on routes

`ng-include` Directive

- Includes HTML content from another file
  - a “partial”
  - for example, each section of a page
    or the content of each tab on a page
- Can specify as an element
  - `<ng-include src="expression"/>`
- Can specify as an attribute
  - `<div ng-include="expression"></div>`
  - can be on an element other than `div`
  - have seen cases where functionality is broken with element form,
    but not with attribute form
- Can specify as a CSS class
  - but why do this?

if ***`expression`*** is a literal string path,
enclose it in single quotes
since it is treated as an expression
(can refer to scope properties)

Routes

- Used to change a part of the content displayed
- Define routes using **`$routeProvider`** service injected into function passed to **`app.config`** method
  - use **`when`** and **`otherwise`** methods to map path strings to route descriptions
  - route descriptions specify a **`templateUrl`** and a **`view`**
    - can also specify content in code with **`template`** property
    - **`view`** is where template content will be inserted, replacing previous content
  - **`otherwise`** is a catch-all that typically uses **`redirectTo`** property to utilize a **`when`** route
- Specify target locations of content with **`ng-view`** directive
- Navigate to a route using **`$location`** service
  - call **`path`** method to change path which triggers a defined route

Routes Example

`Module` `config` method
is passed a function
that is executed
when the module is loaded

```
app.config(function ($routeProvider) {
  $routeProvider
    .when('/baseball', {
      templateUrl: 'partials/baseball.html',
      view: 'center'
    })
    .when('/hockey', {
      templateUrl: 'partials/hockey.html',
      view: 'center'
    })
    .otherwise({
      redirectTo: '/baseball'
    });
});
```

```
$location.path('/hockey');
```

Routes With Data Loading

- Some views require data that must be loaded asynchronously
  - typically using Ajax to invoke REST services
- If the controller initiates loading the data ...
  - perhaps by calling a service method that uses the **`$http`** service
  - the page will render without the data
  - the data will be loaded
  - the parts of the page that use the data will render again
- To load the data before rendering the new view
  - specify **`controller`** and **`resolve`** properties in **`$routeProvider when`** method
  - resolve property value should be an object
    whose properties are promises that must be resolved
    before the view is rendered and the controller function is invoked
  - names of these properties should be injected into the controller
    so it can access the loaded data

Route Resolve Example ...

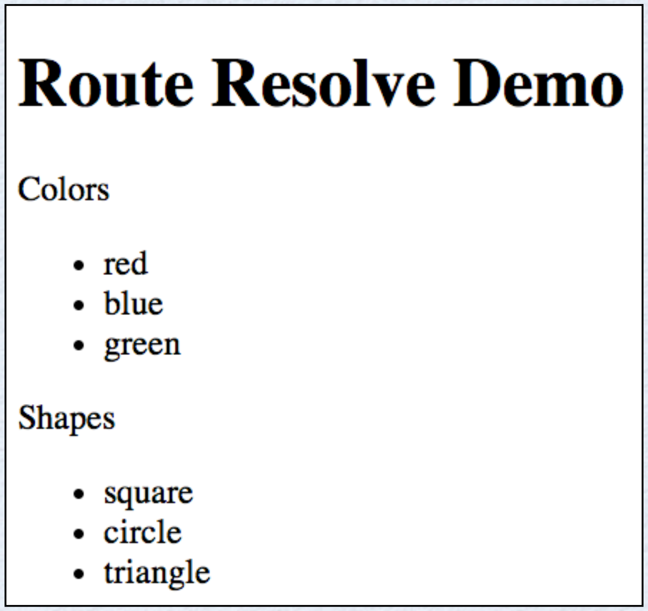
index.html

```html
<!DOCTYPE html>
<html ng-app="CatalogApp">
  <head>
    <script src=".../angular.min.js"></script>
    <script src="resolve.js"></script>
  </head>
  <body>
    <h1>Route Resolve Demo</h1>
    <div ng-view="center"></div>
  </body>
</html>
```

catalog.html

```html
<div>
  <label>Colors</labels>
  <ul>
    <li ng-repeat="color in colors">{{color}}</li>
  </ul>

  <label>Shapes</labels>
  <ul>
    <li ng-repeat="shape in shapes">{{shape}}</li>
  </ul>
</div>
```



Route Resolve Demo

Colors

- red
- blue
- green

Shapes

- square
- circle
- triangle

... Route Resolve Example ...

