ANGULARJS PROGRAMMING COOKBOOK

Hot Recipes for AngularJS Development

WEB CODE GEEKS

WEB CODE GEEKS
WEB DEVELOPERS RESOURCE CENTER

AngularJS Programming Cookbook

Contents

1 Single Page Apps **1**

1.1 Getting started . . . 1
1.2 Setting up AngularJS . . . 1
1.3 The HTML markup of our app . . . 2
1.4 Download . . . 4

2 AngularJS Form Validation **5**

2.1 View Page . . . 5
2.2 Controller's Functionality . . . 6
2.3 Demo . . . 6
2.4 Download . . . 8

3 AngularJS Routing **9**

3.1 What is routing in a web app . . . 9
3.2 Today's example's concept . . . 9
3.2.1 Defining the module . . . 10
3.3 Demo . . . 11
3.4 Download . . . 12

4 AngularJS Controller **13**

4.1 Required Background . . . 13
4.1.1 What are scopes? . . . 13
4.1.2 What are controllers? . . . 13
4.1.2.1 How to initialize the state of a `$scope` object . . . 13
4.1.2.2 A small example . . . 13
4.1.2.3 Controllers' use case . . . 14
4.2 Our Example . . . 14
4.3 Demo . . . 15
4.4 Download . . . 17

5 AngularJS JSON Fetching 18
5.1 Introduction . . . 18
5.1.1 What is JSON . . . 18
5.1.2 Example's concept . . . 18
5.1.2.1 The JSON file . . . 18
5.1.2.2 The app itself . . . 18
5.2 The Example . . . 19
5.2.1 Loading JSON into $scope . . . 19
5.2.2 Displaying JSON data into a table . . . 20
5.3 Demo . . . 21
5.4 Download . . . 21

6 AngularJS Table 22
6.1 Introduction . . . 22
6.2 Our Example . . . 22
6.3 Demo . . . 24
6.4 Download . . . 24

7 AngularJS ng-src 25
7.1 Introduction . . . 25
7.2 Example's concept . . . 25
7.3 The Example . . . 25
7.3.1 The JSON file . . . 26
7.3.2 Fetching the JSON file . . . 26
7.3.3 Displaying the JSON file . . . 26
7.4 Demo . . . 27
7.5 Download . . . 28

8 AngularJS Data Binding 29
8.1 Introduction . . . 29
8.1.1 The concept . . . 29
8.1.2 What you need to know . . . 29
8.1.2.1 Templates . . . 29
8.1.2.2 ngModel . . . 30
8.2 The Example . . . 30
8.3 Demo . . . 30
8.4 Download . . . 31

Preface

AngularJS (commonly referred to as "Angular") is an open-source web application framework maintained by Google and by a community of individual developers and corporations to address many of the challenges encountered in developing single-page applications. It aims to simplify both the development and the testing of such applications by providing a framework for client-side model-view-controller (MVC) and model-view-viewmodel (MVVM) architectures, along with components commonly used in rich Internet applications.

The AngularJS library works by first reading the HTML page, which has embedded into it additional custom tag attributes. Angular interprets those attributes as directives to bind input or output parts of the page to a model that is represented by standard JavaScript variables. The values of those JavaScript variables can be manually set within the code, or retrieved from static or dynamic JSON resources. (Source: https://en.wikipedia.org/wiki/AngularJS)

In this ebook, we provide a compilation of AngularJS based examples that will help you kick-start your own web projects. We cover a wide range of topics, from Single Page Apps and Routing, to Data Binding and JSON Fetching. With our straightforward tutorials, you will be able to get your own projects up and running in minimum time.

About the Author

WCGs (Web Code Geeks) is an independent online community focused on creating the ultimate Web developers resource center; targeted at the technical architect, technical team lead (senior developer), project manager and junior developers alike.

WCGs serve the Web designer, Web developer and Agile communities with daily news written by domain experts, articles, tutorials, reviews, announcements, code snippets and open source projects.

You can find them online at http://www.webcodegeeks.com/

Chapter 1

Single Page Apps

Single Page Apps are becoming increasingly popular as the demand for highly responsive apps is increasing. We could be using Ajax and Javascript to do this, but we will be using Angular as it is a structural framework for dynamic web-apps, efficient and easy to use, as described by the people who built it.

We're going to create a simple app with only a homepage, an about page and a date page, so as to not need to refresh the page to view changes, but have it reflect them immediately.

We will use:

- AngularJS
- Bower
- Twitter's Bootstrap

1.1 Getting started

To go on with the project you might want to download Bootstrap and also Bower (here you will find every information you need on how to do that).

Firstly, we create a folder named `SinglePageApp` (you can name it anything you wish), an HTML file named `index.html` and a Javascript file named `app.js` inside of it, and we're good to go.

1.2 Setting up AngularJS

First we ask Bower to install the Angular package by executing this in the terminal (make sure you're in your project's directory):

```
bower install angular
```

Then, again using Bower we install the Angular-Route package by executing:

```
bower install angular-route
```

After executing these commands we will notice that in our project's directory a `bower_components` directory is added, inside of which are two folders, one named `angular` and one named `angular-route`. We will use them later. Then we open the `app.js` and write the JavaScript code for creating the Angular module, while adding `ngRoute` dependency to it. It should look like this:

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

Now is time to define routes by using the `config()` function, provided by angular.module. Right under the code for the angular module, in the `app.js` file we place this code:

```
app.config(function($routeProvider){
        $routeProvider

                //default page
                .when('/',{
                        templateUrl : 'pages/homepage.html',
                        controller  : 'Homepage'
                })

                //about page
                .when('/about',{
                        templateUrl : 'pages/about.html',
                        controller  : 'About'
                })

                //date page
                .when('/date',{
                        templateUrl : 'pages/date.html',
                        controller  : 'Date'
                });
});
```

We have injected `$routeProvider` as a parameter to the function. Now the `when()` function of the `$routeProvider` can be used to configure the routes. This function takes two parameters: the route name and the route definition object which in itself contains various details for a route. We will use only two of those properties: the `templateURL` which is a relative location of the view file, starting from `index.html`; and the controller associated with the view.

We have reached the point where we should create the controllers for the different views. First we create a directory named `controllers` in the `js` folder, and inside of it we create a JavaScript file named `controllers.js` where we will put this code snippet:

```
app.controller('Homepage',['$scope',function($scope){
        $scope.homepage = "Homepage";
}]);

app.controller('About',['$scope', function($scope){
        $scope.about = "Lorem ipsum...";
}]);

app.controller('Date',['$scope', function($scope) {
        $scope.now = new Date();
}]);
```

Don't forget to script the file `controllers.js` in the `index.html` file. You can even place the above code in the `app.js` file but it is not recommended as it could reduce the readability of the code and in heavier applications could complicate stuff.

