Web Development with AngularJS and Visual Studio

PREREQUISITES

1. A good understanding of JavaScript and C#
2. Microsoft Visual Studio (I used Community Edition 2017.)

INTRODUCTION

In most programming books, the examples are pretty standard and not very interesting – a mockup for a fake school, complete with fake courses, fake professors, and fake students, for instance. I thought, why not create examples that deal with a real subject – something the reader might find interesting, unfamiliar, even educational? To that end, I decided to try combining three of my passions – coding, macro photography, and bugs (of the biological variety.) If you’re one of those unfortunate people who can’t even look at a spider without running away screaming, then, obviously, this book is not for you. If, on the other hand, you think it might be fun to learn a little about insects, spiders, and their relatives, all while learning to code in AngularJS, then please give this book a try.

This book is intended for people with no previous knowledge of Angular, but I’ve tried to include enough in-depth information to make it useful to people who already have some experience with the subject. I’ve tried at all times to be as clear as possible. I hope I’ve succeeded.

IMPORTANT: You should have downloaded the code examples along with the book. You’ll need these files, because they include a lot of photos that are used in the examples.

Now, on with the show!

CHAPTER 1 – THE BASICS

AngularJS is a JavaScript framework developed by Google. It’s based on the MVC pattern (Model, View, Controller). Angular is typically used to build single-page web applications (SPAs). An SPA is an HTML page that updates dynamically without reloads. It makes for a nicer user experience with a more professional feel. Angular also allows the developer to extend HTML by adding new elements and behavior and its modular architecture is designed to make testing easy.

Let’s start with a simple example. Go to the “Example 1” subfolder of the “Chapter 1” folder in the source code and open index.html in a web browser. Here’s the source code for reference:

<!DOCTYPE html>

<html ng-app>

<head>

<meta charset="utf-8" />

<title>Buggy Example 1</title>

<script src="http://code.angularjs.org/1.2.14/angular.js"></script>

</head>

<body>

<h1>Pick a bug. Any bug...</h1>

<select ng-model="selectedName">

<option></option>

<option>Honey Bee</option>

<option>Cranefly</option>

<option>Scorpion</option>

<option>Pregnant Scorpion</option>

</select>

<div ng-show="selectedName">

<h3>You selected: {{ selectedName }}.</h3>

<img ng-src="images/{{ selectedName }}.jpg"></img>

</div>

</body>

</html>

You should see a dropdown menu that asks you to select a bug. When you select a bug from the menu, a message is displayed indicating what bug you picked and a photo of the bug is displayed.

Let’s go through how this example works.

In the second line of the code, you can see that the <html> element has an ng-app attribute:

<html ng-app>

All built-in AngularJS attributes will be prefixed with ng-. The ng-app attribute tells Angular that <html> is the AngularJS root element. In other words, the scope of the AngularJS application will be this element and all its daughter elements. (We’ll talk more about scope later.) ng-app can appear only once in an html file. It can be added to any element, but it’s kind of standard practice to add it to <html>.

At line six, you’ll see that the web page is loading a JavaScript file called angular.js:

<script src="http://code.angularjs.org/1.2.14/angular.js"></script>

This contains the core AngularJS code and is necessary for all the examples in this book to work.

Moving right along, we get to an HTML <select> element, which contains the list of bugs the user can choose from:

<select ng-model="selectedName">

<option></option>

<option>Honey Bee</option>

<option>Cranefly</option>

<option>Scorpion</option>

<option>Pregnant Scorpion</option>

</select>

Notice that the <select> element has an ng-model attribute, which is set to “selectedName”. This “binds” the name “selectedName” to the value of the <select> element and allows us to retrieve the value by referencing that name.

Next, we have a <div> element with an ng-show attribute set to “selectedName”:

<div ng-show="selectedName">

<h3>You selected: {{ selectedName }}.</h3>

<img ng-src="images/{{ selectedName }}.jpg"></img>

</div>

ng-show works like the “visible” HTML attribute. If it’s set to true, then whatever is contained within the <div> element will be displayed on the web page. If it’s false, the contents of <div> will be invisible. The value of the first <option> element of <select> is the empty string. This equates to false in JavaScript. The first option is selected by default, so when the page is first displayed, ng-show will be false and the contents of the <div> element will not be displayed. If you select “Honey Bee” from the drop-down, that becomes the value associated with <select>. In JavaScript, a non-empty string equates to true, so the contents of <div> are displayed.