resolve.js

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

  app.factory('catalogSvc', function ($q, $timeout) {
    var svc = {};

    svc.getColors = function () {
      var dfr = $q.defer();
      // Simulate an async Ajax request.
      $timeout(function () {
        var colors = ['red', 'blue', 'green'];
        dfr.resolve(colors);
      }, 200);
      return dfr.promise;
    };

    svc.getShapes = function () {
      var dfr = $q.defer();
      // Simulate an async Ajax request.
      $timeout(function () {
        var colors = ['square', 'circle', 'triangle'];
        dfr.resolve(colors);
      }, 100);
      return dfr.promise;
    };

    return svc;
  });
```

... Route Resolve Example

resolve.js

```
  app.controller('CatalogCtrl', function ($scope, colors, shapes) {
    $scope.colors = colors;
    $scope.shapes = shapes;
  });

  var catalogResolve = {
    colors: function (catalogSvc) {
      return catalogSvc.getColors();
    },
    shapes: function (catalogSvc) {
      return catalogSvc.getShapes();
    }
  };

  app.config(function ($routeProvider) {
    $routeProvider
      .when('/catalog', {
        // controller must be specified here instead of in catalog.html
        controller: 'CatalogCtrl',
        templateUrl: 'catalog.html',
        view: 'center',
        resolve: catalogResolve
      })
      .otherwise({
        redirectTo: '/catalog'
      });
  });
})();
```

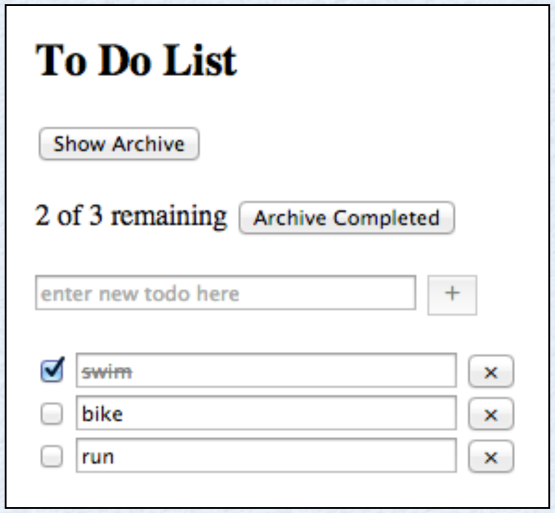
REST

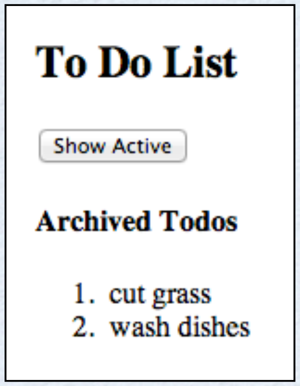
- Stands for "REpresentational State Transfer"
- A recipe for using HTTP to perform CRUD operations on "resources"
- Each resource has a URL
- HTTP verbs
  - `POST` to **create** a resource ("id" that will be assigned is not known)
  - `GET` to **retrieve** a resource
  - `PUT` to **update** or create a resource ("id" that will be assigned is known)
  - `DELETE` to **delete** a resource

JSON

- A data serialization format
  - alternative to XML and YAML
- Very similar to JavaScript literal objects
- Can represent objects, arrays, strings, numbers, booleans
- JavaScript values are easily serialized to JSON
- JSON is easily deserialized to JavaScript values
- Objects and arrays can be nested to any depth
- All object keys must be strings in double quotes
- Does not support reference cycles
- With REST, using JSON in HTTP POST and PUT bodies is common

REST Todo ...





The **HTTP server** that implements the **REST API** and serves static files for this example is written in **NodeJS**. To start it, enter "`node server.js`".

browse http://localhost:1919

Note the lack of DOM manipulation code in this example!

index.html

```
<!DOCTYPE html>
<html ng-app="Todo">
  <head>
    <link rel="stylesheet" href="todo.css"/>
    <script src=".../angular.min.js"></script>
    <script src="todo.js"></script>
  </head>
  <body>
    <h2>To Do List</h2>
    <button ng-controller="TodoCtrl" ng-click="changeView()">
      Show {{otherView}}
    </button>
    <div ng-view="main"></div>
  </body>
</html>
```

todo.css

```
body {
  padding-left: 10px;
}

ul.unstyled {
  list-style: none;
  margin-left: 0;
  padding-left: 0;
}

.todo-text {
  width: 200px;
}

.done-true {
  text-decoration: line-through;
  color: gray;
}
```

... REST Todo ...

These are the **templates** that are inserted into the "main" **view** using `$routeProvider` and `$location`.

partials/active.html

