camptocamp

INNOVATIVE SOLUTIONS
BY OPEN SOURCE EXPERTS

INTRODUCTION TO ANGULARJS

STUDENT GUIDE

for Web Developers

TABLE OF CONTENTS

Course Objective

After completing this course, web developers will be able to start using AngularJS in web applications.

The course does not cover every aspect of AngularJS, but it should give you the basics required to correctly use AngularJS in the future.

Course Overview

- Introduction to Angular
- Overview of Angular's main concepts
- Detailed view of each concept
- Exercises
- Build a TODO application

About Camptocamp

Open Source specialist, innovative company in the software implementation of:

- Geographic Information Systems (**GIS**)
- Business Management (**ERP**)
- Server Management (**IT Automation and Orchestration**)

Present in three countries:

- Switzerland (Camptocamp SA)
- France (Camptocamp France SAS)
- Austria (Camptocamp GmbH)

1 GENERAL INTRODUCTION TO ANGULARJS

Objectives of this chapter

- Provide a very general introduction to Angular
- Discuss the « Why »
- Know how to set up a very simple Angular application

Definition

AngularJS is an open-source web application framework following the Model-View-Controller (MVC) design pattern.

AngularJS's design goals include:

- Decouple DOM manipulation from application logic
- Provide structure
- Improve testability of applications

AngularJS web site: http://angularjs.org.

AngularJS is very often called « Angular ».

What AngularJS is not...

It is not a UI framework!

- It doesn't provide UI widgets
- It doesn't provide CSS

Use other libraries for that. E.g. http://getbootstrap.com.

Declarative Programming

AngularJS is built around the belief that « declarative programming » should be used for building user interfaces.

This is what an « AngularJS-powered » web page looks like:

```
<!doctype html>
<html ng-app="app">
  <head>
    <script src="angular.min.js"></script>
  </head>
  <body>
    <div>
      <label>Name:</label>
      <input type="text" ng-model="yourName" />
      <h1>Hello {{yourName}}</h1>
    </div>
  </body>
</html>
```

This page does not require any application-specific JavaScript code.

MVC

Quoting Wikipedia:

« The **Model** stores data that is retrieved by the controller and displayed in the view. »

« The **View** requests information from model that it uses to generate an ouput representation to the user. »

« The **Controller** can send commands to the model to update the model's state. It can also send commands to its associated view to change the view's presentation of the model. »

We will come back to that...

Why use AngularJS?

The main points:

- AngularJS helps you structure your application
- AngularJS helps you synchronize states across the application
- AngularJS provides useful features (e.g. localization, touch)

AngularJS is well-suited for « Single-Page Applications ».

Gigantic eco-system

The AngularJS eco-system/community is huge!

- AngularUI
- angular-translate
- ionic
- karma
- protractor
- ...

Angular 2

Angular 2 is young and still under devlopment.

Angular 2 will be very different from Angular 1.x. Applications written with Angular 1.x will have to be re-written for Angular 2.

Using good practices will help migrating. Angular 1.4 will provide features helping with future migrations to Angular 2.

Exercise 1.1: Set up an Angular app

Exercise

Create a simple web page using Angular.

Objectives

Discover how to work with Angular.

Demonstrate a key concept of Angular: « data-binding ».

Discover and use Plunker, which is typically used for Angular examples.

2 OVERVIEW OF ANGULAR'S MAIN CONCEPTS

Objectives of this chapter

- Provide an introduction to Angular's main concepts
- Categorize the concepts based on the MVC design pattern

We're not going to go into detail, so don't worry if things are still a bit « abstract » after this chapter!

More explaination there: https://docs.angularjs.org/guide/concepts

Angular has many concepts. Be warned (and

MVC

Angular is based on the MVC Design Pattern.

View

Wikipedia definition: « The **View** requests information from model that it uses to generate an ouput representation to the user. »

The **View** in Angular is the HTML representation. The application developer defines the View in HTML **Templates**.