Look at the next line:

<h3>You selected: {{ selectedName }}.</h3>

The double curly brackets tell Angular to replace {{ selectedName }} with whatever value is bound to “selectedName”. So, in this case, the line becomes:

<h3>You selected: Honey Bee.</h3>

Finally, look at the following line:

<img ng-src=" images/{{ selectedName }}.jpg"></img>

ng-src works just like the “src” attribute in HTML, except that you can use AngularJS binding with it. Just as in the previous line, {{ selectedName }} is replaced with “Honey Bee”, so the <img> element now references an image file named “Honey Bee.jpg” in the “images” subdirectory of the current directory. It just so happens that we have a photo of a honey bee in that directory and it’s named “Honey Bee.jpg”, so it gets displayed.

And that’s it! Cool, huh?

Before continuing, try clicking on the other bugs in the list and read the short descriptions with each one. You might find some of the info interesting.

DIRECTIVES

The special AngularJS attributes, such as ng-model and ng-show, that we added to elements in this example are called directives. A directive is just an attribute that extends the functionality of HTML. As I mentioned earlier, built-in Angular directives like these will always be prefixed with ng-. Later, we’ll see how you can create your own custom directives. You should never prefix your custom directives with ng-, however. That would just confuse any programmers that might have to work with your code in the future.

CONTROLLERS

In an AngularJS application of any size, the data is controlled by an object called, appropriately, a controller. You define a controller by writing its constructor function. AngularJS then uses this function to create the controller when the web page loads.

Take a look at Example 2 in Chapter1:

<!DOCTYPE html>

<html ng-app>

<head>

<meta charset="utf-8" />

<title>Buggy Example 2</title>

<script src="http://code.angularjs.org/1.2.14/angular.js"></script>

</head>

<body ng-controller="BugController">

<h1>Pick a bug. Any bug...</h1>

<select ng-model="selectedName" ng-change="setBug()">

<option></option>

<option>Jerusalem Cricket</option>

<option>Tropical House Cricket</option>

<option>Scorpion</option>

<option>Oleander Aphid</option>

</select>

<div ng-show="selectedName">

<h3>You selected: {{ selectedBug.CommonName }}</h3>

Class: {{ selectedBug.Class }} <br/>

Order: {{ selectedBug.Order }} <br/>

Family: {{ selectedBug.Family }} <br/>

Genus: {{ selectedBug.Genus }} <br/>

Species: {{ selectedBug.Species }} <br/><br/>

{{ selectedBug.Description }} <br/>

<img ng-src="{{ getFilename(selectedBug.Filename) }}"></img>

</div>

<script>

function BugController($scope) {

$scope.bugArray = [

{

CommonName: "Jerusalem Cricket",

Class: "Insecta",

Order: "Orthoptera",

Family: "Stenopelmatidae",

Genus: "Stenopelmatus",

Species: "",

Description: "These odd-looking insects are not \"true crickets\", because they aren't in the family Gryllidae. They are in the same order as crickets and grasshoppers, though. They're found in the western U.S. and Mexico.",

Filename: "Jerusalem cricket"

},

{

CommonName: "Tropical House Cricket",

Class: "Insecta",

Order: "Orthoptera",

Family: "Gryllidae",

Genus: "Gryllodes",

Species: "sigillatus",

Description: "I'm showing you the rear end of this cricket, because I wanted you to see the ovipositor. That's the long, thin rod that looks remarkably like a hyperdermic needle. The cricket uses it to lay eggs. It inserts the ovipositor into soil and literally injects an egg. Many, but not all, insects have ovipositors and they come in many different varieties. Some have even been modified to form stingers and are no longer used for laying egss. (The eggs come out of different opening.)",

Filename: "Tropical House Cricket - Gryllodes sigillatus"

},

{

CommonName: "Scorpion",

Class: "Arachnida",

Order: "Scorpiones",

Family: "",

Genus: "",

Species: "",

Description: "A scorpion's stinger is in its tail (which is why that's what I'm holding it by.)",

Filename: "Scorpion"

},

{

CommonName: "Oleander Aphid",

Class: "Insecta",

Order: "Hemiptera",

Family: "Aphididae",

Genus: "Aphis",

Species: "nerii",

Description: "Aphids feed on the sap of plants. They excrete a sugary liquid called honeydew, which is consumed by many other insects, especially ants. When an aphid sticks its sucking mouthparts into a plant, the pressure of the sap literally forces the honeydew out of the aphid's anus. (See? I told you these example programs were going to be educational!)",

