AngularJS Framework

Trainer: Hai Tang

Course Objectives

At the end of the course, you'll be able to :

- Understanding AngularJS Framework
- Creating a single page web application (SPA)
- Developing new HTML vocabulary with AngularJS (Directive)
- Building end to end web application based on AngularJS
- Writing unit test for web application using AngularJS

Prerequisites

- Prerequisites:
  - Basic understanding of JavaScript and HTML/CSS
  - Hands on working with JavaScript

Assessment Disciplines

- Class Participation: 100%
- Exercise: 60%
- Final assignment (home work): 40%
- Passing Scores: 70%

Course Timetable

- Course Duration: 9 hrs
- Break 15 minutes

Set Up Environment

- To complete the course, your PC must install:
  - IDE (Bracket, Sublime 2, Visual Studio code…)
  - Chrome browser

Course Administration

- In order to complete the course you must:
  - Sign in the Class Attendance List
  - Participate in the course
  - Provide your feedback in the End of Course Evaluation

Further References

- AngularJS official site
  - https://docs.angularjs.org/guide
- Others
  - https://egghead.io/
  - https://github.com/angular-ui/ui-router/wiki

Agenda

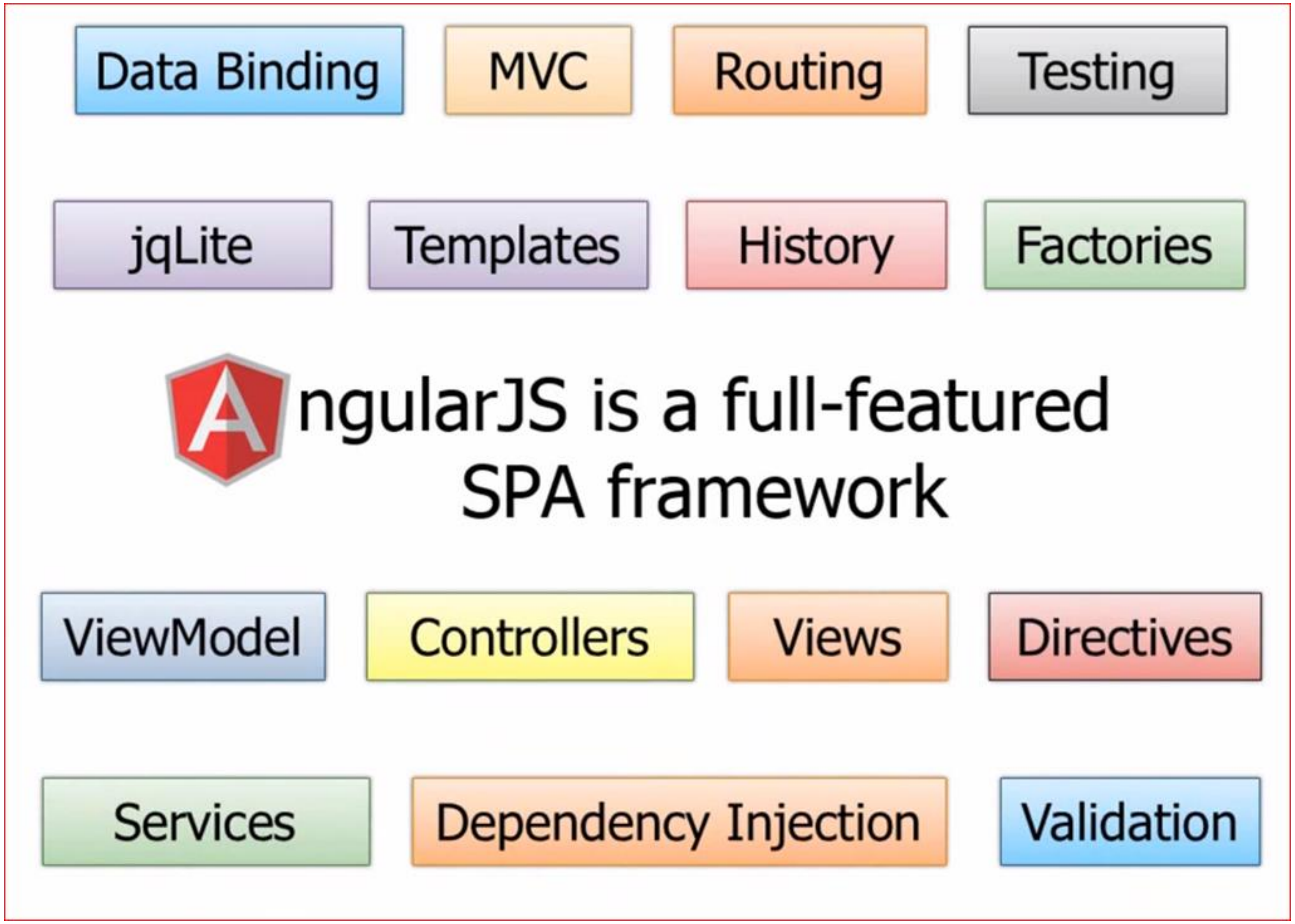
- Introduction to AngularJS
- Controllers and Markup
- Creating and Using Services
- Routing
- Creating custom Directives
- Unit Test

Introduction to AngularJS

What is AngularJS?

- JavaScript framework used for making SPA web applications
- It runs on plain JavaScript and HTML/CSS
- Developed in 2009 by Miško Hevery and Adam Abrons
- Maintained by Google

Full-featured SPA framework



Reference source: http://www.adevedo.com/

Why should we use AngularJS?

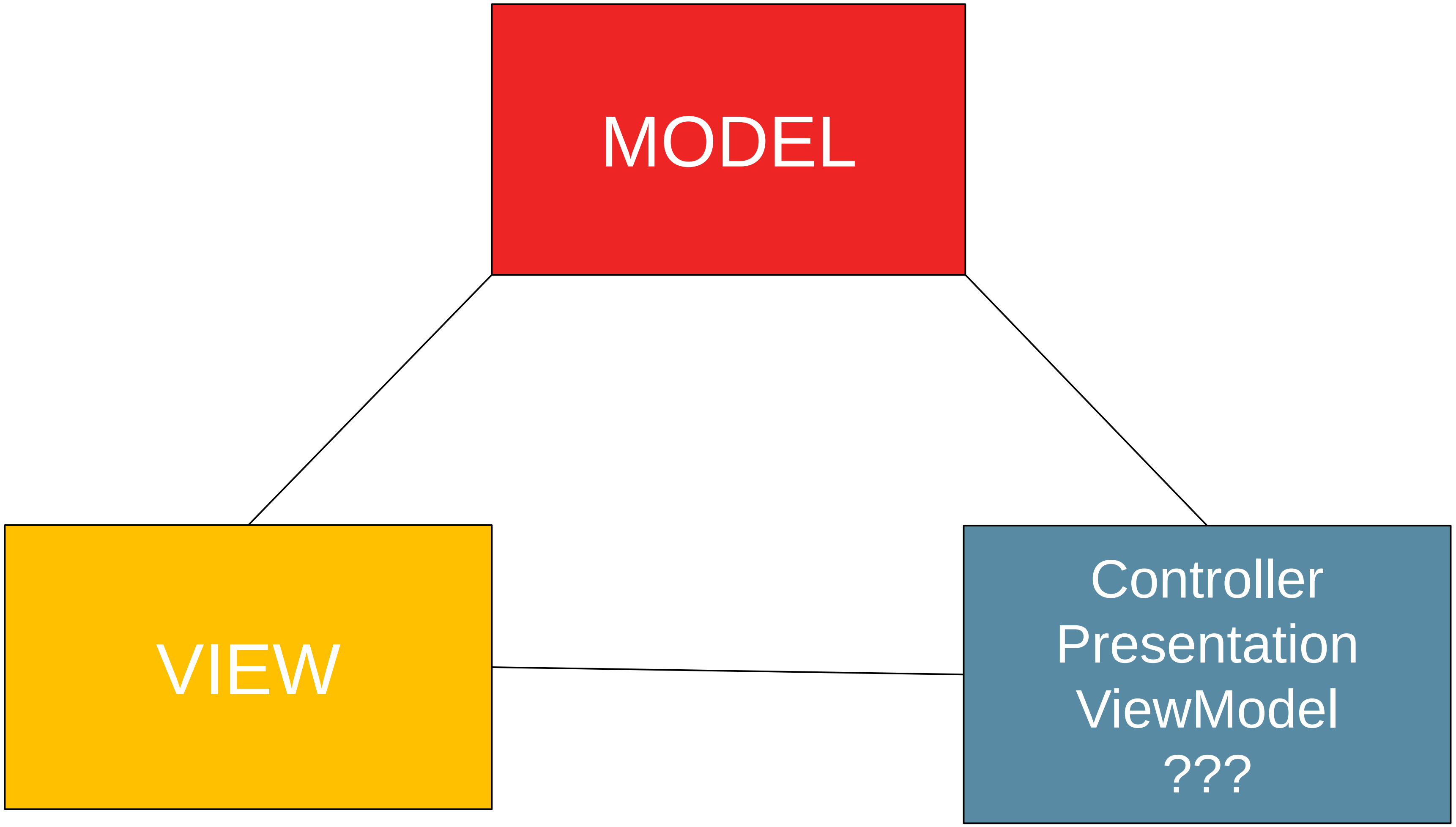
Open Source (Free)