Example:

```
<!doctype html>
<html ng-app="app">
  <head>
    <script src="angular.min.js"></script>
  </head>
  <body>
    <div>
      <label>Name:</label>
      <input type="text" ng-model="yourName" />
      <h1>Hello {{yourName}}</h1>
    </div>
  </body>
</html>
```

`yourName` is an *expression*. It is replaced by a value when Angular *compiles* the HTML.

View-related Concepts

- Templates
- Expressions
- Filters
- Directives (`ng-model` in the previous example)
- Compilation
- Data-binding

The **Compilation** is the phase during which Angular parses the HTML code to evaluate **Expressions**, and detects the use of **Directives** to attach behavior to DOM elements.

Data-binding is about binding the **View** and the **Model** together.

Model

Wikipedia definition: « The **Model** stores data that is retrieved by the controller and displayed in the view. »

In Angular, the **Model** stores the application data and states that are used to generate the **View**.

Model-related Concepts

- Scopes – the objects containing the application data

Data-binding

Data-binding is a core concept of Angular. It is about binding the Model and the View together.

The View changes automatically when the Model changes. And **vice-versa**. The terms « bi-directional » or « two-way » data-binding are very often used.

Controller

Wikipedia definition: « The **Controller** can send commands to the model to update the model's state. It can also send commands to its associated view to change the view's presentation of the model. »

The **Controller** part in Angular is defined as **Controller** (constructor) functions.

Controller functions have a reference to the **Scope** object, and can therefore manipulate the model data.

Controller-related Concepts

- Controllers
- Scopes
- Services

Services are objects that can be **Injected** into **Controllers**.

Services

(Angular-specific notion. Unrelated to MVC.)

Services are named objects, that can be **Injected** in **Controllers** or other **Services**.

Services are Singletons.

Writing **Services** is a good way to encapsulate functionality and logic.

Dependency Injection

As defined in the Angular documentation: « Dependency Injection (DI) is a software design pattern that deals with how components get hold of their dependencies. »

Modules

(Angular-specific notion. Unrelated to MVC.)

Modules are containers for the different parts of an application (Directives, Controllers, Services, Filters).

Review of Concepts

- MVC
- View
  - Templates
  - Expressions
  - Directives
  - Compilation
  - Data-binding
- Model
  - Scopes
- Controller
  - Controllers
- Services
- Dependency Injection
- Modules

3 MORE DETAILS ON THE CONCEPTS

Objectives of this chapter

- Provide more details on each Angular Concept
- Make things more concrete through examples and exercises

This is a long chapter! With exercises...

Templates

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

The View is written as HTML **Templates**. A Template includes HTML markup and **variables** (**expressions** using variables really).

Example:

```
<!doctype html>
<html ng-app="app">
  <head>
    <script src="angular.min.js"></script>
  </head>
  <body>
    <div>
      <label>Name:</label>
      <input type="text" ng-model="yourName" />
      <h1>Hello {{yourName}}</h1>
    </div>
  </body>
</html>
```

`yourName` is an expression. A very simple expression based on a single variable (`yourName`) here.

Expressions

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

The View includes **Expressions** (as already explained in the previous slide).

Examples of expressions:

- `1 + 2`
- `a + b`
- `name`
- `user.name`
- `items[index]`
- `myFunc(myArg)`

Expressions

Angular expressions are like JavaScript expressions, with some differences.

Angular expressions are evaluated against a **Scope** object (the Scope object is the expression evaluation context).

For example, in the expression `a + b`, `a` and `b` are properties of a Scope object.

Exercise 3.1: Play with Expressions

Exercise

Play with Expressions.

Objectives

Get more familiar with the use of Expressions.

Directives

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

Key Concept of Angular!

Angular extends the HTML vocabulary through the concept of **Directives**. Directives are « markers » on DOM elements.

Example:

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

The directive used here is `ngModel`. In this case it is used as an attribute on an `input` element.