1.3 The HTML markup of our app

An important thing to do if we want our app to function properly is to link and script all the files we've downloaded. First we link Bootstrap's CSS file by adding this right under the `</title>` tag:

```
<link rel="stylesheet" href="css/bootstrap.min.css">
```

Now let's script Bootstrap's, Angular's and Angular-Route's JavaScript files by adding this code after the `<link>` tag:

```
<script src="js/bootstrap.min.js"></script>
<script src="bower_components/angular/angular.min.js"></script>
<script src="bower_components/angular-route/angular-route.min.js"></script>
```

Also we script the `app.js` file, as it contains our angular module and also other necessary things, like this:

```
<script src="js/app.js"></script>
```

You might want to place all this in the `<body>` section, right before the `</body>` tag, for performance reasons, as the browser will render the HTML markup first and then the JavaScript code is loaded. Please bear in mind that the scripting of `angular-route` and `app.js` should come after the scripting of `angular`, and `controller.js` should come after the scripting of `app.js`.

Now let's create a directory named `pages` inside of which we will place the HTML files for the views. Beware that the files for the views should not be complete HTML files, they should only have the markup for the specific view.

In the `homepage.html` file we will only place the `{{homepage}}` expression.

The About page will be a typical descriptive page, so we will only need this code:

```
{{ about }}
```

The date page will show us the date of today. The code will go like this:

```
{{ now | date:'medium' }}
```

While by now you know that what we have placed inside the double curly brackets is an expression, it is worth noting that `| date:'medium'` is a filter, a short way to format some of the most used expressions, such as dates, numbers, currency etc. Angular gives us several built-in filters as well as an easy way to create our own. This particular filter formats the date so as to show the month, date and then the year, and also the time in hours, minutes and seconds in AM/PM format.

Time to have a look at `index.html`. We will have to tell Angular in which part of the application it should be active. You saw that when declaring the angular module we named it `app`. To tell it where it should be active we add the attribute `ng-app="app"` in the tag and everything inside of it turns into an AngularJS application. In our case, as the whole page will be an `ng-app` it is better to place the attribute in the `<html>` tag or in the `<body>` tag. I'm choosing the second.

In the `<body>` section we will add the buttons, one for each view. We will use Bootstrap to create them easily but also to make them look stylish. The code will go like this:

```
<a href="#/"><button class="btn btn-danger">Homepage</button></a>
<a href="#/about"><button class="btn btn-success">About</button></a>
<a href="#/date"><button class="btn btn-warning">Date</button></a>
```

Take a look at the `href` attribute in each of the links. Familiar? It is the route we specified when we used `when()` function, only it has a # (hashbang) before it. The class attribute you see in the `<button>` tags is a class provided by Bootstrap, which makes them look fancy and have specific colors (red for `btn-danger`, green for `btn-success` and orange for `btn-warning`).

But where is the app going to display the views we created? We will tell it where, by placing a `` element containing the attribute `ng-view`. It will look like this:

```
<div class="row">
 <div ng-view>
   <!-- Views go here -->
 </div >
</div >
```

I have placed it inside another `div` element, but you can place it anywhere in the application's body. AngularRoute gives it the opportunity to completely replace the `<div ng-app>` element with the element `<ng-app> </ng-app>`, but is not recommended if your target users will use IE as it does not support the element version.

You may have noticed the "row" attribute I gave to the `div` containing the `ng-view` element. It is there for styling purposes only. To make the website look a little more fancy, you create a `styling.css` file inside the `css` folder (don't forget to link it), and place this code inside it:

```
.row{
        text-align: center;
        background-color: steelblue;
        font-family: "Georgia", serif , sans-serif;
        font-size: 30px;
}
```

You can now open your Single-Page App and it will show you different views each time you click the buttons, without reloading the page, looking like this:

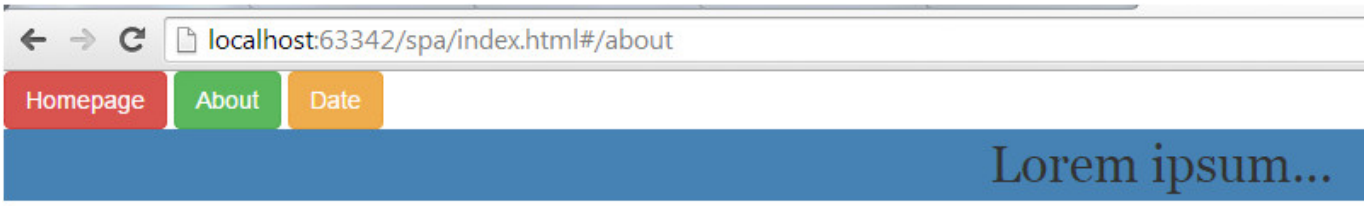

Figure 1.1: Single Page App when the About button is pressed.

Such is the magic of AngularJS.

1.4 Download

Download You can download the full source code of this example here: **SinglePageApp**

Chapter 2

AngularJS Form Validation

One of AngularJS advantages is the simplicity it provides for validating forms. Today's example demonstrates a simple way to implement client-side validation using the AngularJS form properties.

Suppose a form with two fields, username and email and two buttons, one for reset and the other with a submit role. Our example's concept is that both of the form's fields are required, so the form cannot be submitted if either of these is empty or has invalid format (i.e. the email).

2.1 View Page

The best way to take advantage of Angular's abilities, according to form validation, is to attach a controller to our form.

index.html

```
<!DOCTYPE html>
<html>
<head>
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/2.3.2/css/bootstrap. ↩
    min.css">
<script src="http://ajax.googleapis.com/ajax/libs/angularjs/1.2.26/angular.min.js"></script ↩
    >
<script src="script.js"></script>
<link rel="stylesheet" href="style.css" >
</head>

<body>
<h2>AngularJS Form Validation Example</h2>
<form ng-controller="validationCtrl" name="sampleForm" novalidate>

                <label>Username:</label>
                <input type="text" class="form-control" name="username" ng-model="username" ↩
                     required >
                <span ng-show="sampleForm.username.$error.required">Username is required.</ ↩
                    span>


                <label>Email:</label>
                <input type="email" class="form-control" name="email" ng-model="email"  ↩
                    required>
                <span ng-show="sampleForm.email.$error.required">Email is required.</span>
                <span ng-show="sampleForm.email.$error.email">Invalid email address.</span>


                <button class="btn btn-link" ng-click="reset()">Reset</button>
```

```
                <input type="submit" class="btn btn-primary" ng-disabled="sampleForm. ↩
                    $invalid" ng-click="checkData()">

</form>
</body>
</html>
```

As you make clear from the `head`'s definition, I decided to separate HTML from Javascript and CSS.

Line 12 declares clarifies that our form is attached to `validationCtrl` and yes, your guess is right, the `script.js` will contain the controller's logic, which will be nothing more than a handler for our form's buttons.

A typical client-side validation process contains error messages, for the cases of invalid (email) or blank input fields.

In Angular, there are discrete form states, such as `$dirty` or `$invalid`, for user-interaction check with the form and for invalid input fields check, respectively. To read more about form properties, refer to the official documentation.

Generally, this means that we partially want extra messages to be displayed in our webapp. AngularJS provides the `ng-show` directive for this purpose, which, in conjunction with the form's states, results to the client-side validation. Lines 16 and 21-22 implement the validation in the AngularJS way.

We want our submit button to be disabled, while the form is invalid (line 26). AngularJS provides `ng-disabled` directive to make this feasible.

Finally, we clarify that we want our controller to run a specific function for each of the form's buttons, using `ng-click` directive.

2.2 Controller's Functionality

When our form has a valid state, the submit buttons gets enabled. Suppose there is a predefined acceptable username and email from the webapp and that this the purpose of the reset button: to turn our form to its acceptable format. On the other hand, the submit has to check if the form contains the correct data.