MVW Framework

Extend HTML Vocabulary

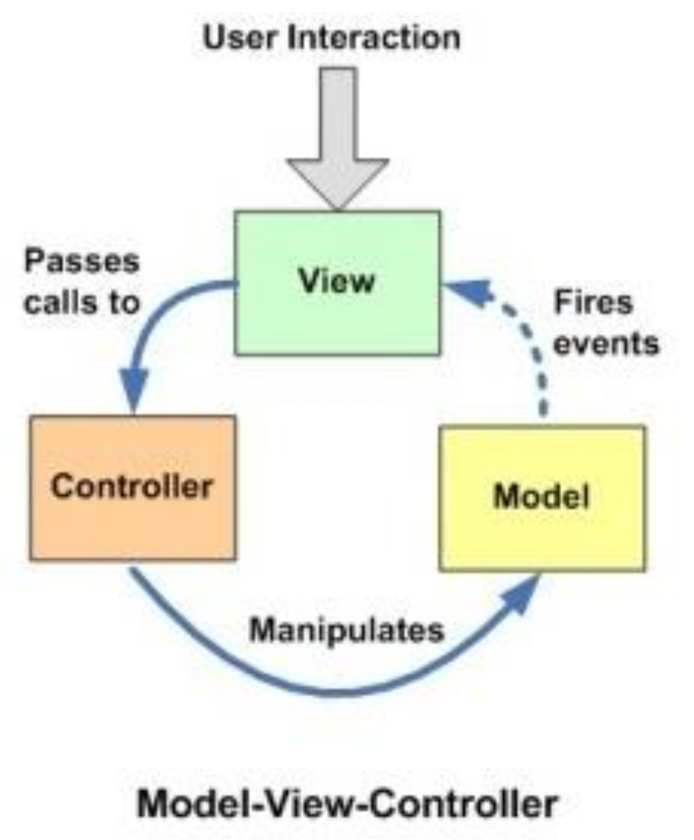
Testable

MVW Framework



Using MVC pattern

- MVC = Model-View-Controller
- Less dependencies
- Improves maintainability
- It is easier to read and understand code



Reference source: http://softwarearchitect123.com

Using MVC pattern

- Model
  - Holds the data
  - Notifies the View and the Controller for changes in the data
  - Stored in scope object.
- View
  - Displays stuff (buttons, labels, …) what your users will see
  - Knows about the Model
  - Sync with the model
- Controller
  - Controls everything
  - Knows about both the Model and the View
  - The “logic” resides in it
  - What to happen, when and how

Using MVC pattern

Model:

```
{  “name": “World"   }
```

View:

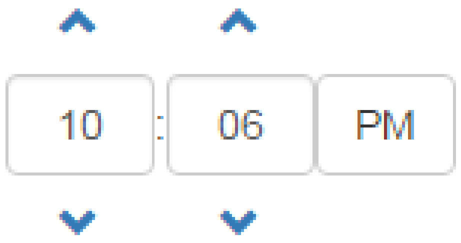
```
<div ng-controller="HelloController">
  Name:
  <input type="text" ng-model="name" />

  <h1>Hello {{ name }}</h1>

</div>
```

Controller:

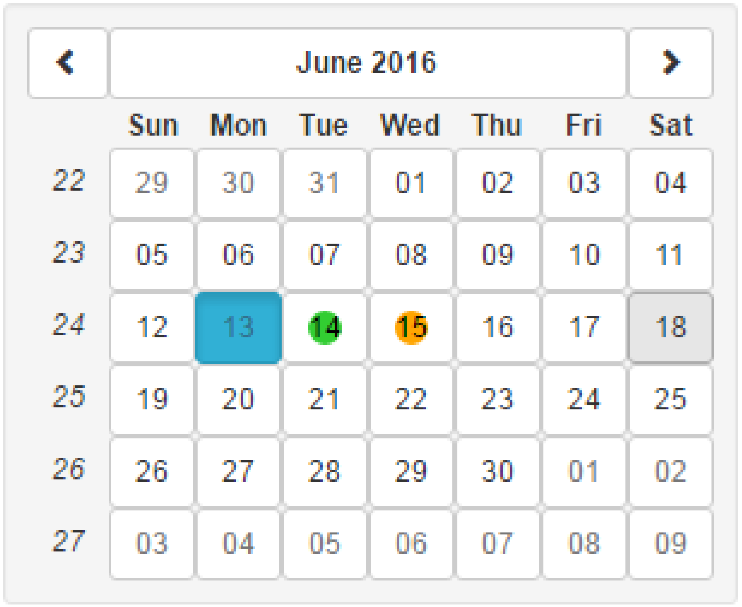
```
app.controller('HelloController', function($scope){
    $scope.name = 'World';
});
```

Extend HTML Vocabulary

```
<uib-timepicker> </uib-timepicker>
```



```
<uib-datepicker class="well well-sm" datepicker-options="options">
</uib-datepicker>
```

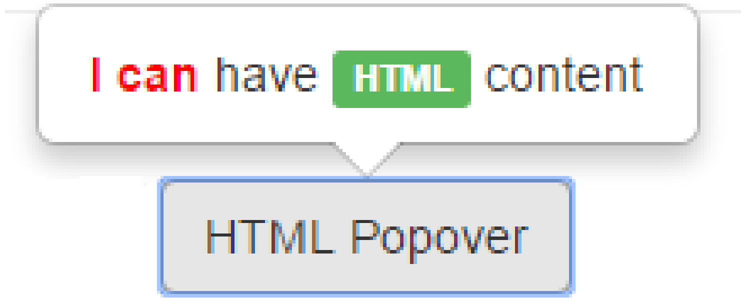


June 2016

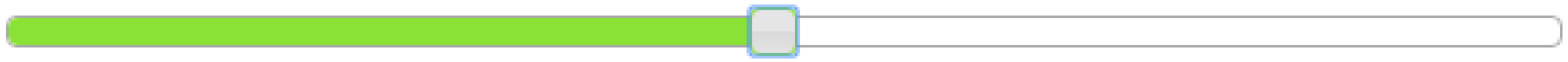
	Sun	Mon	Tue	Wed	Thu	Fri	Sat
22	29	30	31	01	02	03	04
23	05	06	07	08	09	10	11
24	12	13	14	15	16	17	18
25	19	20	21	22	23	24	25
26	26	27	28	29	30	01	02
27	03	04	05	06	07	08	09

Extend HTML Vocabulary

```
<button uib-popover-html="htmlPopover" class="btn btn-default">
        HTML Popover
</button>
```

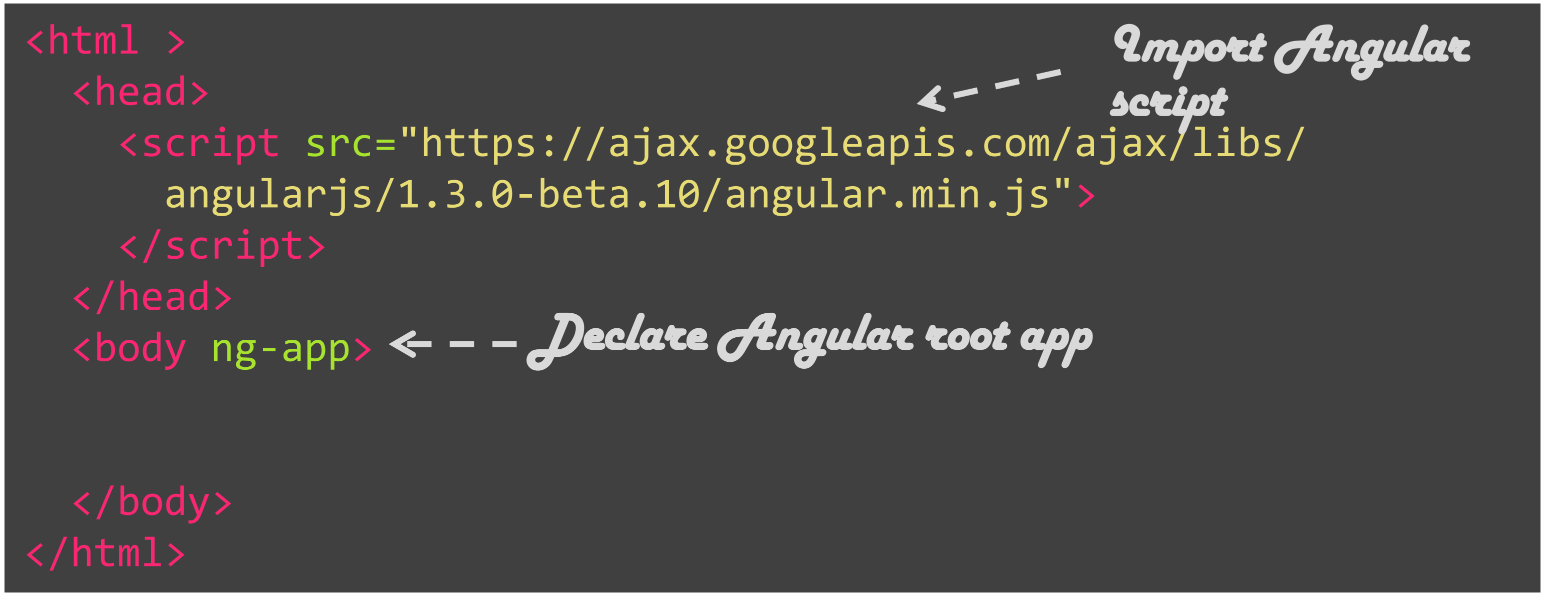


```
<div id="green" ui-slider="colorpicker.options“> </div>
```