```
<br/>
<div ng-controller="TodoCtrl">
  <div>
    {{getUncompletedCount()}} of {{getTotalCount()}} remaining
    <button ng-click="archiveCompleted()">Archive Completed</button>
  </div>
  <br/>
  <form>
    <input type="text" class="todo-text" ng-model="todoText"
      placeholder="enter new todo here"/>
    <button ng-click="createTodo()" ng-disabled="!todoText">&#xFF0B;</button>
  </form>

  <ul class="unstyled">
    <li ng-repeat="(id, todo) in todos">
      <input type="checkbox" ng-model="todo.done" ng-change="updateTodo(todo)"/>
      <input type="text" class="todo-text done-{{todo.done}}"
        ng-model="todo.text" ng-change="updateTodo(todo)">
      <button ng-click="deleteTodo(id)">&#xD7;</button>
    </li>
  </ul>
</div>
```

Wrapping this in a `form` causes the button to be activated when the input has focus and the return key is pressed.

＋

X

partials/archive.html

```
<div ng-controller="TodoCtrl">
  <h4>Archived Todos</h4>
  <ol>
    <li ng-repeat="todo in archive | objToArr |orderBy:'text'"/>
      {{todo.text}}
    </li>
  </ol>
</div>
```

... REST Todo ...

todo.js

```
var app = angular.module('Todo', []);
var urlPrefix = 'http://localhost:1919/todo';

app.config(function ($routeProvider) {
  $routeProvider
    .when('/archive', {
      templateUrl: 'partials/archive.html',
      view: 'main'
    })
    .when('/active', {
      templateUrl: 'partials/active.html',
      view: 'main'
    })
    .otherwise({
      redirectTo: '/active'
    });
});

app.factory('todoSvc', function ($http) {
  var svc = {};

  function errorCb() {
    alert('Error in todoSvc!');
  }

  svc.archive = function (id, cb) {
    $http.post(urlPrefix + '/' + id + '/archive')
     .success(cb).error(errorCb);
  };
```

service used by **TodoCtrl** on next slide

todo.js

```
  svc.create = function (todo, cb) {
    var options =
      {headers: {'Content-Type': 'application/json'}};
    $http.post(urlPrefix, todo, options)
      .success(cb).error(errorCb);
  };

  svc['delete'] = function (id, cb) {
    $http['delete'](urlPrefix + '/' + id)
      .success(cb).error(errorCb);
  };

  svc.retrieve = function (id, cb) {
    $http.get(urlPrefix + '/' + id)
      .success(cb).error(errorCb);
  };

  svc.retrieveActive = function (cb) {
    $http.get(urlPrefix)
      .success(cb).error(errorCb);
  };

  svc.retrieveArchive = function (cb) {
    $http.get(urlPrefix + '/archive')
      .success(cb).error(errorCb);
  };

  svc.update = function (todo, cb) {
    $http.put(urlPrefix + '/' + todo.id, todo)
     .success(cb).error(errorCb);
  };

  return svc;
}); // end of call to app.factory
```

not necessary to set Content-Type since it detects this

... REST Todo ...

the controller

todo.js

```
app.controller('TodoCtrl', function ($scope, $location, todoSvc) {
  $scope.otherView = 'Archive';

  // Load active todos.
  todoSvc.retrieveActive(function (todos) {
    $scope.todos = todos;
  });

  // Load archived todos.
  todoSvc.retrieveArchive(function (archive) {
    $scope.archive = archive;
  });

  $scope.archiveCompleted = function () {
    Object.keys($scope.todos).forEach(function (id) {
      var todo = $scope.todos[id];
      if (todo.done) {
        todoSvc.archive(id, function () {
          // Remove todo from active UI.
          delete $scope.todos[id];
          // Add todo to archive UI.
          $scope.archive[id] = todo;
        });
      }
    });
  };

  $scope.changeView = function () {
    $location.path('/' + $scope.otherView.toLowerCase());
    $scope.otherView =
      $scope.otherView === 'Archive' ? 'Active' : 'Archive';
  };
```

... REST Todo ...

todo.js

the controller

```
  $scope.createTodo = function () {
    var todo = {text: $scope.todoText, done: false};
    todoSvc.create(todo, function (resourceUrl) {
      // Get id assigned to new todo from resource URL.
      var index = resourceUrl.lastIndexOf('/');
      todo.id = resourceUrl.substring(index + 1);

      $scope.todos[todo.id] = todo; // add todo to active UI
      $scope.todoText = ''; // clear input field
    });
  };

  $scope.deleteTodo = function (id) {
    todoSvc['delete'](id, function () {
      delete $scope.todos[id];
    });
  };

  $scope.getUncompletedCount = function () {
    var count = 0;
    // Iterate through object properties.
    angular.forEach($scope.todos, function (todo) {
      if (!todo.done) count++;
    });
    return count;
  };