According to AngularJS, the glue between the application controller and the view, is the `$scope` object.

`script.js`

```
function validationCtrl($scope) {
   var validUsername = "Thodoris Bais";
   var validEmail = "thodoris.bais@gmail.com";

   $scope.reset = function(){
                $scope.username = validUsername;
                $scope.email = validEmail;
   }

   $scope.checkData = function() {
                if ($scope.username != validUsername || $scope.email != validEmail) {
                        alert("The data provided do not match with the default owner");
                } else {
                        alert("Seems to be ok!");
                }
        }
}
```

2.3 Demo

Let's run our app:

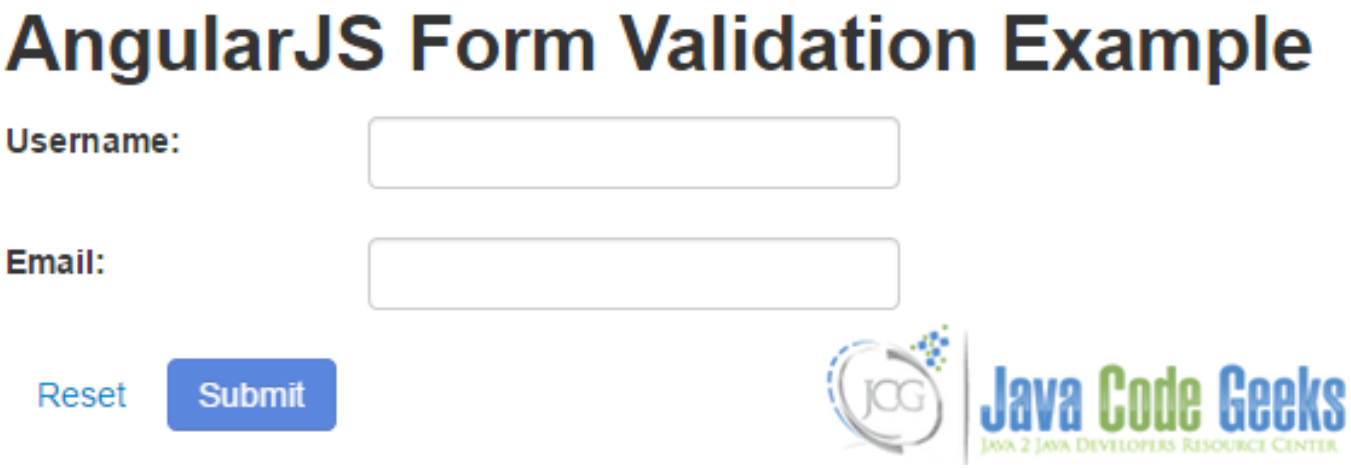

Figure 2.1: Initial screenshot of webapp

After leaving the email field blank, here's the job of `ng-show`:

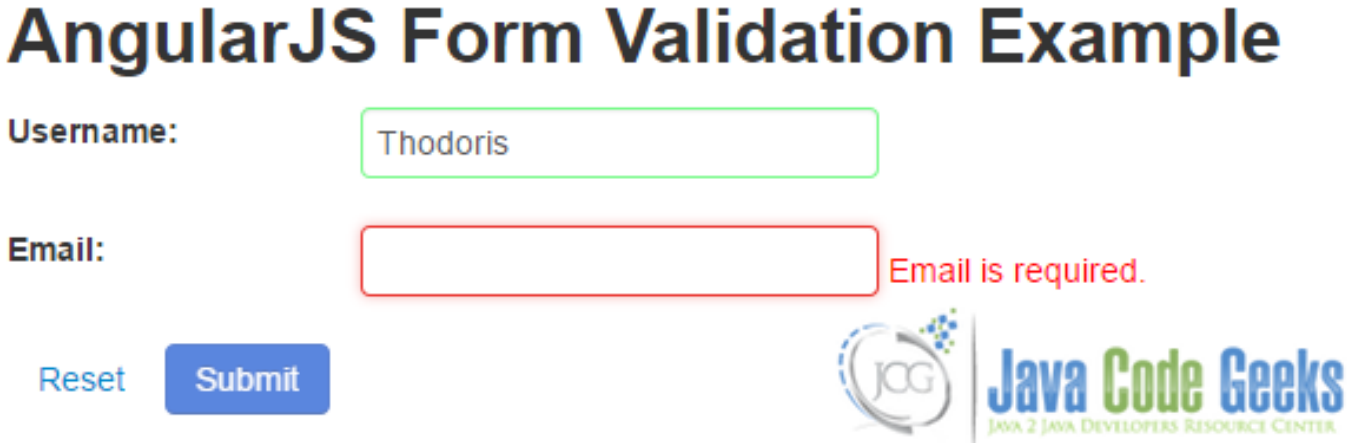

Figure 2.2: Email cannot be blank

Having provided the valid input data, we firstly notice that the submit button get enabled (both fields have now a green/valid border) :

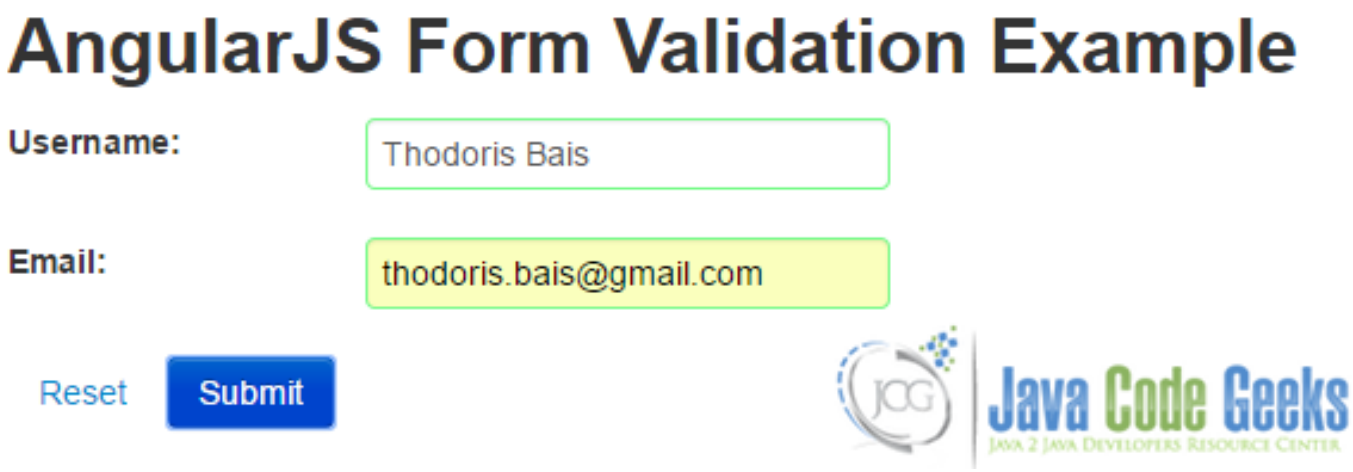

Figure 2.3: Providing valid data

2.4 Download

Download You can download the full source code of this example here: **angularjs_form_validation.zip**

Chapter 3

AngularJS Routing

This example is related with routing in AngularJS. What I firstly found out, while searching over the net for similar resources, is that there isn't yet any simple example, suitable for an Angular newbie. So, I'll try to keep today's post as simple as I can.