How to apply AngularJS

- Import Angular script
- Declare Angular root app



```
<html >
  <head>
    <script src="https://ajax.googleapis.com/ajax/libs/
      angularjs/1.3.0-beta.10/angular.min.js">
    </script>
  </head>
  <body ng-app>


  </body>
</html>
```

Practice Exercise 1

Bootstrap AngularJS

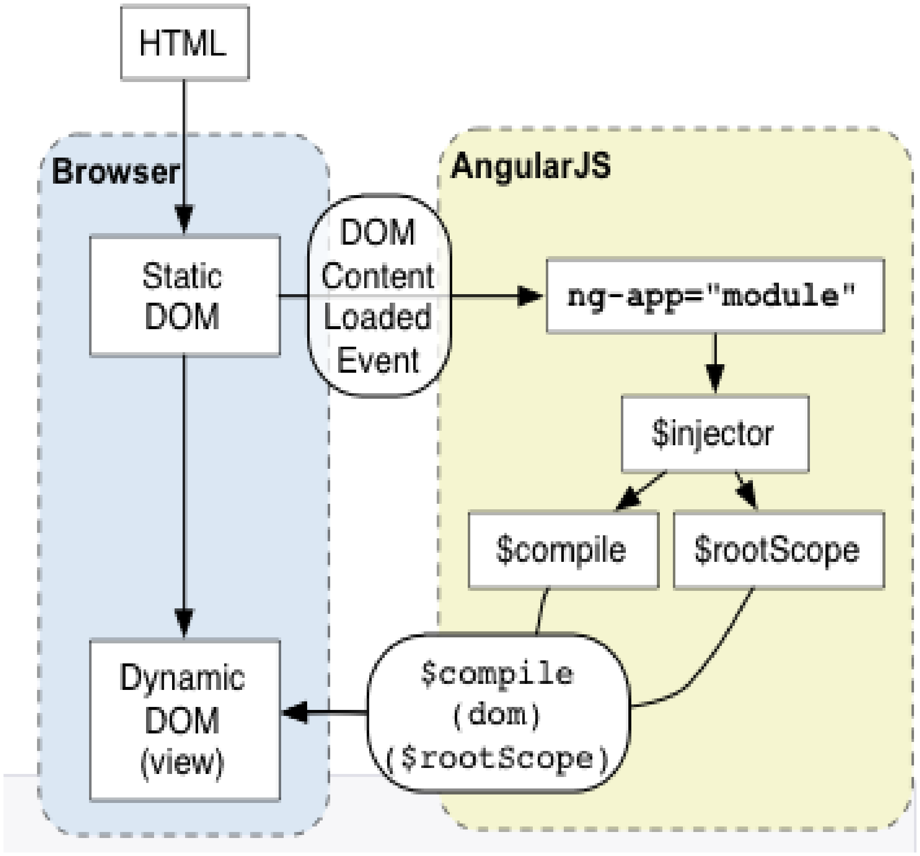
- Two ways: automatic, manually
- Can add controllers, services, directives, etc after an application bootstraps.

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

  <div id="content" ng-app="myApp"
  ...
  </div>

...

  </body>
</html>
```

```
angular.module('myApp', [])
.controller('MyController', function
($scope) {
  $scope.greetMe = 'World';
}]);

angular.element(document).ready(function(){
  angular.bootstrap(document, ['myApp']);
});
```

Bootstrap AngularJS

1. Input: Angular loads the module associated with the directive
2. Creates the application $injector
3. The $injector that will be used for dependency injection is created.
4. The injector will then create the root scope that will become the context for the model of our application.
5. Output: Angular will then "compile" the DOM starting at the ngApp root element, processing any directives and bindings found along the way.

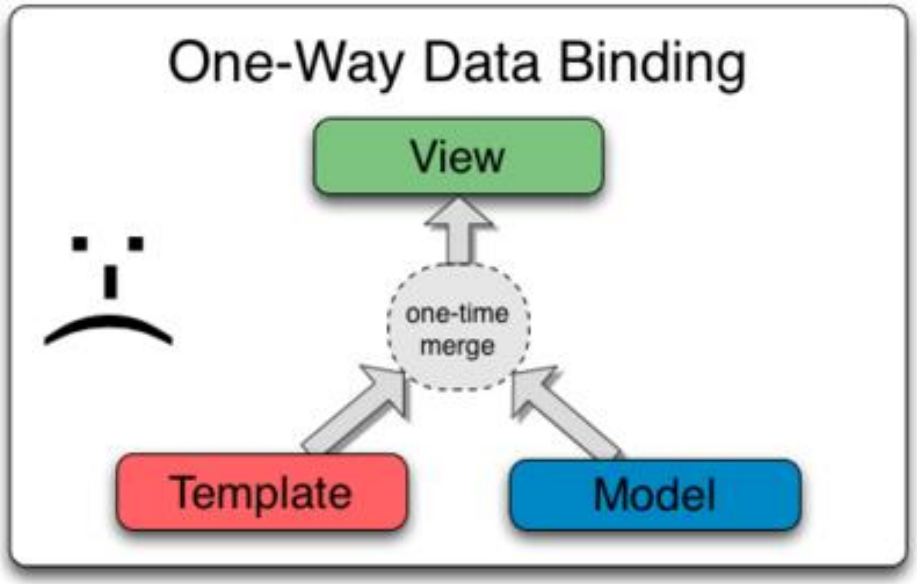


Reference source: http://angularjs.org

Controllers and Markup

Data Binding

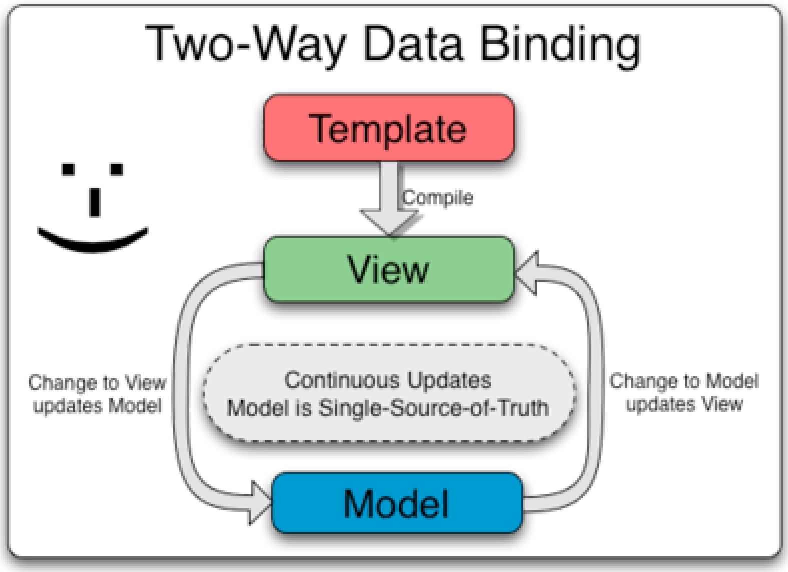
- Automatic synchronization of data between the model and view
- One-way data binding
  - Merge Template and Model into a View one time
  - Need to code to reflect change in Model to View and vice versa



Reference source: http://angularjs.org

Data Binding

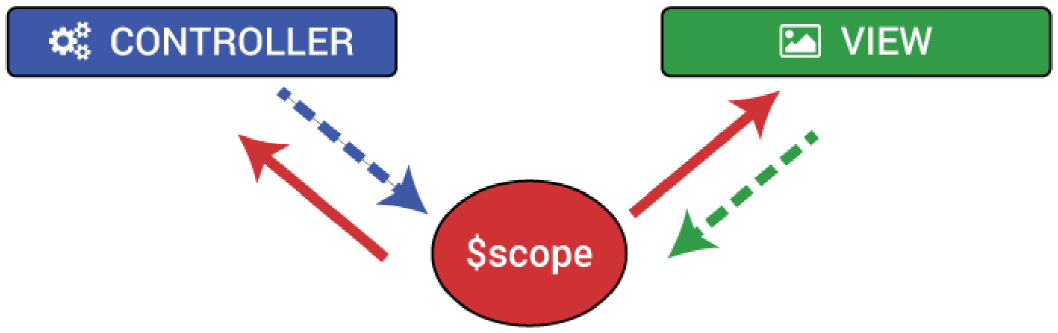
- Two way Data Binding (AngularJS way)
  - View is updated automatically when the Model is changed
  - Model is updated automatically when a value in the View has changed
  - No DOM manipulation boilerplate needed! The model is the single source of truth.


Two-Way Data Binding
Template
Compile
View
Change to View updates Model
Continuous Updates Model is Single-Source-of-Truth
Change to Model updates View
Model

Reference source: http://angularjs.org

Controllers

- Define the application’s behavior
- Controller create its new $scope
- Set up properties/behaviors of the $scope object.
- Controllers use $scope to expose controller’s methods to view



Creating Controllers

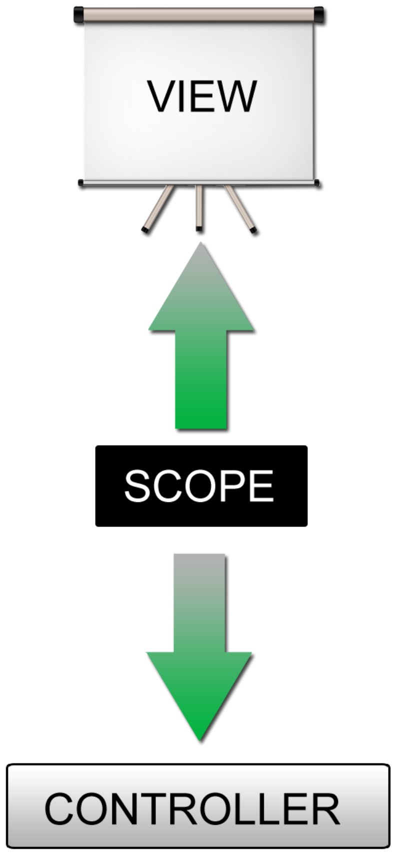
```
myApp.controller('MyController', function($scope) {
  $scope.naomi = { name: 'Naomi',
                   address: '1600 Amphitheatre'
                 };
  $scope.addNumber = function(x, y) {
    return x + y;
  };
});
```

Binding Controller to View by ng-controller

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

        <p>{{ addNumber(3, 5) }}</p>

        <p>{{ naomi.name }}</p>
</div>
```