  $scope.getTotalCount = function () {
    return $scope.todos ? Object.keys($scope.todos).length : 0;
  };

  $scope.updateTodo = function (todo) {
    todoSvc.update(todo, angular.noop);
  };
});
```

provided function that does nothing

... REST Todo

```
// The orderBy filter only works with arrays,
// not object property values.
// This is a custom filter that takes an object
// and returns an array of its property values.
// Use it before orderBy to sort object property values.
app.filter('objToArr', function() {
  return function (obj) {
    if (!angular.isObject(obj)) return [];
    return Object.keys(obj).map(function (key) {
      return obj[key];
    });
  };
});
```

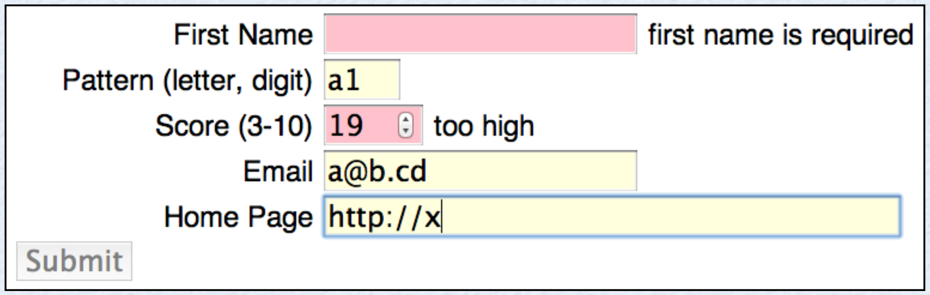
Form Validation ...

- Based on HTML5 input types
  - **`email`** - minimal matching example is “**`a@b.cd`**”
    - when invalid, sets ***`form-name`*****`.`*****`input-name`*****`.$error.email`**
  - **`number`** -
    - when invalid, sets ***`form-name`*****`.`*****`input-name`*****`.$number`** (broken!)
  - **`url`** - minimal matching example is “**`http://x`**”
    - when invalid, sets ***`form-name`*****`.`*****`input-name`*****`.$error.url`**
- Validation directives
  - **`required`** and **`ng-required`** -
    later supports binding to value so it can be turned on and off by changing a property
  - **`ng-minlength`** and **`ng-maxlength`** - to validate number of characters entered
  - **`min`** and **`max`** - for **`type="number"`**
  - **`pattern`** - to test against a regular expression

... Form Validation

- Form valid?
  - sets ***`form-name`*****`.$invalid`** to **`true`**
    if any input is invalid or any required input is missing
  - can use to enable/disable submit button

Validation Example



First Name		first name is required
Pattern (letter, digit)	a1	
Score (3-10)	19	too high
Email	a@b.cd	
Home Page	http://x	

validation.js

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

app.controller('MyCtrl', function ($scope) {
});
```

validation.css

```
body {
  font-family: sans-serif;
  font-size: 18pt;
}

button, input {
  font-size: 18pt;
}

label {
  display: inline-block;
  text-align: right;
  width: 250px;
}

/* ng-pristine class doesn't get removed
  when all content is deleted! */
input.ng-pristine {
  background-color: LightGreen;
}

input.ng-dirty {
  background-color: LightYellow;
}

input.ng-invalid {
  background-color: Pink;
}
```

... Validation Example ...

index.html

```
<!DOCTYPE html>
<html ng-app="MyApp">
  <head>
    <link rel="stylesheet" href="validation.css"/>
    <script src=".../angular.min.js"></script>
    <script src="validation.js"></script>
  </head>
  <body>
    <form name="myForm" ng-controller="MyCtrl">
      <div>
        <label>First Name</label>
        <input type="text" name="firstName" ng-model="firstName"
          ng-minlength="2" ng-maxlength="10" required/>
        <span ng-show="myForm.firstName.$error.min">too short</span>
        <span ng-show="myForm.firstName.$error.max">too long</span>
        <span ng-show="myForm.firstName.$error.required">first name is required</span>
      </div>

      <div>
        <label>Pattern (letter, digit)</label>
        <input type="text" name="letterDigit" size="2" ng-model="charDigit"
          ng-pattern="/^\\w\\d$/"/>
        <span ng-show="myForm.letterDigit.$error.pattern">regex fail</span>
      </div>
```