A directive attaches a specific behavior to a DOM element. It may even transform the DOM element and its children.

Angular comes with a set a directives, like `ngBind`, `ngModel` and `ngClass`. Custom directives can be written by the application developer.

Angular Built-in Directives

https://docs.angularjs.org/api

- `ngApp`
- `ngController`
- `ngBind`
- `ngHide` and `ngShow`
- `input` and `ngModel`
- `ngClick`

Angular Built-in Directives

- `ngInit`
- `ngIf`
- `ngClass`
- `ngRepeat`
- `select` and `ngOptions`
- ...

The next slides provide more details on each of those directives.

ngApp

https://docs.angularjs.org/api/ng/directive/ngApp

Designate the root element of the Angular application.

Example:

```
<!doctype html>
<html ng-app="app">
  <head>
    <script src="angular.min.js"></script>
  </head>
  <body>
    …
  </body>
</html>
```

ngController

https://docs.angularjs.org/api/ng/directive/ngController

Attach a controller to the view to an element.

Example:

```
<div ng-controller="Ctrl"></div>
```

The directive creates a new scope and attaches it to the element.

ngBind

https://docs.angularjs.org/api/ng/directive/ngBind

Replace the text content of the element with the value of the given expression, and udpate the text content when the value of the expression changes.

Example:

```
<div ng-bind="a + b"></div>
```

You can use `ng-bind` or the double curly markup `{{ }}`.

`ngHide` and `ngShow`

https://docs.angularjs.org/api/ng/directive/ngHide

https://docs.angularjs.org/api/ng/directive/ngShow

Show or hide an element based on the given expression.

Example:

```
<div ng-hide="hidden"></div>
```

The div element will be hidden when the `hidden` expression evaluates to `true`.

Same as:

```
<div ng-show="!hidden"></div>
```

`input` and `ngModel`

https://docs.angularjs.org/api/ng/directive/input

https://docs.angularjs.org/api/ng/directive/ngModel

The `ngModel` directive binds an `input` to a property on the scope.

Example #1:

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

Example #2:

```
<input type="checkbox" ng-model="checkboxValue" />
```

Other supported `input` types: `date`, `email`, `number`, `time`, `url`, `month`, `week`.

`ngModel` can also be used together with `select` and `textarea` form fields.

ngClick

https://docs.angularjs.org/api/ng/directive/ngClick

Specify a custom behavior when an element is clicked.

Example #1:

```
<div ng-click="count = count + 1"></div>
```

Example #2:

```
<div ng-click="incCount()"></div>
```

(Which one do you prefer?)

Notes:

You should probably prefer the second form, to avoid adding logic to the view.

ngInit

https://docs.angularjs.org/api/ng/directive/ngInit

Allow to evaluate an expression in the current scope.

Example:

```
<div ng-click="count = count + 1" ng-init="count = 0"></div>
```

Don't abuse of this logic ! Prefer to initialize values in a controller.

ngIf

https://docs.angularjs.org/api/ng/directive/ngIf

Remove and recreate a portion of the DOM tree based on an expression.

Example:

```
<input type="checkbox" ng-model="checked" ng-init="checked = true">
<span ng-if="checked">Removed when the checkbox is unchecked.</span>
```

ngClass

https://docs.angularjs.org/api/ng/directive/ngClass

Set dynamically CSS classes on an HTML element.

Example:

```
<input type="checkbox" ng-model="checked">
<p ng-class="{'in-green': checked, strike: !checked}">I'm checked</p>
```

ngRepeat

https://docs.angularjs.org/api/ng/directive/ngRepeat

Repeat a template for each item of a collection.

Example:

```
<ul>
  <li ng-repeat="item in items">{{item.name}}</li>
</ul>
```

`ngRepeat` can be used in many different ways. This is one of the most complex directives of Angular.

`select` and `ngOptions`

https://docs.angularjs.org/api/ng/directive/select

https://docs.angularjs.org/api/ng/directive/ngOptions

`select`, `ngOptions` and `ngModel` are used together to create a selector.