Scopes

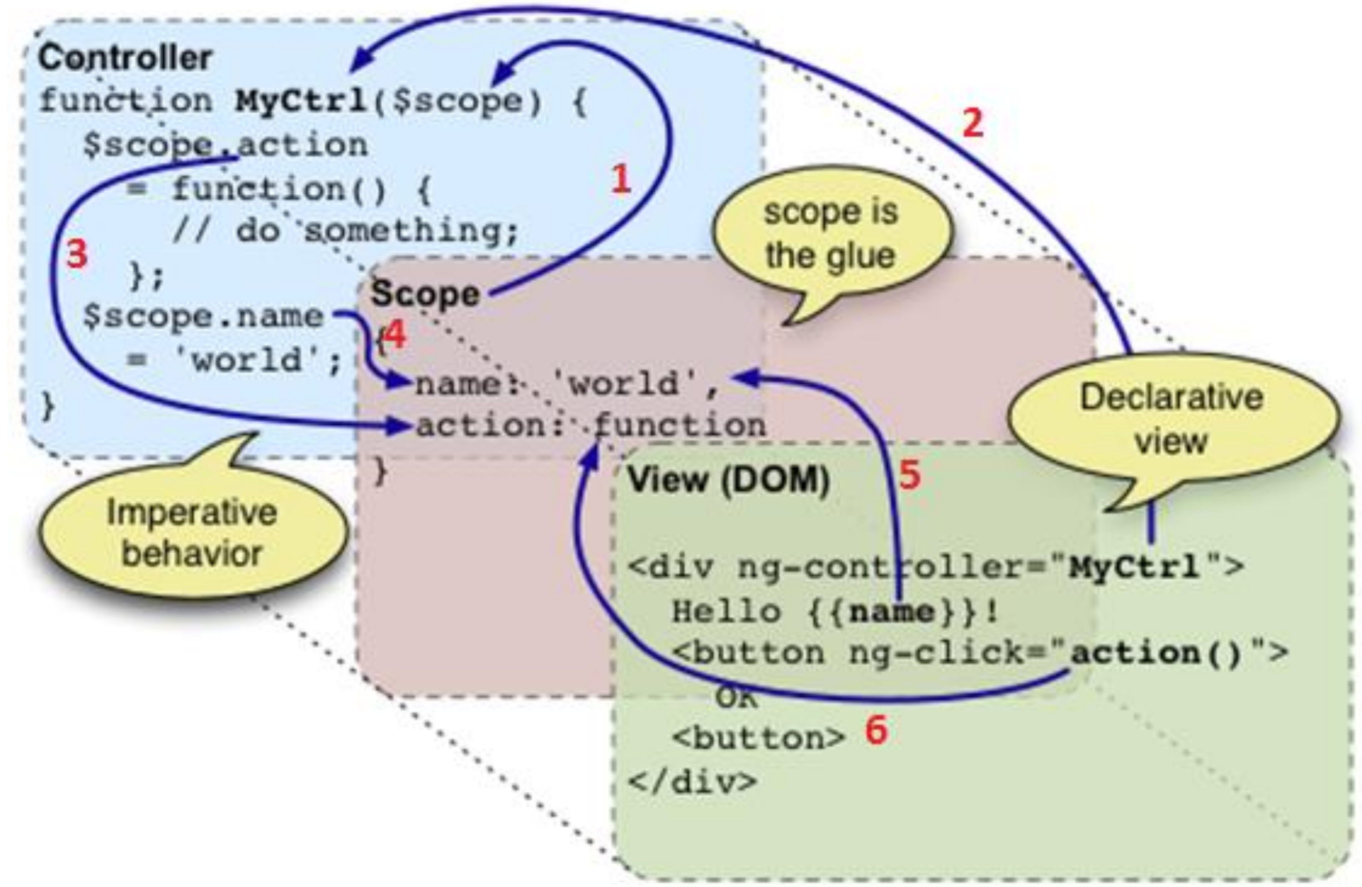
- $scope is an object that refers to the Model.
- A hash of key/values
- One scope for each controller
- The GLUE between the View and the Controller
- An execution context for expressions
- Controllers and directives have reference to the scope but not to each other



Reference source: http://angularjs.org

Scopes

- An execution context for expressions
- Controllers and directives have reference to the scope but not to each other



```
function MyCtrl($scope) {
  $scope.action
    = function() {
      // do something;
    };
  $scope.name
    = 'world';
}
```

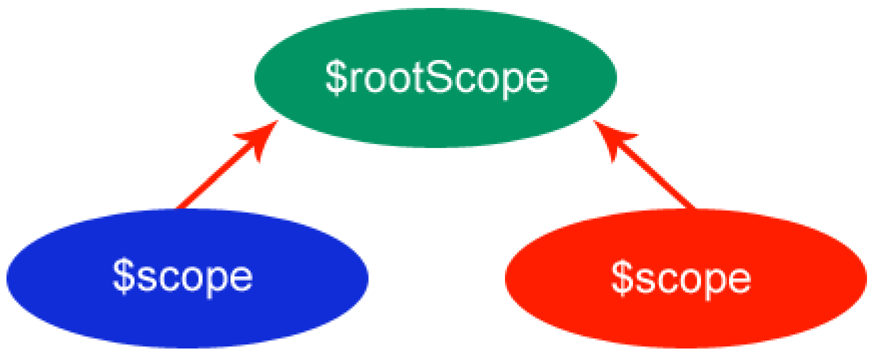
```
{
name: 'world',
action: function
}
```

```
<div ng-controller="MyCtrl">
  Hello {{name}}!
  <button ng-click="action()">
    OK
  <button>
</div>
```

Practice Exercise 2

Scopes

- Scopes are arranged in hierarchical structure which mimic the DOM structure of the application.
- Child scopes prototypically inherit from their parents
- Exactly one $rootScope, but may have several child scopes.
- If a property is not find, angular searches in parent scope and so on.

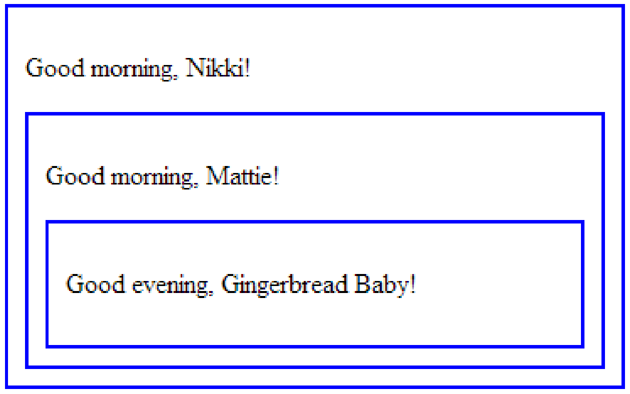


Reference source: http://github.com

Scopes

- Some directives creates new child scopes

```
<div class="spicy">
  <div ng-controller="MainController">
    <p>Good {{timeOfDay}}, {{name}}!</p>

    <div ng-controller="ChildController">
      <p>Good {{timeOfDay}}, {{name}}!</p>

      <div ng-controller="GrandChildController">
        <p>Good {{timeOfDay}}, {{name}}!</p>
      </div>
    </div>
  </div>
</div>
```



Good morning, Nikki!