... Validation Example

index.html

```
      <div>
        <label>Score (3-10)</label>
        <input type="number" name="score" ng-model="score"
          min="3" max="10"/>
        <!-- The following should appear when a non-number is entered,
             but it is broken. -->
        <span ng-show="myform.score.$error.number">not a number</span>
        <span ng-show="myForm.score.$error.min">too low</span>
        <span ng-show="myForm.score.$error.max">too high</span>
      </div>

      <div>
        <label>Email</label>
        <input type="email" name="email" ng-model="email"/>
        <span ng-show="myForm.email.$error.email">invalid email</span>
      </div>

      <div>
        <label>Home Page</label>
        <input type="url" name="homePage" size="40" ng-model="homePage"/>
        <span ng-show="myForm.homePage.$error.url">invalid url</span>
      </div>

      <button ng-disabled="myForm.$invalid" type="submit">Submit</button>
    </form>
  </body>
</html>
```

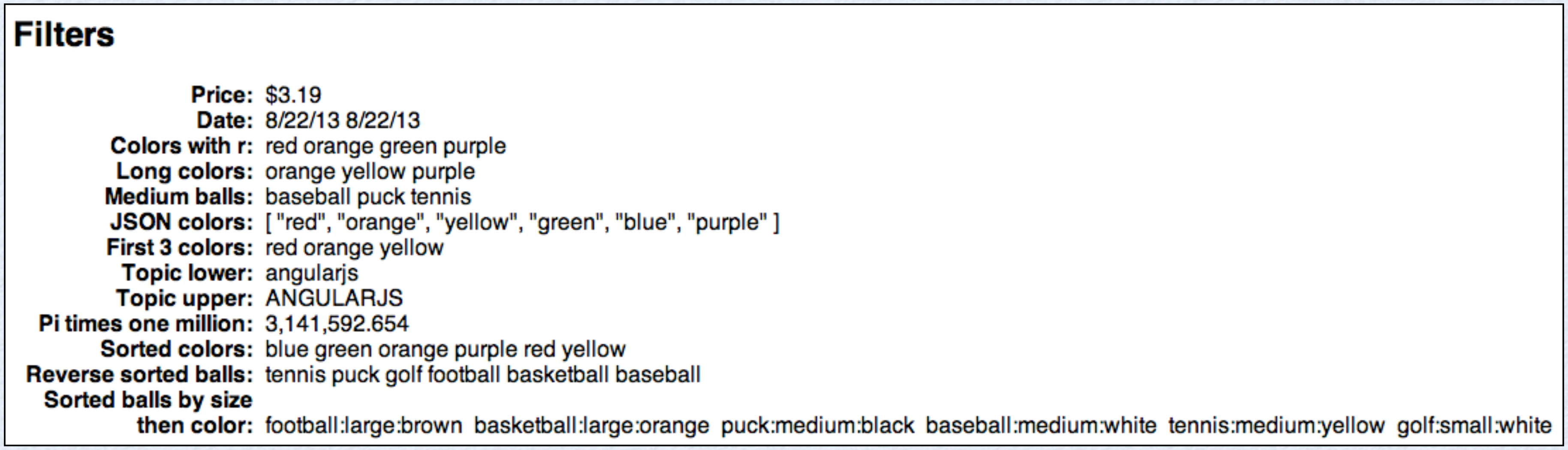
Filters

- Specified in binding expressions
  using the pipe character (|)
  to perform **formatting**, **filtering** and **sorting**
  of data to be rendered

Provided Filters

- Formatting
  - **`currency`** - for numbers; ex. $1,234.56
  - **`date`** - for Date objects, strings is recognized formats, and milliseconds since epoch; 1970-01-01T00:00:00Z
    can specify many formatting options
  - **`json`** - for any JavaScript value; typically used with objects and arrays for debugging
  - **`lowercase`**, **`uppercase`** - for strings
  - **`number`** - for numbers or numeric strings; rounds to given number of decimal places
    and adds comma separators for values of 1000 or more
- Filtering
  - **`filter`** - for arrays; reduces the number of elements processed; often used in **`ng-repeat`** directive
  - **`limitTo`** - for strings or arrays; processes first n characters or elements;
    often used with arrays in **`ng-repeat`** directive
- Sorting
  - **`orderBy`** - for arrays, not object properties; changes order in which elements are processed;
    often used in **`ng-repeat`** directive