Example:

```
<select ng-model="selectedColor"
        ng-options="color.name for color in colors"></select>
```

Exercise 3.2: Play with Directives

Exercise

Play with Angular Directives.

Objectives

Get more familiar with the use of Angular directives.

Scopes

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

We mentioned **Scopes** already when we discussed Expressions. Scopes are the evaluation contexts of Expressions.

Example:

```
<h1>{{a + b}}</h1>
```

`a + b` is an expression. It is evaluated against a given Scope object, meaning that `a` and `b` are properties of the Scope object.

Another example:

```
<h1>{{ sum(a, b) }}</h1>
```

`sum(a, b) is an expression`.

`sum`, `a` and `b` are properties of a given Scope object. The `sum` property is a reference to a function.

Scopes

When are Scope objects created?

→ Some directives create new scopes. Some don't.

Scopes are arranged in a hierarchical structure that mimics the DOM structure of the web page.

Example:

```
<div ng-controller="Ctrl1">
  <div ng-controller="Ctrl2"></div>
</div>
```

Both the outer and inner `div` elements have a `ngController` directive attached to them. The `ngController` directive creates a new Scope.

This means that each div has an associated Scope. The Scope associated to the inner `div` is a child Scope of the Scope associated to the outer `div`.

The `$watch` Scope method

https://docs.angularjs.org/api/ng/type/$rootScope.Scope

Scope objects expose a number of methods. `$watch` is one of the most important/used one.

`$watch` is used to watch an expression and be notified when the value of that expression changes.

Example:

```
$scope.$watch('foo', function(newVal, oldVal) {
  // function executed when the value of the "foo" property
  // changes on the scope
});
```

Related methods: `$watchCollection` and `$watchGroup`.

/!\ Watch methods are heavy and can slow down you application.

The `$apply` Scope method

https://docs.angularjs.org/api/ng/type/$rootScope.Scope

`$apply` is also one of the most important/used Scope methods.

`$apply` evaluates an expression and fires a « Digest Cycle » using the `$digest` method on the root scope.

The Digest Cycle is the phase where Angular examines all the `$watch` expressions and compare them with the previous value.

Example:

```
element.on('click', function() {
  // "click" handler executed outside the Angular context, use $apply to
  // execute a function and trigger a Digest Cycle.
  $scope.$apply(function() {
    $scope.state = true;
  });
});
```

Or as an argument: `$scope.$apply('state = true')`;

Controllers

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

A **Controller** is a JavaScript constructor function.

A Controller instance is created when the `ngController` directive is attached to a DOM element.

Example:

```
<div ng-controller="Ctrl">
  <p>a + b : {{ sum(a + b) }}</p>
</div>
```

This assumes that a Controller named `Ctrl` was previously declared.

Declaring a Controller

This is how a Controller is declared:

```
aModule.controller('Ctrl', function($scope) {
  $scope.a = 1;
  $scope.b = 2;
  $scope.sum = function(a, b) {
    return a + b;
  };
  // ...
});
```

`controller` is the function used to create a Controller. The first argument is the Controller name. The second argument is the Controller itself (i.e. the constructor function).

Note that the Controller may receive the new Scope as an argument.

Exercise 3.3: Play with Controllers and Scopes

Exercise

Play with Controllers and Scopes.

Objectives

Know how to declare a controller.

Know how to instantiate/use a controller.

Know how to get a reference to a controller's scope.

Know how to refer to scope properties from the view.

Know how to use built-in directives such as ngRepeat and ngOptions.

Filters

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

A filter formats the value of an Expression. Filters are commonly used in templates, but they can also be used in controllers (or elsewhere).

Example:

```
<p>{{ 1234 | number:2 }}</p>
```

This formats the number 1234 with 2 decimal points using the `number` filter. The resulting value is `1234.00`.

Custom filters may be written by the application developer.