3.1 What is routing in a web app

Generally, web applications make use of readable URLs that describe the content that resides there. A common example could be clicking on a homepage's link: this means that a back-end action is being executed, that results to a different view on the client-side. We often confirm similar situations after interacting in the root of a web app (`/` or `index.html`), by noticing a change to the browser's url bar.

3.2 Today's example's concept

In this example we will demonstrate a simple page navigation application. Suppose a homepage with two links and each one of them will redirect to a specified page.

To get a better understanding of our concept, we ll here implement an inline navigation. This means that we want our pages *content to be displayed inside the initial/home page*.

AngularJS provides the `ngView` directive to implement the fore-mentioned functionality. Specifically, `ngView` directive complements the $route service by including the rendered template of the current route into the main layout file. That is, each time the current route changes, the included view changes with it according to the configuration of the `$route` service.

So, keeping in mind that our `index.html` contains a simple sentence with two links provided, assume we want to display the rendered templates (according to the clicked links) below that sentence; this should seem like:

```
Jump to the <a href="#first">first</a> or <a href="#second">second page</a>
<div ng-view>
```

As you see, the anchors' targets are already named, so what is left, is to configure the respective routings for Angular. So, at this point we should have a complete homepage:

`index.html`

```
<!DOCTYPE html>
<html>
<head>
        <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.2/css/ ↩
            bootstrap.min.css">
        <script src="http://ajax.googleapis.com/ajax/libs/angularjs/1.2.26/angular.min.js"> ↩
            </script>
```

```
        <script src="http://ajax.googleapis.com/ajax/libs/angularjs/1.2.26/angular-route. ↩
            min.js"></script>
        <script src="script.js"></script>
</head>

<body ng-app="RoutingApp">
        <h2>AngularJS Routing Example</h2>
        Jump to the <a href="#first">first</a> or <a href="#second">second page</a>


</body>
</html>
```

In AngularJS, we can use the `ngRoute` module for routing and deeplinking services and directives.

Here are some important points before going on:

- In order to use the `ngRoute` module, you have to include `angular-route.js` to your app, which, obviously, has to be loaded, after including the `angular.js` script.
- The routings we need, have to be configured inside the module's functionality, so it would be easier to define our module in a separate file, `script.js`.
- You have to provide same name for the `ng-app` tag (in html file that contains the Angular app) and the module's definition.

3.2.1 Defining the module

Now, let's define the Angular module, by providing our app's name (as in `index.html`) and declaring that it depends on the `ngRoute` module; the last words mean that we have to "inject" `ngRoute` into our module (`script.js`), like this:

```
angular.module('RoutingApp', ['ngRoute']);
```

That's why we had to include the `angular-route.js` file in our app. In order to use `ngRoute`, we have to call the `angular.config` method:

```
angular.module('RoutingApp', ['ngRoute'])
    .config(function() {

     });
```

As you noticed, I also created an anonymous function inside the method, 'cause, otherwise, we 'll get a script error from our browser, as `angular.config` method **requires calling it with a function**.

From the official documentation, we can use the `$routeProvider` to configure Angular's routes, so we should pass `$routeProvider` as a parameter in our anonymous function:

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

     }]);
```

The `$routeProvider`'s method to add a new route definition to the `$route` service is `when(path, route)`:

- `path` corresponds to the requested from the client path.
- `route` is an object parameter and contains mapping information that have to be assigned while matching the requested route (i.e. we may want to handle the newly registered route with a specific controller; `controller` property is responsible for this scope).

You can read about the rest of `route`'s object properties, but as I fore-mentioned, here, we 'll implement a simple routing between two html files, so, I'll only use the `templateUrl`.

Please take a look at the module's final structure:

`script.js`

```
angular.module('RoutingApp', ['ngRoute'])
        .config( ['$routeProvider', function($routeProvider) {
                $routeProvider
                        .when('/first', {
                                templateUrl: 'first.html'
                        })
                        .when('/second', {
                                templateUrl: 'second.html'
                        })
                        .otherwise({
                                redirectTo: '/'
                        });
        }]);
```

Now, let me explain: the first `when` means that when the `/first` is requested as a route, the `first.html` will be loaded. Same for the "second".

The `$routeProvider`'s `otherwise(params)` method sets route definition that will be used on route change when no other route definition is matched. Practically, this means, that if the client requests a route that isn;t defined in the `when` method, this method will be executed. Imagine this as a general if-else statement.

In our case, I assume we want to display just the homepage (`'/'`), when no route definition gets matched.

3.3 Demo

Firstly, please take a look at this post, just to understand why you should deploy this app in a local server rather than just executing it in a browser.

Access the web app from your local server:

Figure 3.1: Homepage of the app

Now, click on "first":

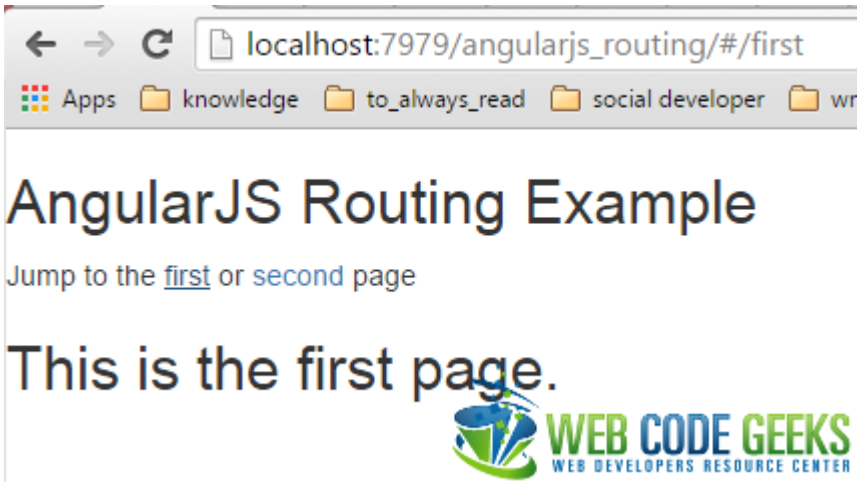

Figure 3.2: App's "first" page

Same, for the "second":

Figure 3.3: App's "second" page

3.4 Download

Download You can download the full source code of this example here: **angularjs_routing.zip**

Chapter 4

AngularJS Controller

Hello! In today's example we 'll see how do Angular's `controllers` work.

To get into this, I chose a simple form concept, where the user is prompt to insert his username. This updates an informative message (i.e. "Your username is "); I'll also include a reset button, for demonstration purposes.

4.1 Required Background

Before getting into the technical part of this example, let me introduce you the `$scope` element.

4.1.1 What are scopes?

AngularJS supports the MVC pattern, with the `$scope` object associated with the application model. In fact, it's the glue between View and Controller. In addition, they hold the Model data that we need to pass to View and use Angular's two way data binding to bind model data to View.

Their responsibility is to initialize the data that the View needs to display. Scopes are arranged in hierarchical structure which mimic the DOM structure of the application.

4.1.2 What are controllers?

Generally, an Angular controller is a JavaScript constructor function that augments the `$scope` object. It can be attached to the DOM through the `ng-controller` directive, where Angular instantiates a new Controller object, using the specified controller's constructor function. Using a controller, we can instantiate a new child scope as an injectable parameter to controller's function. The injected parameter is accessible through `$scope`.

4.1.2.1 How to initialize the state of a `$scope` object

When creating an application, we need to set up the initial state for the Angular `$scope`. This can be feasible by attaching properties to the `$scope` object. These properties contain the View (the Model, which will be presented by the View). All the `$scope` properties will be available to the template at the point in the DOM, where the controller is registered.

4.1.2.2 A small example

We here create a HelloController, which attaches a `helloMessage` property containing the string "Hello World!" to the `$scope`:

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

myApp.controller('HelloController', ['$scope', function($scope) {
  $scope.helloMessage= 'Hello World!';
}]);
```

What is actually done, is an Angular module creation (`myApp`) for our app. We then add the controller's constructor function to the module, using the `.controller()` method. This keeps the controller's constructor function out of the global scope.