Filter Examples



```
Filters

                 Price:  $3.19
                  Date:  8/22/13 8/22/13
         Colors with r:  red orange green purple
           Long colors:  orange yellow purple
          Medium balls:  baseball puck tennis
           JSON colors:  [ "red", "orange", "yellow", "green", "blue", "purple" ]
        First 3 colors:  red orange yellow
           Topic lower:  angularjs
           Topic upper:  ANGULARJS
   Pi times one million:  3,141,592.654
         Sorted colors:  blue green orange purple red yellow
   Reverse sorted balls:  tennis puck golf football basketball baseball
     Sorted balls by size
            then color:  football:large:brown  basketball:large:orange  puck:medium:black  baseball:medium:white  tennis:medium:yellow  golf:small:white
```

index.html

```html
<html ng-app="Filters">
  <head>
    <link rel="stylesheet" href="filters.css"/>
    <script src=".../angular.min.js"></script>
    <script src="filters.js"></script>
  </head>
  <body>
    <h2>Filters</h2>
```

filters.css

```css
body {
  font-family: sans-serif;
}

label {
  display: inline-block;
  font-weight: bold;
  margin-right: 5px;
  text-align: right;
  width: 170px;
}
```

... Filter Examples ...

index.html

```
<div ng-controller="FilterCtrl">
  <div>
    <label>Price:</label>
    <span>{{price | currency}}</span>
  </div>
  <div>
    <label>Date:</label>
    <span>{{now | date:'shortDate'}}</span>
    <span>{{now.getTime() | date:'shortDate'}}</span>
    <!-- can specify many formatting options after date: -->
  </div>
  <div>
    <label>Colors with r:</label>
    <span ng-repeat="color in colors | filter:'r'">
      <!-- make case insensitive with filter:'r':false -->
      {{color}}
    </span>
  </div>
  <div>
    <label>Long colors:</label>
    <span ng-repeat="color in colors | filter:longString">
      {{color}}
    </span>
  </div>
```

... Filter Examples ...

index.html

```
<div>
  <label>Medium balls:</label>
  <span ng-repeat="ball in balls | filter:{size: 'medium'}">
    {{ball.sport}}
  </span>
</div>
<div>
  <label>JSON colors:</label>
  <span>{{colors | json}}</span>
</div>
<div>
  <label>First 3 colors:</label>
  <span ng-repeat="color in colors | limitTo:3">
    {{color}}
  </span>
</div>
<div>
  <label>Topic lower:</label>
  <span>{{topic | lowercase}}</span>
</div>
<div>
  <label>Topic upper:</label>
  <span>{{topic | uppercase}}</span>
</div>
<div>
  <label>Pi times one million:</label>
  <span>{{bigPi | number:3}}</span>
</div>
```

... Filter Examples ...

index.html

```
      <div>
        <label>Sorted colors:</label>
        <span ng-repeat="color in colors | orderBy:identity">
          {{color}}
        </span>
      </div>
      <div>
        <label>Reverse sorted balls:</label>
        <span ng-repeat="ball in balls | orderBy:[-'sport']">
          {{ball.sport}}
        </span>
      </div>
      <div>
        <label>Sorted balls by size then color:</label>
        <span ng-repeat="ball in balls | orderBy:['size', 'color']">
          {{ball.sport}}:{{ball.size}}:{{ball.color}}&nbsp;
        </span>
      </div>
    </div>
  </body>
</html>
```

... Filter Examples

filters.js

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

app.controller('FilterCtrl', function ($scope) {
  $scope.price = 3.19;
  $scope.now = new Date();
  $scope.colors =
    ['red', 'orange', 'yellow', 'green', 'blue', 'purple'];
  $scope.topic = 'AngularJS';
  $scope.bigPi = Math.PI * 1e6;

  $scope.balls = [
    {sport: 'baseball', color: 'white', size: 'medium'},
    {sport: 'basketball', color: 'orange', size: 'large'},
    {sport: 'football', color: 'brown', size: 'large'},
    {sport: 'golf', color: 'white', size: 'small'},
    {sport: 'puck', color: 'black', size: 'medium'},
    {sport: 'tennis', color: 'yellow', size: 'medium'}
  ];

  $scope.longString = function (text) {
    return text.length > 5;
  };