Good morning, Mattie!

Good evening, Gingerbread Baby!

Practice Exercise 3

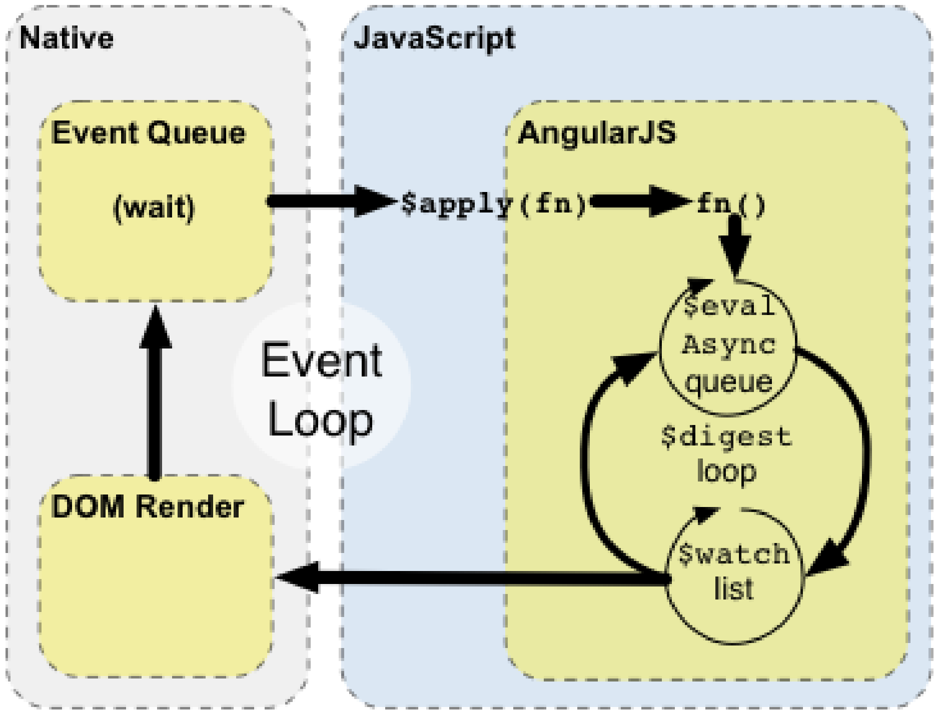
Scopes

- Core methods for two way binding
  - scope.$watch to observe model mutations
  - scope.$apply to propagate any model changes through the system into the view from outside of the "Angular Realm”

```
$scope.$watch('elapsed', function(){
  if ($scope.elapsed > 0)
    $scope.text = $scope.text + " World";
});

setTimeout(function(){
  var delta = (new Date().getTime() -
$scope.startedTime.getTime());
  $scope.elapsed = Math.round(delta / 1000);
  $scope.$apply();
}, 3000);
```

Scopes

- Every time we bind something in the UI we insert a $watch in a $watch list

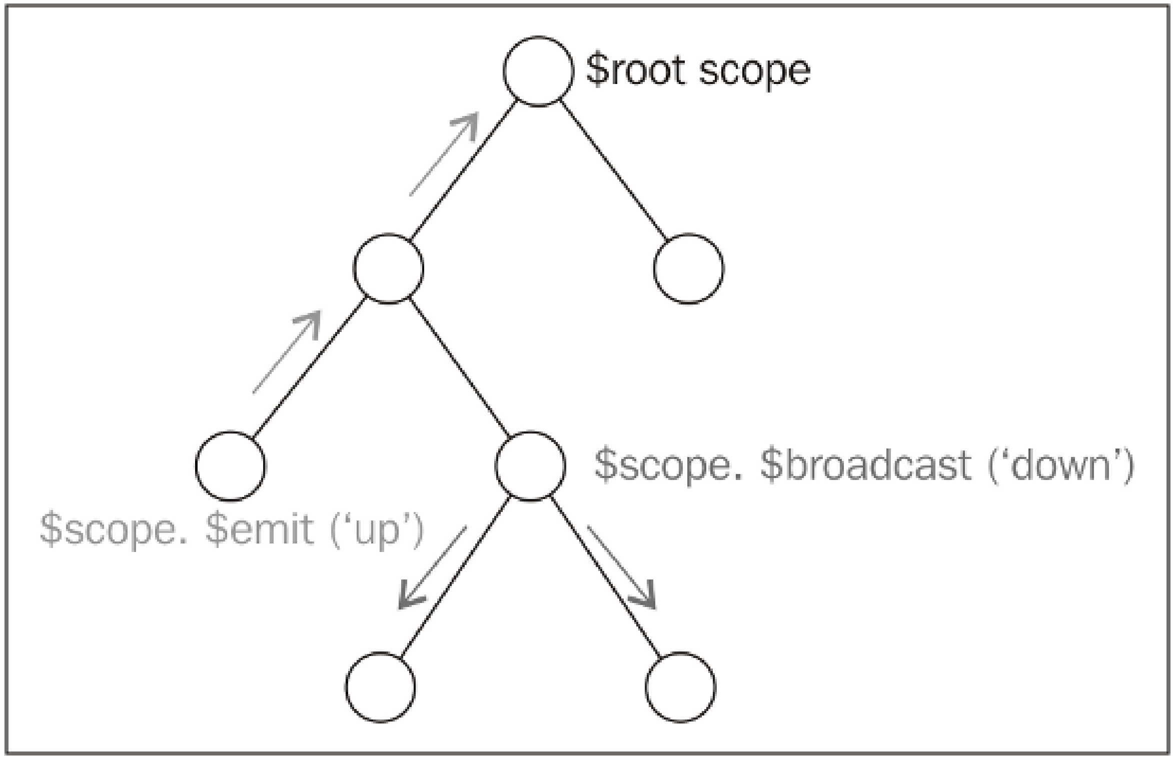


Reference source: http://angularjs.org

Practice Exercise 4

Scopes

- Scopes can propagate events in similar fashion to DOM events
  - $scope.$broadcast (‘paid‘, data): propagate to child scopes
  - $scope.$emit(‘paid‘, data): propagate to parent scope
  - $scope.$on(‘paid’, fn): capture event in destination scope



Practice Exercise 5

Expressions

- An “expression" in a template is a JavaScript-like code snippet that allows to read variables.

```
{{1+2}}
{{a+b}}
{{user.name}}
```

- In Angular, expressions are evaluated against a scope object, JavaScript expressions are evaluated against the global window
- Use filters within expressions to format data before displaying it.

```
{{ expression | filter }}
```