Filename: "Oleander Aphid"

}

];

// Search bugArray for the bug with CommonName equal to name

$scope.getBug = function(name) {

for (i in $scope.bugArray){

if (name == $scope.bugArray[i].CommonName)

return $scope.bugArray[i];

}

};

$scope.setBug = function() {

$scope.selectedBug = $scope.getBug($scope.selectedName);

};

$scope.getFilename = function(filename) {

if (filename) {

return "images/" + filename + ".jpg";

} else {

return '';

}

}

}

</script>

</body>

</html>

The first thing to notice is that I’ve changed the items in the selection list. In keeping with my goal of teaching a bit of bug biology in the course of these tutorials, I’m going to try to use a different set of photos with each example.

When you run this example and select one of the bugs from the list, some info about the bug is displayed in addition to the photo.

Let’s see how this works. In the code for this example, an ng-controller attribute has been added to the <body> element:

<body ng-controller="BugController">

The constructor function for BugController is defined in the script at the bottom of the file. Notice that this function has one parameter: $scope. The scope of a block of code is the portion of the application that the code has access to. Angular automatically creates a global scope for the application called $rootScope when it loads the application. In the first example, the “selectedName” variable is defined on the $rootScope. When AngularJS creates a controller, it automatically creates a new scope specifically for that controller. This new scope is nested inside the $rootScope. The $scope parameter is how we access the controller’s scope.

You may be wondering how this parameter gets passed to the controller. After all, when we set ng-controller equal to “BugController” in the <body> element, we didn’t even include parentheses, much less pass anything to the function. This is an example of “dependency injection”. Modern frameworks like AngularJS handle a lot of stuff behind the scenes. As long as we set things up like we did in this example, Angular is smart enough to know that we’re expecting it to create a new scope and “inject” it into the constructor. The $scope parameter is the clue that Angular uses to figure this out. We’ll be seeing a lot of other examples of dependency injection as we go along.

The first thing BugController does when it’s created is to create an array of objects called bugArray. Each object in the array represents one bug and includes information such as the common name, the class, the order, etc. The last member variable in each object is “Filename”. This is the name of the image file for the bug without the .jpg extension. All the image files for this book are jpegs and, at least for the early examples, they will be located in an images subdirectory of the directory where the example code is found.

A quick note on taxonomy: Scientists have developed a standard system of classification using the following divisions (the lower divisions being nested within the higher ones):

Kingdom

Phylum

Subphylum

Class

Order

Suborder

Family

Subfamily

Tribe

Genus

Species

All of the critters we (that is, me) are interested in here are members of the kingdom Animalia and the phylum Arthropoda (i.e. they are arthropods), so the first couple of divisions will be the same for all of them:

Kingdom Animalia (Animals)

Phylum Arthropoda (Arthropods)

For example, the complete classification for a certain type of fire ant is as follows:

Kingdom Animalia (Animals)

Phylum Arthropoda (Arthropods)

Subphylum Hexapoda (Hexapods)

Class Insecta (Insects)

Order Hymenoptera (Ants, Bees, Wasps and Sawflies)

Superfamily Formicoidea (Ants)

Family Formicidae (Ants)

Subfamily Myrmicinae

Tribe Solenopsidini

Genus *Solenopsis* (Fire Ants and Thief Ants)

Species *amblychila* (Solenopsis amblychila)

The most important of these is the Genus and species, which are often written as [Genus] [species] -- Solenopsis amblychila, in this case. The genus name is always capitalized and the species is always lowercase.

Our example program only lists the class through the species, leaving out relatively minor divisions like suborder, tribe, etc. In many cases, we will not know (or not bother specifying) the species name. In other cases, we may only classify a bug down to the level of family or even order.

Next, the program defines a function named “getBug” on the controller scope, using the syntax:

$scope.getBug = function(name) {

for (i in $scope.bugArray){

if (name == $scope.bugArray[i].CommonName)

return $scope.bugArray[i];

}

};

getBug() searches for the object in bugArray whose value for CommonName matches the value of the “name” parameter and returns it.