  $scope.identity = angular.identity;
});
```

Watching for Scope Changes

- To watch for changes to a scope property
  - `$scope.$watch(`***`expression`***`,` ***`listenerFn`***`, [`***`objectEquality`***`]);`
  - ***`expression`*** is a JavaScript expression that returns the scope property to be watched
    - can be just the string name of a single scope property
    - can be a function that returns the name of the property to watch (reevaluated after every call to **`$digest`**)
    - Is there alternate string syntax that is supported to watch multiple properties?

when does AngularJS call **`$digest`**?

  - ***`listenerFn`*** is a function that is passed the new and old values
  - objectEquality is an optional boolean parameter
    - if true changes to objects are evaluated based on equality (same property values) rather than reference (same object)
- For scope properties with object or array values,
  can watch for changes to any top-level property of the value
  - `$scope.$watchCollection(`***`expression`***`, function (`***`newObj`***`,` ***`oldObj`***`) {`
  - `  ...`
  - `});`
- Watches are reevaluated after user events, XHR ready and **`setTimeout`** firings

Scope Events

- Can broadcast events to lower scopes in hierarchy
  - `$scope.$broadcast(eventName, args);`
- Can emit an event to higher scopes in hierarchy
  - `$scope.$emit(eventName, args);`
- Can listen for events from other scopes in hierarchy
  - `$scope.$on(name, listenerFn);`
  - `listenerFn` is passed an event object and an `args` array
  - returns a function that can be called to stop listening
  - event object has these properties
    - `name` - of the event
    - `targetScope` - scope that broadcasted or emitted the event
    - `currentScope` - scope in which `listenerFn` is running
    - `stopPropagation` - function to call to prevent further propagation of the event to other listeners
    - `preventDefault` - function to call to set `defaultPrevented` flag in event object to true
    - `defaultPrevented` - true if any listener called `preventDefault`; false otherwise

Custom Directives

- To create a new directive
  - ***module*.directive(*name*, *factoryFn*);**

Custom Filters

- To create a new filter
    - ***module*.filter(*name*, *factoryFn*);**

`angular` Properties ...

- `angular.bind(self, fn, args)`
  - same as `Function` `bind` in ES5; provided for non-ES5 environments
- `angular.copy(src, [dest])`
  - if *`dest`* is supplied, deep copies properties from *`src`* into *`dest`* and returns *`dest`*
  - otherwise creates a new object that is a deep copy of *`src`*
  - *`src`* and *`dest`* should be an object or array
  - if *`src`* is not an object or array, it is simply returned
- `angular.element(val)`
  - see earlier slide on jqLite
- `angular.equals(val1, val2)`
  - performs a deep equality test on objects and arrays
    or a normal equality test on other kinds of values
- `angular.extend(dest, src1, src2, ...)`
  - copies all properties from one or more *`src`* objects to a *`dest`* object

… `angular` Properties …

- **`angular.forEach(`*`obj, iterator, [context]`*`)`**
  - iterates through properties of ***`obj`***, passing them to ***`iterator`*** (value then key)
  - ***`context`*** is value of **`this`** inside ***`iterator`***
  - ***`obj`*** can be an object or array
- **`angular.fromJson(`*`v`*`)`**
  - like **`JSON.parse`**, but returns ***`v`*** if it is not a string
- **`angular.toJson(`*`v, [pretty]`*`)`**
  - like **`JSON.stringify`**, but optionally returns a pretty-printed string
- **`angular.identity(`*`args`*`)`**
  - returns its first argument; useful for sorting primitive arrays and other functional programming idioms
- **`angular.injector(`*`moduleArray`*`)`**
  - advanced feature related to dependency injection

… `angular` Properties …

- **`angular.isKind(`*`v`*`)`**
  - where ***`Kind`*** is **`Array`**, **`Date`**, **`Defined`**, **`Element`**, **`Function`**, **`Number`**, **`Object`**, **`String`**, or **`Undefined`**