Services

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

Services are named objects, that can be **injected** in controllers or other services.

Example:

```
aModule.controller('Ctrl', function($http) {
  // ...
});
```

The `$http` service is injected in the Controller `Ctrl`.

Services

Another example:

```
aModule.factory('myService', function($http) {
  return {
    sendXHR: function() {
      // use $http...
    }
  };
});
```

This code declares a service named `myService` which is an object with a `sendXHR` function using Angular's `$http` service.

Services

Services can be anything from a `number` to a more complex `object`. It can be a `string` or an `array`.

How to declare a Service?

A Service is declared by calling one of the *service creation* functions on a module object.

The service creation functions are:

- `factory`
- `service`
- `value`
- `provider`
- `constant`

Declare a Service using `value`

Example:

```
aModule.value('myService', {myProp: 1});
```

The first argument is the service name. The second argument is the service object.

When using `value` the service is created by the application developer, at service declaration time.

Declare a Service using `factory`

Example:

```
aModule.factory('myService', function() {
  return {
    // ...
  };
});
```

The fist argument is the service name. The second argument is a factory function that returns the service.

Angular will call the factory function, and effectively create the service, the first time `myService` is injected into a controller or a service.

Declare a Service using `service`

Example:

```
aModule.service('myService', function() {
  this.myMethod = function() {
    // ...
  };
});
```

The first argument is the service name. The second argument is a constructor function for the service, which Angular will call using `new` (more or less).

In the above example the `myService` service is an object with a `myMethod` property which is a function.

Declare a Service using `provider`

The canonical form!

Example:

```
aModule.provider('myService', function() {
  this.$get = function() {
    return {
      // ...
    };
  };
});
```

The first argument is the service name. The second argument is a constructor function. The object created by that constructor function must have a `$get` method returning the service. That object is called a Provider.

`provider` is rarely used, but may be useful in cases where the creation of the service should be configurable.

Declare a Service using `constant`

Example:

```
aModule.constant('myService', {myProp: 1});
```

The first argument is the service name. The second argument is the service object.

Very similar to `value`, except that, with `constant`, the service provider and the service are the same object.

Also, the `constant` services are instanciated earlier (at the application bootstrap time), then they can be used for the configuration of the module.

When to declare new services?

Services are a good way to encapsulate functionality and logic.

Do not refrain yourself from creating services. Creating many small services is often a good idea!

Angular Built-in Services

https://docs.angularjs.org/api

Angular comes with a number of built-in services. They include, but are not limited to:

- `$http`
- `$location`
- `$rootScope`
- ...

`$http`

https://docs.angularjs.org/api/ng/service/$http

Object to use to communicate with remote HTTP services via XMLHttpRequest or JSONP.

Example:

```
myModule.controller('MyCtrl', function($scope, $http) {
  $http.get('http://search.com', {
    params: {
      q: 'angular'
    }
  }).success(function(data) {
    $scope.result = data;
  });
});
```

In this example, the `$http` service is injected and used in the `MyCtrl` controller function.

$location

https://docs.angularjs.org/api/ng/service/$location

Service that can parse the URL in the browser address bar and make the URL available to the application.

Example:

```
myModule.controller('MyCtrl', function($location) {
  // set or update the "foo" parameter in the address bar URL
  $location.search('foo', 'bar');
});
```

In this example, the `$location` service is injected and used in the `MyCtrl` controller function.

$rootScope

https://docs.angularjs.org/api/ng/service/$rootScope

Every application has a single root scope. This service is a reference to the application's root scope.

Example:

```
myModule.controller('MyCtrl', function($rootScope) {
  $rootScope.globalState = 'foo';
});
```

In this example, the `$rootScope` service is injected and used in the `MyCtrl` controller function.

Dependency Injection

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

« Dependency Injection (DI) is a software design pattern that deals with how components get hold of their dependencies. »

DI is pervasive throughout Angular. You can use DI in Controller constructors, Service factories, etc.

DI is about getting a reference to a service by its name.