Filters

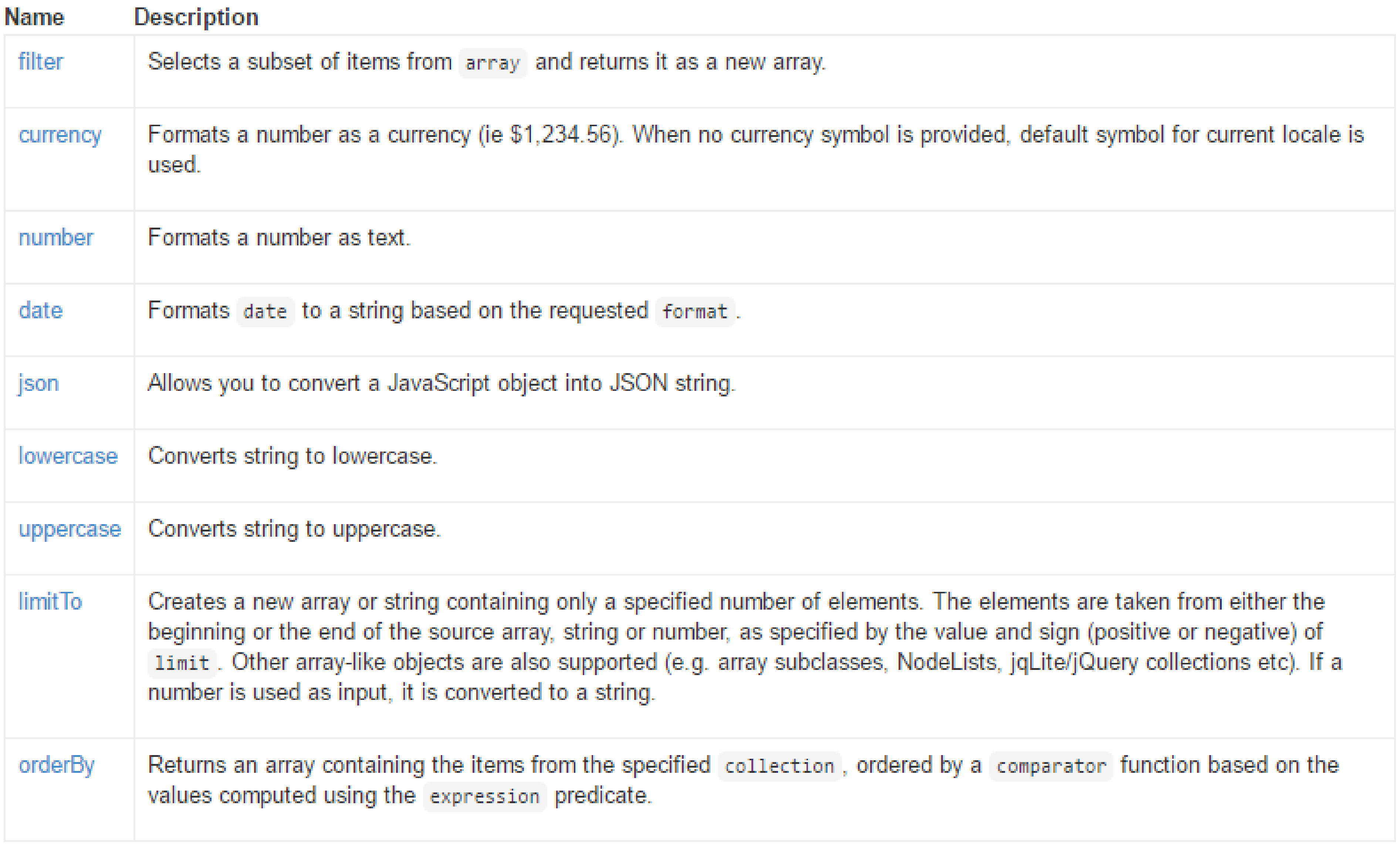
- A filter is used to format the value of an expression for displaying to the user
  - Uppercase a value,
  - Filter search results, etc.

```
{{user.name || upperCase}}
```

- Can be used in view templates, controllers or services
- Create a custom filter

```
module.filter('upperCase', function() {
     return  function(text){
        return text.toUpperCase();
     }
});
```

Built-in Filters



Name	Description
filter	Selects a subset of items from `array` and returns it as a new array.
currency	Formats a number as a currency (ie $1,234.56). When no currency symbol is provided, default symbol for current locale is used.
number	Formats a number as text.
date	Formats `date` to a string based on the requested `format`.
json	Allows you to convert a JavaScript object into JSON string.
lowercase	Converts string to lowercase.
uppercase	Converts string to uppercase.
limitTo	Creates a new array or string containing only a specified number of elements. The elements are taken from either the beginning or the end of the source array, string or number, as specified by the value and sign (positive or negative) of `limit`. Other array-like objects are also supported (e.g. array subclasses, NodeLists, jqLite/jQuery collections etc). If a number is used as input, it is converted to a string.
orderBy	Returns an array containing the items from the specified `collection`, ordered by a `comparator` function based on the values computed using the `expression` predicate.

Directives

- Directives are markers on a DOM element
  - Attach a specified behavior to that DOM element or
  - Transform the DOM element and its children.
- The HTML compiler traverses the DOM matching directives against the DOM elements.
- Should be the only place in AngularJS app do manipulate DOM

Directives

- Matching directives are called normalization
  - Strip x- and data- from the front of the element/attributes.
  - Convert the :, -, or _-delimited name to camelCase.

```
<input ng-model="foo">

<input data-ng:model="foo">
```

- Best practice: use the dash-delimited format (e.g. ng-bind for ngBind)

```
<my-dir></my-dir>
<span my-dir="exp"></span>
<!-- directive: my-dir exp -->
<span class="my-dir: exp;"></span>
```

Built-in Directives

- ng-model: binds an input,select, textarea (or custom form control) to a property on the scope
- ng-bind: replace the text content of the specified HTML element with the value of a given expression
- ng-click: specify custom behavior when an element is clicked
- ng-controller: attaches a controller class to the view
- ng-repeat: instantiates a template once per item from a collection. Each template instance gets its own scope
- ng-class: dynamically set CSS classes on an HTML element
- More information:
- https://docs.angularjs.org/api/ng/directive

Form Validation

- AngularJS offers client-side form validation.
- AngularJS monitors the state of the form and input fields (input, textarea, select), and lets you notify the user about the current state.
- AngularJS also holds information about whether they have been touched, or modified, or not.
- You can use standard HTML5 attributes to validate input, or you can make your own validation functions

```
<form name="myForm">

       <input name="myInput" ngmodel="myInput" required>

</form>
```

Angular Form Properties

Property	Class	Description
$valid	ng-valid	*Boolean* Tells whether an item is currently valid based on the rules you placed.
$invalid	ng-invalid	*Boolean* Tells whether an item is currently invalid based on the rules you placed.
$pristine	ng-pristine	*Boolean* True if the form/input **has not**been used yet.
$dirty	ng-dirty	*Boolean* True if the form/input **has** been used.
$touched	ng-touched	*Boolean* True if the input **has** been blurred.

Angular Form Properties

- **To access the form:** <form name>.<angular property>
- **To access an input:** <form name>.<input name>.<angular property>

```
<form name="myForm">

       <input name="myInput" ngmodel="myInput" required>

</form>
```

- **access form property**: myForm.$invalid
- **access input property**: myForm.myInput.$dirty

Validation Rules

```
<input    ng-model="{ string }"
          name="{ string }“
          required
          ng-required="{ boolean }"
          ng-minlength="{ number }"
          ng-maxlength="{ number }"
          ng-pattern="{ string }"
          ng-change="{ string }">
</input>
```

CSS classes

- Angular automatically sets below CSS classes
  - ng-scope angular applies this class to any element for which a new scope is defined
  - ng-binding: angular applies this class to any element that is attached to a data binding
  - ng-invalid, ng-valid: whether input does not pass validation
  - ng-pristine, ng-dirty: whether user has interaction on input widget element

CSS classes

- Source before typing email

```
<input name="author" ng-model="reviewCtrl.review.author"
type="email" required />
```

- Source after typing, with invalid email

```
<input name="author" . . . class="ng-pristine ng-invalid">
```

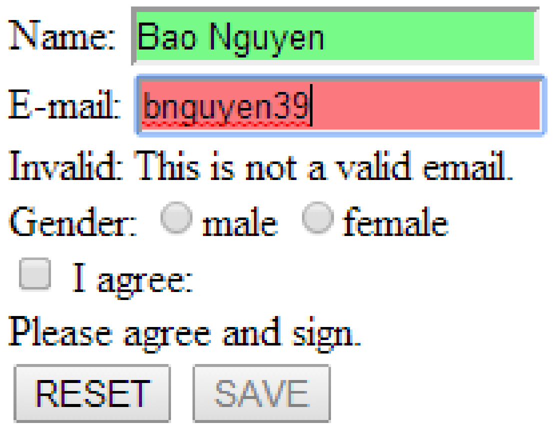
- Source after typing, with valid email

```
<input name="author" . . . class="ng-dirty ng-valid">
```

CSS classes

- Lets highlight the form field using classes

```
.ng-invalid.ng-dirty {
  border-color: #FA787E;
}
.ng-valid.ng-dirty {
  border-color: #78FA89;
}
```


Name: Bao Nguyen
E-mail: bnguyen39
Invalid: This is not a valid email.
Gender: ○ male ○ female
☐ I agree:
Please agree and sign.

Practice Exercise 6

Custom Directives

Custom Directives

- Why to use?
  - Custom directives let you create functionality that goes beyond the built-in directives AngularJS provides
- When to use?
  - Built-in directives don’t do what you want
  - You want to create self-contained functionality that can be reused in different applications.

Define Custom Directives

- How to use?
  - Define directives on modules

```
module.directive(‘customDirective', factoryFn);
```

  - **factoryFn** returns a directive definition object that defines directive properties and/or post link function.