Just below getBug(), we define another function called setBug():

$scope.setBug = function() {

$scope.selectedBug = $scope.getBug($scope.selectedName);

};

This function is assigned to the ng-change attribute of the <select> element:

<select ng-model="bug" ng-change="setBug()">

When the user selects a bug from the drop-down, “selectedName” is set to the value of the selected item, which is the common name of the bug. The selected item for the drop-down has changed, so setBug() gets executed. setBug() calls getBug() and passes it the value of “selectedName”. getBug() finds the object for the selected bug and passes it back to setBug(), which assigns it to the “selectedBug” variable.

Now, the new lines of HTML can get the data for the bug from the “selectedBug” object, using the double curly brackets. For example:

Class: {{ selectedBug.Class}} <br/>

Finally, we’ve add a function called getFilename() to the controller :

$scope.getFilename = function(filename) {

if (filename) {

return "images/" + filename + ".jpg";

} else {

return '';

}

}

This function just gets the complete path to an image file when passed the filename. In our HTML code, we’ve modified the value of the ng-src attribute to call this function:

<img ng-src="{{ getFilename(selectedBug.Filename) }}"></img>

REFINING OUR CODE

You’ll find the source code for this next section in the Chapter 1\Example 3 folder.

In our last example, bugArray was defined in the controller. There are two problems with this. First, it violates MVC standards. Data belongs in the model, not the controller. Second, it really adds to the length of the index.html file. Eventually, we’ll learn how to store our data in a database, but, for now, we’ll just put it in a separate file. This file is named bugList.js and will be in the same directory as index.html. All bugList.js does is define an array called “bugs”:

var bugs = [

{

CommonName: "Tarantula Hawk Wasp",

Class: "Insecta",

Order: "Hymenoptera",

Family: "Pompilidae",

Genus: "Pepsis",

Species: "",

Description: "The tarantula hawk is an example of a parasitoid wasp. It will sting a taratula to paralyse it, then drag the spider to a brood chamber, where it lays a single egg on the spider's abdomen. When the egg hatches, the wasp larva begins eating the spider -- slowly. It actually eats the non-vital organs first to keep it alive for as long as possible! ",

Filename: "Tarantula Hawk"

},

{

CommonName: "Crayfish",

Class: "Malacostraca",

Order: "Decapoda",

. . .

. . .

. . .

},

{

CommonName: "Conura Wasp",

Class: "Insecta",

Order: "Hymenoptera",

Family: "Chalcididae",

Genus: "Conura",

Species: "",

Description: "This is a very tiny wasp -- about a millimeter long. It's an example of a hyperparasitoid. A parasitoid is a parasite that kills it's host. Many parasitoid wasps will inject an egg into the body of a host, which is often a larval insect. After the egg hatches, the wasp larva begins feeding on the host. The conura wasp, however, is not just a parasitoid -- it's a hyperparasitoid. It injects its egg into the larva of another type of parasitoid wasp!",

Filename: "conura wasp"

}

];

In index.html, I’ve added a line that loads the contents of bugList.js:

<script type="text/javascript" src="bugList.js"></script>

Now, let’s take a moment to refine our code a bit. First, we’ll provide a value for the ng-app directive:

<html ng-app="bugApp">

Next, we’ll change the contents of the <script> element at the end of the file as follows:

<script>

var bugAppModule = angular.module('bugApp', []);

bugAppModule.controller('BugController', function ($scope) {

$scope.bugArray = bugs; // from bugList.js

// Search bugArray for the bug with CommonName equal to name

$scope.getBug = function(name) {

for (i in $scope.bugArray){

if (name == $scope.bugArray[i].CommonName)

return $scope.bugArray[i];

}

};

$scope.setBug = function() {

$scope.selectedBug = $scope.getBug($scope.selectedName);

};

$scope.getFilename = function(filename) {

if (filename) {

return "images/" + filename + ".jpg";

} else {

return '';

}

}

});

</script>

The first thing we’ve done here is to use the AngularJS module() method to create a module for our Angular app. An Angular module is an object that acts as a container for the components of our app. When it’s being used to create a module, the module() method takes two arguments. The first is the name that will be used with ng-app to reference the module. The second is an array of dependencies. The dependencies are other Angular modules defined in other files which contain code that our app needs to work. (We’ll see how to define such modules later.) In this case, our app doesn’t have any dependencies, so we pass in an empty array.

Now why, you might ask, do we need to pass in a second argument at all if there are no dependencies? Isn’t Angular smart enough to just figure out that no second argument means no dependencies? The reason is that passing in an array (even an empty one!) tells Angular that we are *creating* a module. If we leave out the second argument entirely, Angular will think that we’re trying to load an already existing module. (That’s another thing we’ll get to later.)