Example:

```
aModule.factory('aService', function(anotherService) {
  // The factory function has a reference to `anotherService`,
  // which will be created by Angular if it's not created yet.
});
```

Dependency Injection

The « minification » issue

The following won't work as expected if the JavaScript code is minified.

```
aModule.controller('Ctrl', function($http) {
  $http.get('aURL').success(function() {
    // ...
  });
});
```

Can you see the reason?

This is because the minifier tool is going to rename the `$http` variable to a shorter name, making it impossible for Angular to know that the `$http` service is to be injected.

Dependency Injection

Solution to the « minification » issue

Use the « Inline Array Notation ».

Example:

```
aModule.controller('Ctrl', ['$http', function($http) {
  $http.get('aURL').success(function() {
    // ...
  });
}]);
```

Exercise 3.4: Declare and use Services

Exercise

Declare and use Services.

Objectives

Know how to declare a service using `value`.

Modules

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

A module is a container for the different parts of an application (Directives, Controllers, Services, Filters).

Directives, Controllers, Services and Filters are declared in a given module.

Example:

```
aModule.controller('Ctrl', function() {
  // ...
});
```

In this example the `Ctrl` Controller is declared in the module referenced to by the `aModule` variable.

Modules

How to create a module?

→ Using the `angular.module` function.

Example:

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

This creates a module named `app` that depends on the `ngAnimate` module.

Modules

How to get a reference to an existing module?

→ Using the `angular.module` function again, but with just one argument this time.

Example:

```
var aModule = angular.module('app');
```

Angular Built-in Modules

https://docs.angularjs.org/api

In Angular itself, Services, Directives and Filters are grouped by modules.

The main module, `ng`, is implicit, in the sense that, when you create a module, you don't need to specify that your module depends on `ng`. That is implicit (and always the case).

The use of an Angular module requires loading another Angular script in the page. For example using `ngAnimate` requires loading `angular-animate.js` in the page.

Angular Built-in Modules

Other modules provided by Angular:

- `ngAnimate` https://docs.angularjs.org/api/ngAnimate
- `ngAria` https://docs.angularjs.org/api/ngAria
- `ngCookies` https://docs.angularjs.org/api/ngCookies
- `ngResource` https://docs.angularjs.org/api/ngResource
- `ngRoute` https://docs.angularjs.org/api/ngRoute
- `ngSanitize` https://docs.angularjs.org/api/ngSanitize
- `ngTouch` https://docs.angularjs.org/api/ngTouch

Exercise 3.5: Avoid leaking into the global object

Exercise

Avoid leaking into global object

Objectives

Learn about JavaScript self-executing functions. (This exercise is not directly related to Angular.)

4 HANDS-ON EXERCISE: BUILD A TODO APPLICATION

Exercise 4.1: Set up the application

Create the initial HTML and JavaScript files for the TODO application.

At this point the files are empty shells, with code just to test that Angular is correctly set up.

Objectives:

- **Set up the application**
- **Learn about the `controller as` syntax**

Exercise 4.2: Create a service to load the initial todo list

The initial list of todo items is in a JSON file, and Angular's `$http` service is used to load that file.

Objectives:

- **Review the declaration of a service using `module.factory`**
- **Use Dependency Injection to inject a service into another service**
- **Introduction to Promises (HTTP Promises in that case)**

Exercise 4.3: Display the todo list

Objectives:

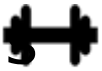
- **Use Dependency Injection to inject a service into a controller**
- **Use `ng-repeat` in the view**

Exercise 4.4: Add checkboxes to mark todo items as "done"

Objectives:

- **Use `ng-model` with a checkbox input**
- **Use Angular to control the class applied to an element**

Exercise 4.5: Add a way to add todo items to the list



Objectives:

- **Use `ng-model` with an input text element**
- **Use `ng-click` and function expressions**

Exercise 4.6: Add an "archive" button

Objectives:

- **Use function expressions again**
- **Use JavaScript's `splice` function**

5 WRITE CUSTOM DIRECTIVES

Objectives of this chapter

Provide an introduction to writing custom directives

Directives

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

Key Concept of Angular!