Next, we attach our controller to the DOM using the `ng-controller` directive. The `helloMessage` property can now be data-bound to the template:

```
< div ng-controller="HelloController" >
  {{ helloMessage }}
</div >
```

4.1.2.3 Controllers' use case

Generally, we usually use controllers either to setup the initial state of the `$scope` object or to add behavior to the `$scope` object.

Cases that we cannot use controllers are to:

- Manipulate DOM (controllers should contain only business logic).
- Format input; use angular form controls instead.
- Filter output; use angular filters instead.
- Share code or state across controllers; use angular services instead.
- Manage the lifecycle of other components (i.e. to create service instances).

4.2 Our Example

As I fore-mentioned, a username text field, with a reset button and an updateable message, related to the controller's logic.

`index.html`

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>AngularJS Controller Example</title>

  <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/2.3.2/css/ ↩
      bootstrap.min.css">
  <script src="http://ajax.googleapis.com/ajax/libs/angularjs/1.2.26/angular.min.js"></ ↩
      script>
  <script src="script.js"></script>
  <link rel="stylesheet" href="style.css" >
</head>

<body ng-app="myApp">
        <h2>AngularJS Controller Example</h2>

                <form>
                <label>Name:</label>
```

```
                <input type="text" class="form-control" name="username" ng-model="username" ↩
                    >
                <button class="btn btn-link" ng-click="reset()">Reset</button>

                Your username is {{username}}
                </form>

</body>
</html>
```

As line 9 clarifies, our controller's functionality will be defined into the `script.js` file. In line 13, we declare our Angular application in the name of `myApp`. Lines 15-23 contain the important `div` which interacts with `UserController` (this, in conjunction with line's 9 included script, means that our controller's logic is contained int the javascript file.

What else is important here, from Angular's perspective, is the `ng-model` attribute of the text field, in line 18. This is actually a directive that binds our text field to the username property on the scope, with the help of UserController, which is created and exposed by this directive.

That is, the text field's value can be changed inside the controller, by accessing the username property from the scope object (i.e. `$scope.username`).

What about line 19 and `ng-click`? Well, using the `ng-controller` directive, we bind the `div` element including its children to the context of `UserController` controller. The `ng-click` directive will call the `reset()` function of our controller, when the reset button is clicked.

In line 21, there is the message that we earlier discussed, where, in general, the double curly braces notation (`{{ }}`) binds expressions to elements.

Enough said for our View's implementation. Let's now code the Controller and then explain the necessary details.

`script.js`

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

myApp.controller('UserController', ['$scope', function($scope) {
        $scope.username = 'unknown';

        $scope.reset = function() {
        $scope.username = '';
    };
}]);
```

At first, we have to define that this script is about an Angular module; especially the one defined in our `index.html`, the one named as `myApp` application. We then define the controller's name and initialize the `username` with the value of *unknown*. That is, when the application is loaded to the browser, the text field will contain a predefined value, but on user change, `ng-model` interferes with the controller and updates the username's value (the curly braces variable binded in the View).

Obviously, the reset function is used to empty the text fiel's content.

4.3 Demo

Let's now display a quick demo. This is the initial view:

Figure 4.1: Initial state of the app

If we hit the reset button, our text field gets empty:

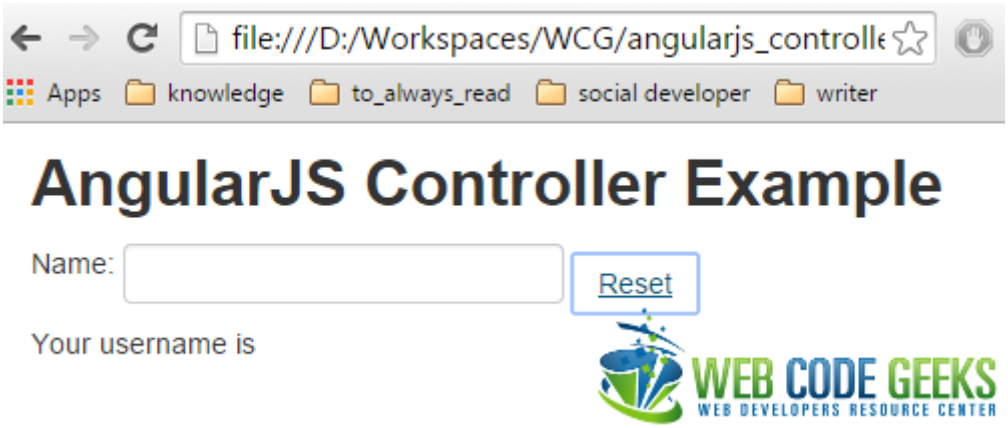

Figure 4.2: The reset button

Finally, here is the updated message, which, in conjuction with the user input, result to a dynamic view in our application:

Figure 4.3: User input

4.4 Download

Download You can download the full source code of this example here: **angularjs_controller.zip**

Chapter 5

AngularJS JSON Fetching

Hello there! Today's example's about displaying data from a JSON file to an Angular.js application.

5.1 Introduction

5.1.1 What is JSON

JSON is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers.

It is built on two structures:

- A collection of name/value pairs. In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array.
- An ordered list of values. In most languages, this is realized as an array, vector, list, or sequence.

5.1.2 Example's concept

In this example we will demonstrate a simple way to retrieve a `JSON` file and display the respective information to our Angular application, in a user-friendly way. Specifically, we *ll here deal with countries* population around the world.

5.1.2.1 The JSON file

I have already prepared a **countries json file**, which provides codenames and population data for several countries around the world, like this:

5.1.2.2 The app itself

So, what's the plan?

- Fetch the `JSON`.
- Display `JSON`'s information in a more human-readable way.

Fetching the JSON at first, means that this has to be done each time the app is being loaded to the browser. That is, the corresponding action will take place into the head's script.

Ok, that was the easy part, what about the difficult one? The fore-mentioned "action" is actually a call to the $http service, a core Angular service that facilitates communication with the remote HTTP servers via the browser's XMLHttpRequest object or via JSONP.

Practically, this means that you have to deploy your app in a web server, rather than executing it in a browser. For further details about this fact, please consult this post.

The general usage of the $http service is a single argument (configuration object) that is used to generate an HTTP request. It returns a promise with two $http methods: success and error.