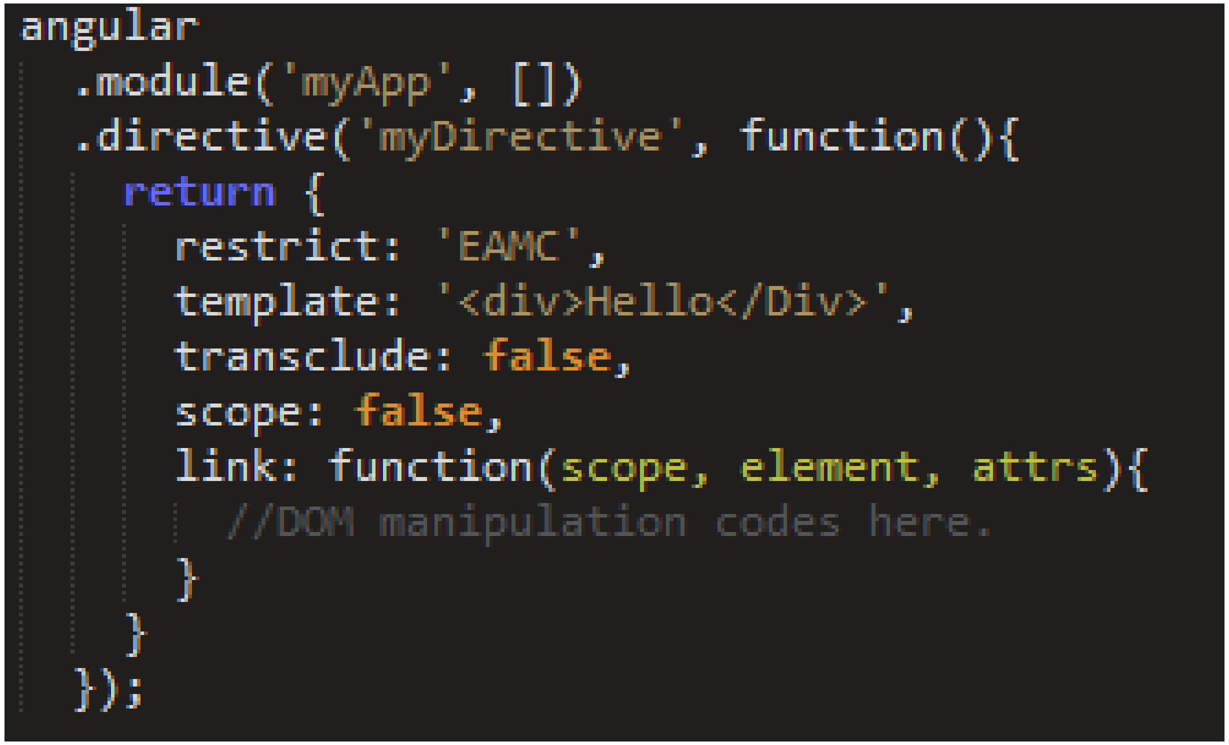
  - Use it like using DOM element

    <custom-directive> </custom-directive>

    <div custom-directive></div>

Building Custom Directives

- Everything but link is optional.



```
angular
    .module('myApp', [])
    .directive('myDirective', function(){
      return {
        restrict: 'EAMC',
        template: '<div>Hello</Div>',
        transclude: false,
        scope: false,
        link: function(scope, element, attrs){
          //DOM manipulation codes here.
        }
      }
    });
```

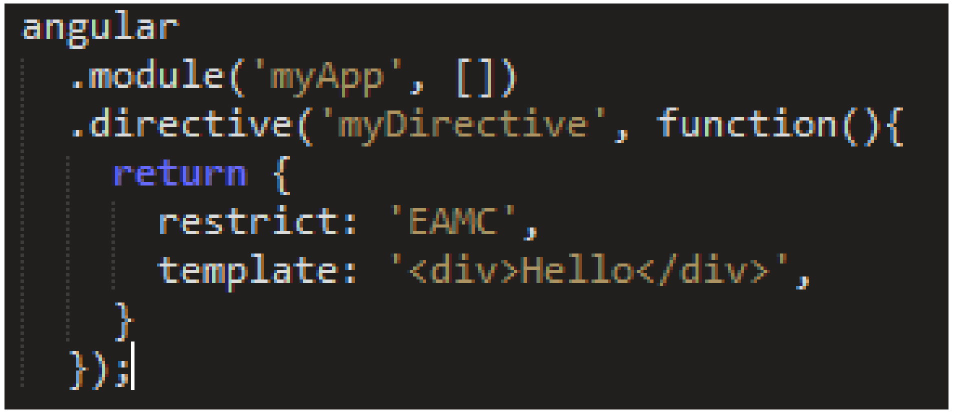
CSC

Restrict Property

- Directives can be restricted to a specific context
  - E: Element
  - A: Attribute
  - C: Class
  - M: Comment
- Restrict is declared as a single string, like ‘EA’. Default is ‘A’
- Common uses:
  - Use an element (E) when you want to create a new component that is in control of the template.
  - Use an attribute when you want to decorate an existing element with new functionality.

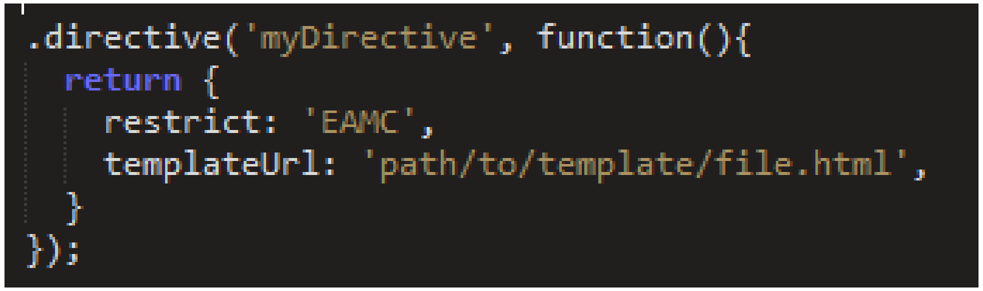
Template Property

- Template can be stored as strings on the **template** property



```
angular
  .module('myApp', [])
  .directive('myDirective', function(){
    return {
      restrict: 'EAMC',
      template: '<div>Hello</div>',
    }
  });
```

- Use **templateUrl** property in case html template is loaded from a file



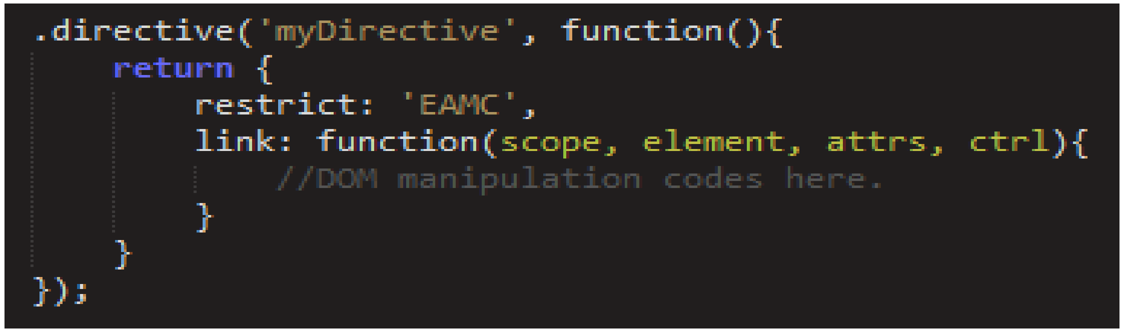
```
.directive('myDirective', function(){
  return {
    restrict: 'EAMC',
    templateUrl: 'path/to/template/file.html',
  }
});
```

Custom Directive - Demo

- Directive to wrap reusable component.

Link Function Arguments

- When to use?
  - To register DOM listeners
  - To update the DOM



```
.directive('myDirective', function(){
    return {
        restrict: 'EAMC',
        link: function(scope, element, attrs, ctrl){
            //DOM manipulation codes here.
        }
    }
});
```

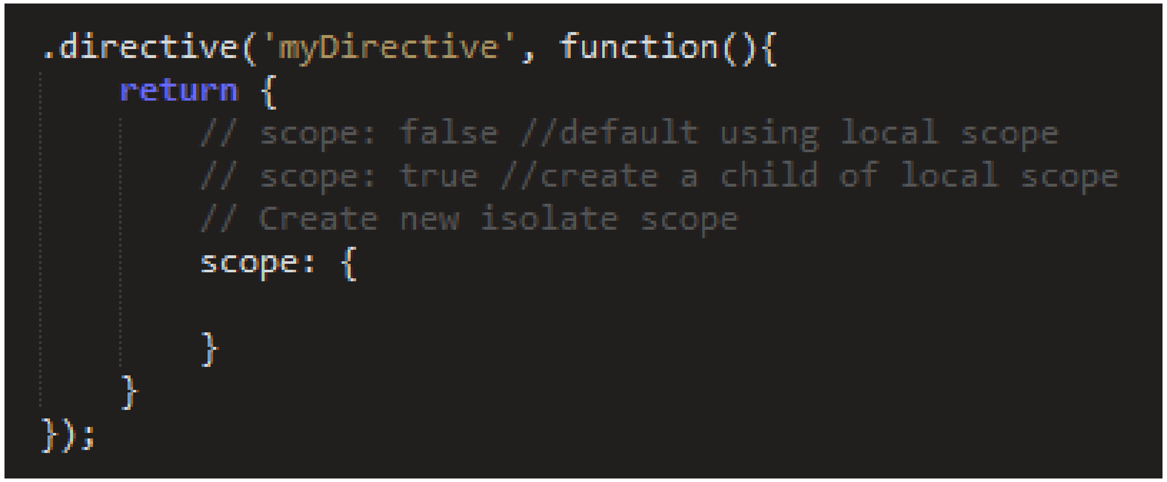
- Link function params:
  - **scope**: scope object associated with the directive
  - **element**: DOM element wrapped by jqLite
  - **attrs**: is a hash object with key-value pairs of normalized attribute names and their corresponding attribute values.
  - **ctrl**: reference to controller object.