  - the only value that is not "defined" is **`undefined`**
  - a value is an "element" if it is a DOM element or a jQuery object that wraps one
  - a value is an "object" if it an object or array (not **`null`** like JavaScript **`typeof`** reports)
- **`angular.lowercase(`*`v`*`)`**
  - returns a lowercase copy of ***`v`*** if it is a string and ***`v`*** otherwise
- **`angular.uppercase(`*`v`*`)`**
  - returns an uppercase copy of ***`v`*** if it is a string and ***`v`*** otherwise

... `angular` Properties

- **`angular.module(`*`name, dependencyArray`*`)`**
  - creates (if ***`dependencyArray`*** are passed) or retrieves (otherwise) a module
- **`angular.noop()`**
  - does nothing; useful in some functional programming idioms
- **`angular.version`**
  - object with properties that describe the version of AngularJS being used

Bootstrap Process

- When main HTML file for an AngularJS app is loaded in browser
  - walks DOM
  - finds **`ng-app`** which defines part of DOM managed by AngularJS
  - finds and evaluate all directives
    - creates scope hierarchy
    - associates directives with appropriate scope
    - adds a "watch" for all **`ng-model`** directives?
  - adds a "watch" for all binding expressions?
- For more detail, see the "Startup" section
  at http://docs.angularjs.org/guide/concepts
  - addresses **`$injector`**, **`$compile`** and **`$rootScope`** service

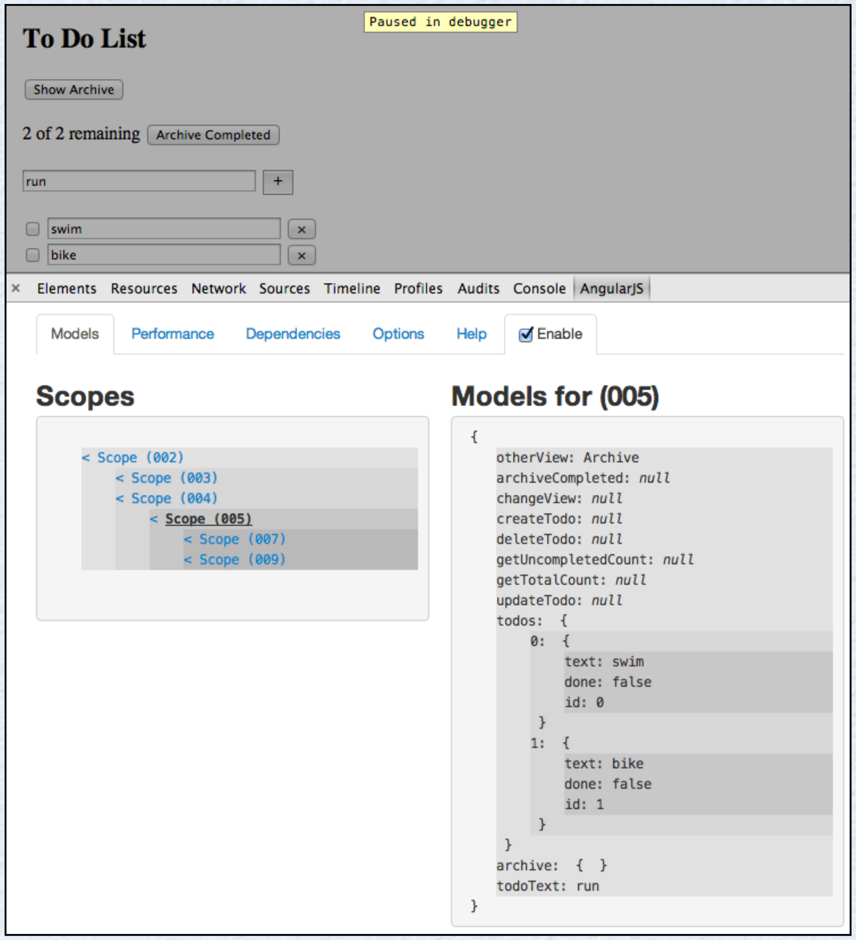
need more clarification on this!

`$scope.$apply()`

- Processes the **`$watch`** list
  - "set of expressions that many have changes since the last iteration"
    - runs registered listener functions when the value of their corresponding scope property changes
- Passed a function to be executed in the "Angular execution context" which takes note of scope property changes
- Called automatically when ...
- Only need to call this when
  - implementing custom event callbacks
  - working with third-party library callbacks
- Continues iterating through the watch list until no more listener functions need to be run
  - stops after a maximum of ten passes to prevent infinite loops
- Finally the browser DOM is updated to reflect changes to **`ng-model`** values

Debugging

- In addition to using browser supplied developer tools, the AngularJS team created Batarang, a Chrome extension
    - free from the Google Web Store
    - https://github.com/angular/angularjs-batarang
        - includes a video tutorial on using it
- Shows nested scopes and the properties on each
- Can modify scope properties
- Measures performance of service functions
    - useful for identifying bottlenecks


Paused in debugger
To Do List
Show Archive
2 of 2 remaining
Archive Completed
run
swim
bike
Elements Resources Network Sources Timeline Profiles Audits Console AngularJS
Models Performance Dependencies Options Help Enable
Scopes
Models for (005)

```
< Scope (002)
    < Scope (003)
    < Scope (004)
        < Scope (005)
            < Scope (007)
            < Scope (009)
```

```
{
    otherView: Archive
    archiveCompleted: null
    changeView: null
    createTodo: null
    deleteTodo: null
    getUncompletedCount: null
    getTotalCount: null
    updateTodo: null
    todos: {
        0: {
            text: swim
            done: false
            id: 0
        }
        1: {
            text: bike
            done: false
            id: 1
        }
    }
    archive: { }
    todoText: run
}
```

Testing

- Describe using Karma with Mocha

Mock Dependency Injection

- In test code, mock implementations of services can be used in place of real implementations
- One way to do this
  - create an object that has all the methods of the service
  - register that object under the name of the service to be replaced
    - `app.value(svcName, mockSvcObject);`
- Other ways?

Resources

- Main site
  - http://angularjs.org
- Year of Moo
  - http://www.yearofmoo.com
  - a blog on AngularJS written by contributor Matias Niemela
  - currently working on improving the official AngularJS documentation