Here's how a simple GET request looks like:

```
// Simple GET request example :
$http.get('/someUrl').
  success(function(data, status, headers, config) {
    // this callback will be called asynchronously
    // when the response is available
  }).
  error(function(data, status, headers, config) {
    // called asynchronously if an error occurs
    // or server returns response with an error status.
  });
```

We'll only make use of the success and guess what?! On service's successful call/execution, we want to load our JSON file into a global variable, in order to be accessible all over the app and yes, your guess was right, $scope is the global variable that we 're searching for!

In addition, the $http core service, provides shortcut methods, where the only requirement is the URL that has to be processed, whereas the request data must be passed in for POST/PUT requests:

```
$http.get('/someUrl').success(successCallback);
```

5.2 The Example

Time for action!

5.2.1 Loading JSON into $scope

According to the fore-mentioned notes, we have to load our JSON file into a $scope variable, let's say "countries":

```
$http.get('countries.json').success(function(data) {
    $scope.countries = data;
});
```

Obviously, this service call has to be a part of an Angular's app controller definition, so, assuming that we named our angular app as "countryApp", here's the updated format of our service call:

```
countryApp.controller('CountryCtrl', function ($scope, $http) {
   $http.get('countries.json').success(function(data) {
       $scope.countries = data;
   });
});
```

If you need further assistance according to Angular Controllers, please take a look at this post.

5.2.2 Displaying JSON data into a table

Now that we 've loaded all the JSON's data into `$scope.countries`, let's display them to a table with three columns: code, name, population. This fact is translated into two requirements:

- We have to find a way to repeatedly parse all the data from `$scope.variables` (as we obviously don't want to exercise our handwriting for more than 70 separate countries).
- We want to divide each country's data, into code, name and population, in order to display each one in the corresponding table's column

Defining the table's headers is very easy:

```
<th>Code</th >
        <th>Country</th >
        <th>Population</th >
</tr >
```

Now, to fullfil the first requirement, we 'll use Angular's `ngRepeat` directive, which instantiates a template once per item from a collection. Each template instance gets its own scope, where the given loop vriable is set to the current location item, and `$index` is set to the item index or key.

In our case, to repeatedly loop over each country, we have to assume that each country is a table row:

```
<tr ng-repeat="country in countries">
```

Fullfilling the second requirement is now easier, that the country processed each time can be accessed from the `country` variable:

```
< tr ng-repeat="country in countries | orderBy: 'code' ">
        < td>{{country.code}}</td >
        < td>{{country.name}}</td >
        < td>{{country.population}}</td >
< / tr>
```

Here is the final structure of our app:

`index.html`

```
<html ng-app="countryApp">
  <head>
    <meta charset="utf-8">
    <title>Angular.js JSON Fetching Example</title>
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.2/css/ ↩
        bootstrap.min.css">
        <script src="http://ajax.googleapis.com/ajax/libs/angularjs/1.2.26/angular.min.js"> ↩
            </script>
    <script>
      var countryApp = angular.module('countryApp', []);
      countryApp.controller('CountryCtrl', function ($scope, $http){
        $http.get('countries.json').success(function(data) {
          $scope.countries = data;
        });
      });
    </script>
  </head>
  <body ng-controller="CountryCtrl">
        <h2>Angular.js JSON Fetching Example</h2 >
    <table >
      <tr >
        <th>Code</th >
                <th>Country</th >
```

```
        <th>Population</th >
      </tr >
      <tr ng-repeat="country in countries | orderBy: 'code' " >
        <td>{{country.code}}</td >
                <td>{{country.name}}</td >
        <td>{{country.population}}</td >
      </tr >
    </table >
  </body>
</html>
```

5.3 Demo

Let's run it in a local server.

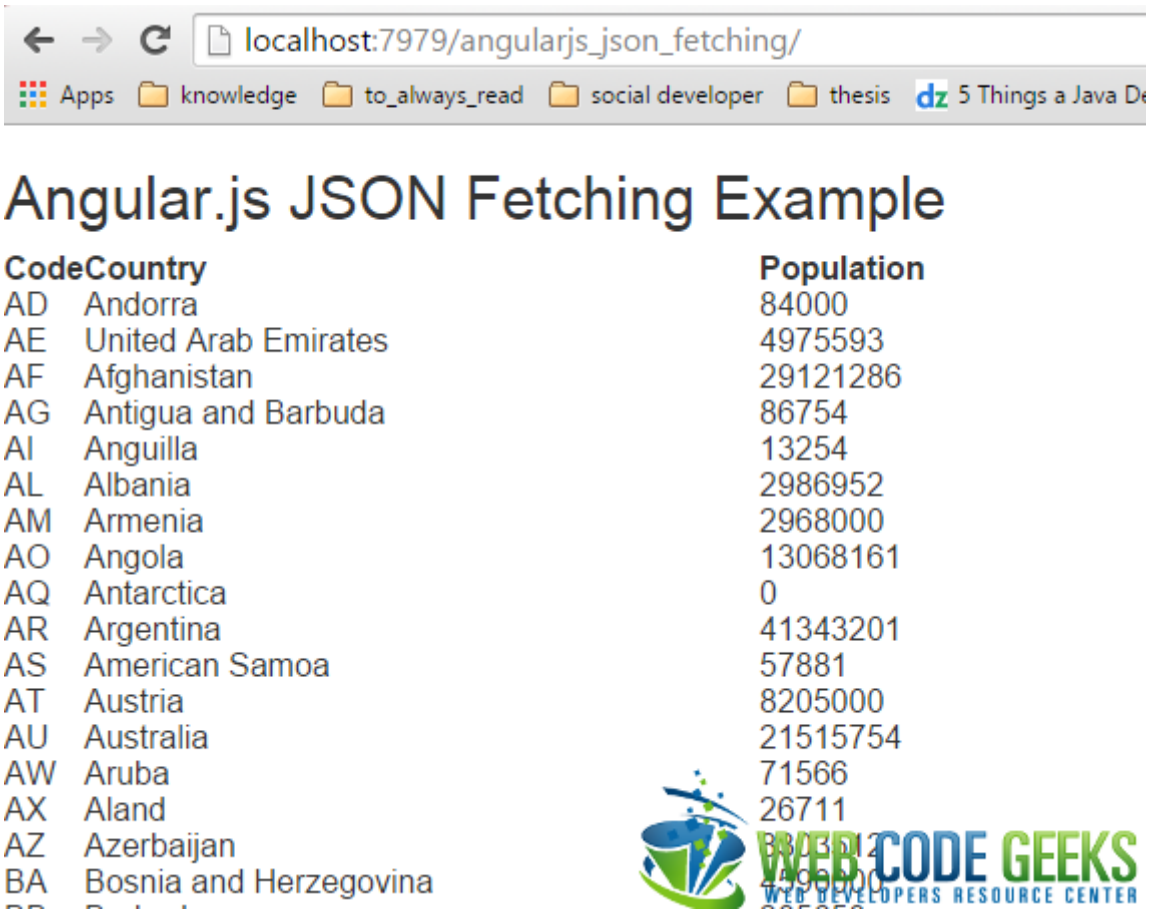

Figure 5.1: App screenshot

5.4 Download

Download You can download the full source code of this example here: **angularjs_json_fetching.zip**

Chapter 6

AngularJS Table

Hi there! Today we 'll examine a simple table solution, using the `Angular.js` framework. Suppose we want to display a list of persons, accompanied with their hobbies, in the gentle packaging of an `HTML table`. Let's see the way over it!