Custom Directive - Demo

- Directive to manipulate DOM element.

Custom directive scope

- Scope options:
  - The same scope – scope from the environment that directive lives
  - A child scope that inherits from current scope
  - A new isolate scope



```
.directive('myDirective', function(){
    return {
        // scope: false //default using local scope
        // scope: true //create a child of local scope
        // Create new isolate scope
        scope: {

        }
    }
});
```

Custom directive – Isolate scope

- If set to true or { }:
  - New scope will be created for the directive
  - If multiple directives on the same element request a new scope, only one new scope is created
- Notation for property in scope:
  - @ or @attr - bind a local scope property to the value of DOM attribute
  - = or =attr - set up bi-directional binding between a local scope property and the parent scope property
  - & or &attr - provides a way to execute an expression in the context of the parent scope
- If no attr name is specified then the attribute name is assumed to be the same as the local name.

Custom Directive - Demo

- Directive with isolated scope.

Custom directive transclusion

- When we want to pass in entire template not string or object
- Using transclude to true compile the content of the element and make it available to the directive (where place ng-transclude)

```
angular.module('docsTransclusionDirective',
[])
  .directive('myDialog', function() {
    return {
      restrict: 'E',
      transclude: true,
      templateUrl: 'my-dialog.html'
    };
  });
```

- Use &attr to allow to pass in function which also execute within outside scope context

Custom Directive - Demo

- Directive with transclusion.

Custom Directive using ngModel

- The most popular directive of AngularJS
- Augments the HTML controls (input, radio,…) by providing the two-way data-binding
- Support CSS classes:
  - Class attribute of control is changed according to state
  - ng-valid; ng-invalid; ng-pristine; ng-dirty
- Binding to form and control state
  - State of control is automatically stored/updated in properties
  - $dirty; $error; $invalid; $pristine; $valid
- Custom triggers
  - Any change to the content trigger a model update and form validation
  - This behavior can be overridden by using ng-model-options
- AngularJS provides some validation directives for using with ngModal (required, pattern, minlength, maxlength, min, max)

Custom Form Directive

- Adds a custom validation function to the ngModel.
- The validation can occur in two phases
- Model to View update phase:
  - Whenever the bound model changes
  - All functions in NgModelController#$formatters array are pipe-lined
  - Format the value and change validity state of the form control through NgModelController#$setValidity.
- View to Model update phase:
  - Whenever a user interacts with a control
  - NgModelController#$setViewValue is called → NgModelController#$parsers array are pipe-lined
  - Format the value and change validity state of the form control through NgModelController#$setValidity.

Custom Form Directive

- Write your own form control as a directive using along with ngModal.
- Implement $render() method
  - To render the data
  - Call after it passed the NgModelController#$formatters
- Call $setViewValue() method
  - Whenever the user interacts with the control and model needs to be updated.
  - Usually done inside a DOM Event listener.

Custom Directive - Demo

- Custom form controls with ngModel

Unit Testing

Unit Test

- Why need to test
  - Ensure reliability in production
  - If it's hard to test, maybe it needs refactored?
  - Let developers refactor with confidence if any
- Angular Mock
  - Inject and mock Angular services into unit tests
  - Extends various core ng services
- Recommended Testing Suite: Mocha + Chai
- Recommended Test Runner: Karma

Unit Test Steps

- Install Node.js
- Install Karma via npm
- Configure Karma
- Write Unit test using Mocha + Chai framework
- Run Karma to connect to a browser to run unit test

Karma Test Runner

- Install Karma and command line

```
npm install -g karma;

npm install -g karma-cli;
```

- Init Karma configuration for our app, from command line type

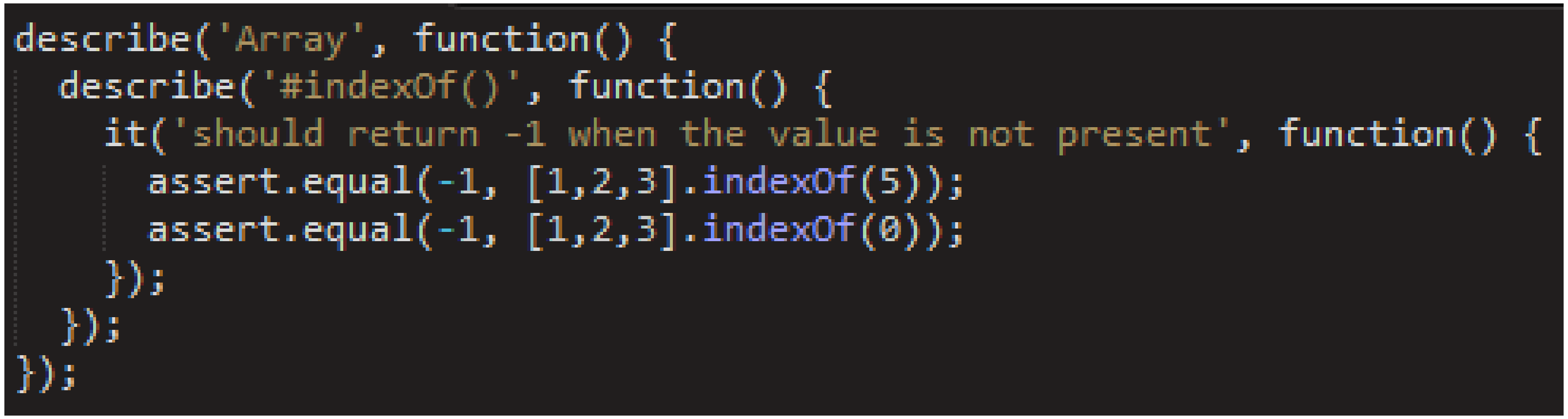
```
karma init my.conf.js;
```

- Start Karma server

```
karma start my.conf.js;
```

Mocha Framework

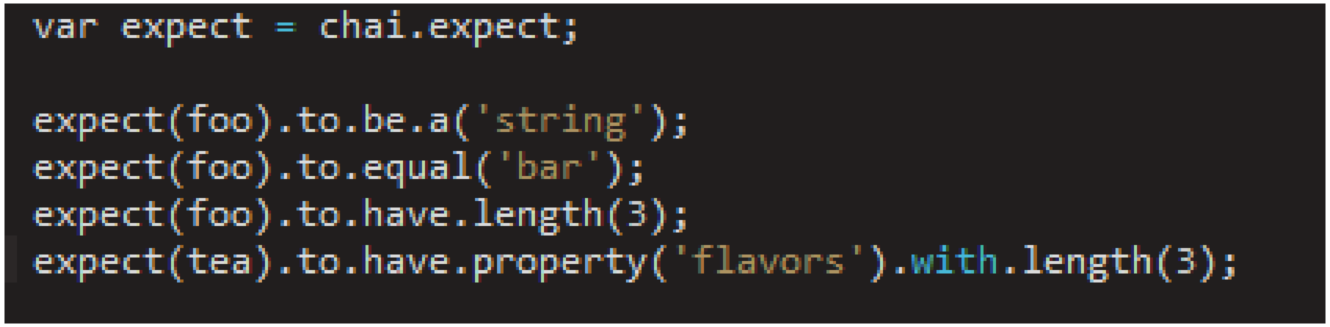
- Mocha is a testing framework for JavaScript. It handles test suites and test cases.
  - Describe a suite → describe
  - Describe a spec → it
  - Setup and teardown → beforeEach and afterEach
- Reference: https://mochajs.org/



```
describe('Array', function() {
  describe('#indexOf()', function() {
    it('should return -1 when the value is not present', function() {
      assert.equal(-1, [1,2,3].indexOf(5));
      assert.equal(-1, [1,2,3].indexOf(0));
    });
  });
});
```

Chai

- Chai offers various ways of checking things in test cases
- Chai checks are performed through assertions.
- Chai mark a test case as failed or passed.
- 3 styles available
  - Should()
  - Expect() → Most common used
  - Assert
- Reference: http://chaijs.com/



```
var expect = chai.expect;

expect(foo).to.be.a('string');
expect(foo).to.equal('bar');
expect(foo).to.have.length(3);
expect(tea).to.have.property('flavors').with.length(3);
```

Sinon

- Sinon describes itself as a standalone test spies, stubs and mocks for JavsScript.
- Sinon provides ways to replace one thing with another when running a test.
- Sinon provides a smart and concise way to monitor whether the function is called and much more (which arguments, how many times, etc.)
- Reference: http://sinonjs.org/

Unit Testing - Demo

- Controller testing
- Directive testing
- Factory testing
- Http request testing

Q&A

Thank You.

Revision History

Date	Versi on	Description	Updated by	Reviewed and Approved By
06/20/2016	1.0	Initial Version	Nguyen Ho Duy Nguyen Hai Tang	Quang Le