Angular extends the HTML vocabulary through the concept of **Directives**. Directives are « markers » on DOM elements.

Example:

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

The directive used here is `ngModel`.

Custom directives

Angular comes with built-in directives (`ngModel`, `ngClick`, ...), and **Custom Directives** can be created.

Example #1:

```
<div my-custom-directive></div>
```

Example #2:

```
<my-custom-directive></my-custom-directive>
```

In both cases a directive named `myCustomDirective` is used.

Declaring a Custom Directive

The `directive` module method is used to declare a custom directive.

Example:

```
myModule.directive('myCustomDirective', function() {
  return {
    ...
  };
});
```

The first argument is the directive name. The second argument is a function that returns a **Directive Definition Object**.

A **Directive Definition Object**, as its name suggests, is an object that defines the directive that is being declared.

Directive Definition Object (DDO)

A more complete example:

```
myModule.directive('myCustomDirective', function() {
  return {
    restrict: 'E',
    scope: {
      name: '='
    },
    template: '<span>{{name}}</span>',
    link: function(scope, element, attrs) {
      element.on('click', function() {
        // …
      });
    }
  };
});
```

DDO Properties

The most commonly used DDO properties are:

- `template`
- `templateUrl`
- `restrict`
- `link`
- `scope`
- `controller`
- `controllerAs`
- `bindToController`

There are other DDO properties. Only those properties are covered by this course.

The `template` DDO property

Example of a DDO using `template`:

```
myModule.directive('myCustomer', function () {
  return {
    template: 'Name: {{customer.name}}<br /> Street: {{customer.street}}'
  };
});
```

Use of the `myCustomer` directive:

```
<div my-customer></div>
```

The HTML specified with the `template` property is added to the `<div my-customer` element.

Exercise 5.1: The `template` DDO property

Declare and use a directive `myCustomer` whose definition includes `template`.

The `templateUrl` DDO property

With `templateUrl`, the template for the directive is loaded form a URL, asynchronously.

Example of a DDO using `templateUrl`:

```
myModule.directive('myCustomer', function () {
  return {
    templateUrl: 'my-customer.html'
  };
});
```

Exercise 5.2: The `templateUrl` DDO property

Use `template` instead of `templateUrl`.

The `restrict` DDO property

Restrict the directive to a specific directive declaration style. Possible values are `E` (element), `A` (attribute), `C` (class), and `M` (comment).

Example:

```
myModule.directive('myCustomer', function () {
  return {
    restrict: 'E',
    templateUrl: 'template.html'
  };
});
```

With this definition the directive should be used as an element:

```
<my-customer><my-customer>
```

`C` (class) and `M` (comment) are very rarely used.

Exercise 5.3: The `restrict` property

Change the code so that the directive is declared using an HTML tag.

The `link` DDO property

The `link` property is a reference to a function, which is used to associate some behavior to the element the directive is attached to. The `link` function is the right place to manipulate the DOM.

Example:

```
app.directive('myCustomer', function() {
  return {
    restrict: 'E',
    templateUrl: 'my-customer.html',
    link: function($scope, element, attrs) {
      element.bind('mouseenter', function() {
        element.css('background-color', 'yellow');
      });
      element.bind('mouseleave', function() {
        element.css('background-color', 'white');
      });
    }
  };
});
```

In this example the `link` function registers `mouseenter` and `mouseleave` listeners on the element.

Exercise 5.4: The `link` property

Add some behavior to the element the directive is attached to.

The `scope` DDO property

The `scope` property defines whether a scope should be created for the directive.

The `scope` is either a boolean or an object literal.