6.1 Introduction

If you 're not enough experienced with Angular, you should first take a glance at a previous example of mine, where the required background for this example, is provided (paragraphs 1.1 and 1.2).

Now that you have a basic understanding of `scopes` and `cotrollers`, let's solve our problem!

6.2 Our Example

To begin with, let's again separate our View from our Controller (MVC pattern). That is, we 'll provide our view in `index.html`, whereas the controller's functionality will remain in `script.js` file.

Having understood the `MVC pattern`, too, we need our controller to pull the necessary data from the `Model` and "feed" the view with them.

I here want to demonstrate a sample table implementation, not an MVC example, so, to make it easier for you, the model's data is created into the controller, instead of being pulled from the model.

Besides the basic functionality that the `$scope` object provides, we can also use it to handle an array with it. We can define an array of persons (with their names and hobbies), into the `$scope` object, like below:

```
$scope.persons = [
          {"name": "Thodoris", "hobby": "Gym"},
          {"name": "George", "hobby": "Fishing"},
          {"name": "John", "hobby": "Basketball"},
                  {"name": "Nick", "hobby": "Football"},
                  {"name": "Paul", "hobby": "Snooker"}
        ];
```

Ok, now that we know what our controller's definition will contain, let's see our view file and we 'll get back to the definition of our script, after it:

`index.html`

```
<html ng-app="tableApp">
  <head>
    <meta charset="utf-8">
    <title>Angular.js Table Example</title>
```

```
    <script src="http://ajax.googleapis.com/ajax/libs/angularjs/1.2.26/angular.min.js"></ ↩
        script>
        <script src="http://ajax.googleapis.com/ajax/libs/angularjs/1.2.26/angular-route. ↩
            min.js"></script>
    <script src="script.js"></script>
  </head>
  <body ng-controller="HobbyCtrl" >
    <table >
      <tr>
        <th>Name</th >
        <th>Hobby</th >
      </tr >
      <tr ng-repeat="person in persons" >
        <td>{{person.name}}</td >
        <td>{{person.hobby}}</td >
      </tr >
    </table >
  </body >
</html>
```

We here defined an Angular app (line 1), in the name of `tableApp` and attached a controller to it (line 9). This means that our view will search for the controller's definition/functionality in the `script.js` file.

Lines 12 and 13 declare our table's headers, inside a table row:

```
<tr >
    <th>Name</th >
    <th>Hobby</th >
</tr >
```

At this point, we have to find a way to repeatedly parse all the data from the controller's `persons` array to our table, as we obviously don't want to exercise our HTML typing.

This results to an extra requirement, too: we want to divide each person's data, into name and hobby, in order to display each person in the corresponding table's column.

In order to meet the fore-mentioned requirements, we 'll use Angular's ngRepeat directive, which instantiates a template once per item from a collection. Each template instance gets its own scope, where the given loop variable is set to the current location item, and $index is set to the item index or key.

In our case, to repeatedly loop over each person, we have to assume that each person is a table row.

Now, we can access the person that is processed each time, using the `person` variable:

```
<tr ng-repeat="person in persons" >
     <td>{{person.name}}</td >
     <td>{{person.hobby}}</td >
</tr >
```

We 've analyzed our view page, so after a small introduction to the controller, here's its final structure:

`script.js`

```
angular.module('tableApp', [])
        .controller('HobbyCtrl', function ($scope){
        $scope.persons = [
          {"name": "Thodoris", "hobby": "Gym"},
          {"name": "George", "hobby": "Fishing"},
          {"name": "John", "hobby": "Basketball"},
                  {"name": "Nick", "hobby": "Football"},
                  {"name": "Paul", "hobby": "Snooker"}
        ];
    });
```

6.3 Demo

Here's a quick demo of the app:

Figure 6.1: App demo

6.4 Download

Download You can download the full source code of this example here: **angularjs_table.zip**

Chapter 7

AngularJS ng-src

Hi there! Today we 're gonna see how to include images in our Angular applications.

7.1 Introduction

In plain HTML, we use the `<img>` tag to insert an image to our site. For example, this could be a sample image insertion:

```
<img src="image.png" alt="A sample image" height="64" width="64">
```

Now, in terms of Angular.js and image insertion, one could easily claim, that we can similarly refer to an image, by using one of the standard Angular.js templates. I suppose, everyone would go for the "markup" one, with which, we can use the double curly brace notation `{{ }}` to bind expressions to elements.

However, using Angular markup like `{{hash}}` in the fore-mentioned form, within an HTML `src` attribute doesn't work well, as the browser will fetch from the URL with the literal text `{{hash}}`, until Angular replaces the expression inside `{{hash}}`.

Angular's `ngSrc` directive solves this problem.

The fore-mentioned "buggy" way to implement is, seems like:

```
<img src="http://www.gravatar.com/avatar/{{hash}}" alt="Sample avatar"/>
```

But the correct way to write it, is by using the `ngSrc` directive:

```
<img ng-src="http://www.gravatar.com/avatar/{{hash}}" alt="Sample avatar" />
```

7.2 Example's concept

For our convenience, we *ll make use of an existing example concept, the one where I demonstrated the Angular.js JSON Fetching. That is, assuming that we have to display some countries* data in our page, we may also want to let the user view the country's flag, too. So, we 'll use the `ngSrc` directive in conjuction with a JSON file. This is not so close to the introduction section, but I 'll cover that usage, too.

7.3 The Example

My previously linked example, according to JSON fetching, describes in depth how to load and "consume" a json file, into an Angular.js application, so I'll quickly get into this.

7.3.1 The JSON file

Here's a minified version of our JSON file:

`countries.json`

```
[
  {
    "name": "Greece",
    "population": 11000000,
    "flagURL": "//upload.wikimedia.org/wikipedia/commons/5/5c/Flag_of_Greece.svg"
  },
  {
    "name": "United Kingdom",
    "population": 62348447,
    "flagURL": "//upload.wikimedia.org/wikipedia/en/a/ae/Flag_of_the_United_Kingdom.svg"
  },
  {
    "name": "United States of America",
    "population": 312247000,
    "flagURL": "//upload.wikimedia.org/wikipedia/en/a/a4/Flag_of_the_United_States.svg"
  }
]
```

That is, it includes one additional key, compared to Angular.js JSON Fetching's one: the `flagURL`.

7.3.2 Fetching the JSON file

We can load a JSON file to an Angular app into the `$scope` variable, by calling the `$http` service. a core Angular service that facilitates communication with the remote HTTP servers via browser's `XMLHttpRequest` or via `JSONP`.

Practically, this means that you have to deploy your app in a web server, rather than executing it in a browser. For further details about this fact, please consult this post.