The module() method returns the module object that it creates. We store that object reference in the variable “bugAppModule”, so that we can use it to call the module’s controller() method in the next line. The controller() method defines a controller constructor function and adds it to the module. The method takes two arguments – the name for the controller (“BugController”) and the definition for the contructor function. Pay careful attention to the line right before the end tag for the script:

});

</script>

The curly bracket closes the function definition and the parenthesis is the closing parenthesis for the controller() method.

The first line in our new controller function sets the bugArray variable equal to the “bugs” array that we loaded from bugList.js:

$scope.bugArray = bugs; // from bugList.js

The rest of the function is unchanged.

This completes the example. If you open index.html in your browser, it should work just like it did in the last example.

FURTHER REFINEMENTS

As you probably know, in JavaScript we can chain together method calls. So, instead of saving a reference to the module object in a variable and then using this variable to run the controller() method, we can get rid of the variable altogether and just do this:

<script>

angular.module('bugApp', [])

.controller('BugController', function ($scope) {

$scope.bugArray = bugs; // from bugList.js

. . .

. . .

. . .

};

});

</script>

It doesn’t really matter which way you do it. I like the second method just because it gets rid of the extra variable.

Chapter1, Example 4 contains this refinement.

ADDING BUGS (the good kind)

Let’s say we want to give the user the ability to define a new type of bug and add it to the drop-down list. We’ll need a new form for entering in the data for the bug and for specifying the image file that will be displayed when the bug is selected. To keep things simple, we’ll put all the HTML for the new page in index.html and use the ng-show attribute to determine which of the two pages (the Pick A Bug page or the Add A Bug page) is displayed at any given time.

We’ll do this in two steps. The first step is in Chapter 1, Example 5:

<!DOCTYPE html>

<html ng-app="bugApp">

<head>

<meta charset="utf-8" />

<title>Buggy Example 5</title>

<script src="http://code.angularjs.org/1.2.14/angular.js"></script>

<script type="text/javascript" src="bugList.js"></script>

</head>

<body ng-controller="BugController">

<div>

<input type="button" ng-click="goToLookUp()" value="Look up a bug">&nbsp;&nbsp;&nbsp;&nbsp;

<input type="button" ng-click="goToAdd()" value="Add a bug">

</div>

<div ng-show="currentPage == 'Lookup'">

<h1>Pick a bug. Any bug...</h1>

<select ng-model="selectedName" ng-change="setBug()">

<option></option>

<option>Giant Water Bug</option>

<option>Wind Scorpion</option>

<option>Ant</option>

<option>Scorpion (under black light)</option>

</select>

<div ng-show="selectedName">

<h3>You selected: {{ selectedBug.CommonName }}</h3>

Class: {{ selectedBug.Class }} <br/>

Order: {{ selectedBug.Order }} <br/>

Family: {{ selectedBug.Family }} <br/>

Genus: {{ selectedBug.Genus }} <br/>

Species: {{ selectedBug.Species }} <br/><br/>

{{ selectedBug.Description }} <br/>

<img ng-src="{{ getFilename(selectedBug.Filename) }}" />

</div>

</div>

<div ng-show="currentPage == 'AddBug'">

<h1>Add A Bug</h1>

<label>Common Name:</label>

<input type="text" placeholder="Please enter the common name" ng-model="commonName2"><br/>

<label>Class:</label>

<input type="text" placeholder="Please enter the class" ng-model="className2"><br/>

<label>Order:</label>

<input type="text" placeholder="Please enter the order" ng-model="orderName2"><br/>

<label>Family:</label>

<input type="text" placeholder="Please enter the family" ng-model="familyName2"><br/>

<label>Genus:</label>

<input type="text" placeholder="Please enter the genus" ng-model="genusName2"><br/>

<label>Species:</label>

<input type="text" placeholder="Please enter the species" ng-model="speciesName2"><br/>

<label>Description:</label>

<input type="text" placeholder="Please enter a description" ng-model="description2"><br/><br/>

<input type="button" ng-click="addTheBug()" value="Add it to the list">

</div>

<script>

angular.module('bugApp', [])

.controller('BugController', function ($scope) {

$scope.currentPage = "Lookup";