- `false` means no specific scope is created for the directive
- `true` means a specific, non-isolate, scope is created
- `{}` means that a specific, **isolate**, scope is created

Example:

```
app.directive('myCustomer', function() {
  return {
    restrict: 'E',
    templateUrl: 'my-customer.html',
    scope: {}
  };
});
```

Isolate scope

An **isolate scope** is a scope that does not *prototypically* inherit from its parent scope.

An **isolate scope** has therefore no access to its parent scope's properties.

A directive with an isolate scope is isolated from its environment. This may be useful when building reusable components, which **should not accidentally read or modify data** in the parent scope.

Isolate scope properties

The `scope` DDO property may include properties which define the local scope properties derived from the parent scope.

There are multiple ways to specify how local scope properties derived from the parent scope:

- Using `&` or `&attr` (One-time binding)
- Using `@` or `@attr` (One-time binding)
- Using `=` or `=attr` (Two-ways binding)
- Using `<` or `<attr` (One-way binding)

Where `attr` is an attribute of the element who stand this directive.

Isolate scope properties with `&attr`

Declaration example:

```
app.directive('myCustomer', function() {
  return {
    scope: {
      'increment': '&customerNumber'
    },
    link: function(scope, element, attrs) {
      scope.increment();
      // ...
    }
  };
});
```

Use example:

```
<my-customer customer-number="count = count + 1"></my-customer>
```

One-time binding: The `customer-number` value will be passed as a function. The value in the parent scope and in the directive are completely separated.

Isolate scope properties with `@attr`

Declaration example:

```
app.directive('myCustomer', function() {
  return {
    scope: {
      'name': '@customerName'
    },
    link: function(scope, element, attrs) {
      scope.$watch('name', function(newVal) {
        // ...
      });
    }
  };
});
```

Use example:

```
<my-customer customer-name="Hello {{ customer.name }}"></my-customer>
```

One-time binding: The customer-name value will be exactly the same in the directive scope. So the $scope.name will value `Hello {{ customer.name }}`. Useful to keep a value 'as is'.

Exercise 5.5: Isolate scope properties with @attr

Create an isolate scope with properties and the @attr notation.

Isolate scope properties with `<attr`

One-way binding: Like with `=attr`, if the directive scope change, the parent scope will change. But that's not the case in the other way !

Isolate scope properties - Live example

Edit this live example: http://plnkr.co/edit/7Ii9VibepeHdDeDKKfmt?p=preview

Exercise 5.6: Isolate scope properties with =attr

Create an isolate scope with properties and the =attr notation.

The `controller`, `controllerAs` and `bindToController` DDO properties

Example:

```
app.directive('myCustomer', function() {
  return {
    scope: {
      name: '=customerName'
    },
    template: '<li ng-click="ctrl.onClick()">{{ctrl.name}}</li>'
    bindToController: true,
    controller: function() {
      this.onClick = function() {
        alert('clicked');
      };
    },
    controllerAs: 'ctrl'
  };
});
```

Explanations:

`controller` specifies that a controller object should be created and attached to the directive. The provided function is a constructor for that object.

`bindToController` specifies that the properties specified with the `scope` DDO property (`name` here) are set on the controller rather than on the scope.

`controllerAs` specifies the name of the property that references the controller on the directive scope.

Exercise 5.7: Create a custom directive

Objectives:

- **Use a custom directive to display each todo item.**

WWW.CAMPTOCAMP.COM

Open Source specialist, Camptocamp consists of three divisions: GEOSPATIAL SOLUTIONS, BUSINESS SOLUTIONS and INFRASTRUCTURE SOLUTIONS. Our professional, innovative and responsive services help you implement your most ambitious projects.

SWITZERLAND Quartier de l'Innovation EPFL / PSE-A / CH-1015 Lausanne / Tel +41 21 619 10 10
FRANCE Savoie Technolac / BP 352 / F- 73372 Le Bourget-du-Lac / Tel +33 4 79 44 44 94
AUSTRIA Am Heumarkt 13 / A-1031 Wien / Tel +43 1 712 21 94 0

Follow us