To do so, we 'll reuse the following code snippet from the existing example (JSON fetching), assuming that:

- The Angular app is named as `contryApp`.
- Our controller is named as `CountryCtrl`.
- Our JSON file is named as `countries.json` and is placed in app's root folder.

```
$http.get('countries.json').success(function(data) {
    $scope.countries = data;
});
```

7.3.3 Displaying the JSON file

Keeping a similar concept as in the Angular.js JSON Fetching Example let's see how to display the country's flag, using the `ngSrc` directive:

`index.html`

```
<html ng-app="countryApp">
  <head>
    <meta charset="utf-8">
    <title>Angular.js Example</title>
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.4/css/ ↩
        bootstrap.min.css">
```

```
    <script src="//cdnjs.cloudflare.com/ajax/libs/angular.js/1.4.0/angular.min.js"></script ↩
        >
    <script>
      var countryApp = angular.module('countryApp', []);
      countryApp.controller('CountryCtrl', function ($scope, $http){
        $http.get('countries.json').success(function(data) {
          $scope.countries = data;
        });
      });
    </script>
  </head>
<body ng-controller="CountryCtrl" >
    <div class="container" >
    <h2>Angular.js ng-src Example</h2 >
    <table class="table table-striped" >
      <tr >
        <th>Country</th >
        <th>Population</th >
        <th>Flag</th >
      </tr >
      <tr ng-repeat="country in countries" >
        <td>{{country.name}}</td >
        <td>{{country.population}}</td >
        <td >
          <img ng-src="{{country.flagURL}}" width="100" >
        </td >
      </tr >
    </table >
    </div >
  </body >
</html>
```

Lines 1, 7-14 and 16, fulfill the assumptions made in JSON fetching section. Let's use an HTML table to group the JSON's data properly. Lines 21-23 define the table's headers for the country name, population and flag.

That is, we want to divide each country's data, into name, population and flag, in order to display each one in the corresponding table's column. For this purpose, we 'll use the `ngRepeat` directive, which instantiates a template once per item from a collection. Each template instance gets its own scope, where the given loop variable is set to the current location item, and `$index` is set to the item index or key.

When it comes to our case, in order to repeatedly loop over each country, we have to assume that each country is a table row.

So, now that the country processed each time can be accessed from the `country` variable, it's easier to understand that lines 26-30 parse each country's name, population and flag to the corresponding table column.

In addition to what is mentioned in the Introduction section of this example, the official documentation for the `ngSrc`, states that the directive can accept any argument of type `template`. That is, it accepts any string which can contain `{{ }}` markup.

7.4 Demo

Let's run it in a local server. _ For this example, the app can be accessed from a browser in following url:_ `http://localhost:8080/angularjs_ngsrc/`

Figure 7.1: App screenshot

7.5 Download

Download You can download the full source code of this example here: **angularjs_ngsrc.zip**

Chapter 8

AngularJS Data Binding

Hey! Today we 'll see how we can easily update our angular views, using ngModel. To begin with, according to angular applications, data-binding is the automatic synchronization of data between the model and view components (for those that didn't get this at all, please take a look over the MVC pattern).

Regarding Angular's implementation, the view is a projection of the model at all times. When the model changes, the view reflects the change, and vice versa.

For the purpose either of a getting a better understanding of the fore-mentioned or the data-binding comparison between classical template systems and Angular's, please refer to the official documentation.

8.1 Introduction

8.1.1 The concept

Today's concept will demonstrate the implementation of data-binding by displaying the user's input into the View, according to the Model. Especially, there are two text fields, one that matches a user's first name, while the second matches the last name and a greeting message, depending on the user's input.

8.1.2 What you need to know

8.1.2.1 Templates

In Angular, templates are written with HTML that contains Angular-specific elements and attributes. Angular combines the template with information from the model and controller to render the dynamic view that a user sees in the browser.

These are the types of Angular elements and attributes you can use:

- Directive - An attribute or element that augments an existing DOM element or represents a reusable DOM component.
- Markup - The double curly brace notation {{ }} to bind expressions to elements is built-in Angular markup.
- Filter - Formats data for display.
- Form controls - Validates user input.

For further details, according to angular templating, please refer to the official documentation.

8.1.2.2 ngModel

The `ngModel` directive binds an input,select, textarea (or custom form control) to a property on the scope using NgModelController, which is created and exposed by this directive.

ngModel has several responsibilities (you can read all of them in the official documentation, as well), but for the purpose of this example, the important one is the responsibility for binding the view into the model, which other directives such as input, textarea or select require.

What is important to know, before diving into the source code of this example, is that `ngModel` will try to bind to the property given by evaluating the expression on the current scope. This means that if the property doesn't already exist on this scope, it will be created implicitly and added to the scope.

8.2 The Example

One file is enough!

`index.html`

```
<!DOCTYPE html>
<html ng-app>
  <head>
    <meta charset="utf-8">
    <title>Angular.js Data Binding Example</title>
    <script src="//cdnjs.cloudflare.com/ajax/libs/angular.js/1.3.15/angular.min.js"></ ↩
        script>
  </head>
  <body >
    <h2>Angular.js Data Binding Example</h2 >
        <table >
          <tr >
                <td>First name:</td >
                <td><input ng-model="firstName" type="text"/></td >
          </tr >
      <tr >
                <td>Last name:</td >
                <td><input ng-model="lastName" type="text"/></td >
          </tr >
    </table >
        <br/ >
    Hello {{firstName}} {{lastName}}
  </body >
</html>
```

First of all, we obviously have to define that this is about an angular application (line 2). Secondly, we build our table view, according to our needs: two rows (first and last name), each one with two columns (the corresponding text field for first and last name). Finally, we display our message in line 21.

The whole process is based on the assignable variables to the `ng-model`. That is, this works as a nested angular controller, binding data to our view. This kind of data can be accessed/displayed using the curly brackets angular templating markup.

Generally, HTML's `input` tag, when used together with `ng-model`, provides data-binding, input state control, and validation

8.3 Demo

Firstly, please take a look at this post, just to understand why you should deploy this app in a local server rather than just executing it in a browser.

Access the web app from your local server:

Figure 8.1: Application's initial state

Now, let's insert some values:

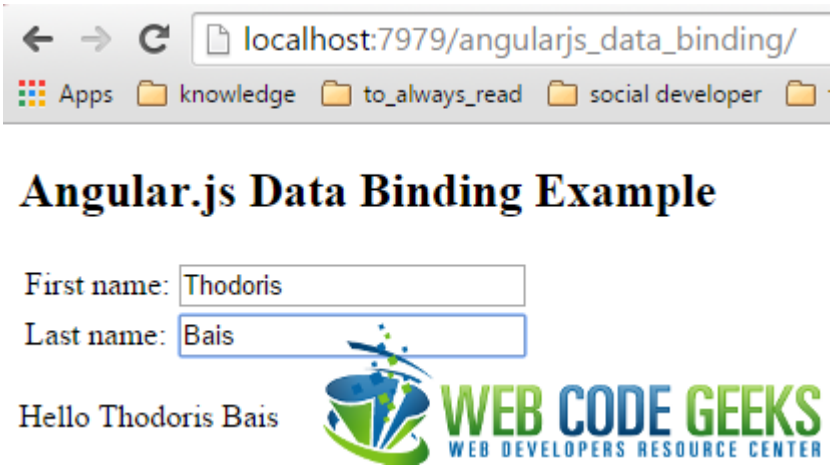

Figure 8.2: Inserting sample data

Hooray, it's giving feedback!

8.4 Download

Download You can download the full source code of this example here: **angularjs_data_binding.zip**