$scope.bugArray = bugs; // from bugList.js

$scope.goToLookUp = function() {

$scope.currentPage = "Lookup";

};

$scope.goToCatalog = function() {

$scope.currentPage = "Catalog";

}

$scope.goToAdd = function() {

// Make sure all these variables are initialized to the empty string

$scope.commonName2 = "";

$scope.className2 = "";

$scope.orderName2 = "";

$scope.familyName2 = "";

$scope.genusName2 = "";

$scope.speciesName2 = "";

$scope.description2 = "";

$scope.filename2 = "";

// Display the Add A Bug page

$scope.currentPage = "AddBug";

};

$scope.addTheBug = function() {

$scope.bugArray.push(

{

CommonName: $scope.commonName2,

Class: $scope.className2,

Order: $scope.orderName2,

Family: $scope.familyName2,

Genus: $scope.genusName2,

Species: $scope.speciesName2,

Description: $scope.description2,

Filename: $scope.filename2

}

);

console.log($scope.bugArray);

};

// Search bugArray for the bug with CommonName equal to name

$scope.getBug = function(name) {

for (i in $scope.bugArray){

if (name == $scope.bugArray[i].CommonName)

return $scope.bugArray[i];

}

};

$scope.setBug = function() {

$scope.selectedBug = $scope.getBug($scope.selectedName);

};

$scope.getFilename = function(filename) {

if (filename) {

return "images/" + filename + ".jpg";

} else {

return '';

}

}

});

</script>

</body>

</html>

The first thing we’ve done is to add two buttons at the top of the website:

<div>

<input type="button" ng-click="goToLookUp()" value="Look up a bug">&nbsp;&nbsp;&nbsp;&nbsp;

<input type="button" ng-click="goToAdd()" value="Add a bug">

</div>

Each button has an ng-click attribute set to a function call. As you can probably guess, ng-click works like the onclick HTML attribute. The two functions – goToLookUp() and goToAdd() will be defined in the controller.

Next, we’ve wrapped all the HTML for the Pick A Bug page inside a <div> element, as follows:

<div ng-show="currentPage == 'Lookup'">

<h1>Pick a bug. Any bug...</h1>

<select ng-model="selectedName" ng-change="setBug()">

<option></option>

. . .

. . .

. . .

<img ng-src="{{ getFilename(selectedBug.Filename) }}" />

</div>

</div>

We’ve also added an ng-show attribute to the <div> element and set it equal to “currentPage==’LookUp’”. So, whenever the controller sets the variable currentPage to “Lookup”, the display page will become visible. Otherwise, it will be hidden.

Next, we’ve added the HTML code for our Add A Bug page. This code is likewise wrapped in a <div> element, this time with ng-show set to “currentPage == 'AddBug'":

<div ng-show="currentPage == 'AddBug'">

<h1>Add A Bug</h1>

<label>Common Name:</label>

<input type="text" placeholder="Please enter the common name" ng-model="commonName2"><br/>

<label>Class:</label>

<input type="text" placeholder="Please enter the class" ng-model="className2"><br/>

<label>Order:</label>

<input type="text" placeholder="Please enter the order" ng-model="orderName2"><br/>

<label>Family:</label>

<input type="text" placeholder="Please enter the family" ng-model="familyName2"><br/>

<label>Genus:</label>

<input type="text" placeholder="Please enter the genus" ng-model="genusName2"><br/>

<label>Species:</label>

<input type="text" placeholder="Please enter the species" ng-model="speciesName2"><br/>

<label>Description:</label>

<input type="text" placeholder="Please enter a description" ng-model="description2"><br/><br/>

<label>Filename:</label>

<input type="text" placeholder="Please enter a filename for the image" ng-model="filename2"><br/><br/>

<input type="button" ng-click="addTheBug()" value="Add it to the list">

</div>

You should be able to see what is going on here. If currentPage==’Lookup’ is true, then currentPage == 'AddBug' will be false and vice versa, so only one page will be displayed at a time. The contents of the Add A Bug page are very simple – mostly a bunch of textboxes, each with an ng-model attribute set to a new variable. At the very bottom of the page, there’s a button control with an ng-click attribute set to “addTheBug()”. (I haven’t bothered trying to make the page look pretty -- I want to keep the code as simple as possible for now.)

Now let’s look at the changes to the controller. A new line has been added at the very beginning of the constructor function:

$scope.currentPage = "Lookup";

This line creates the currentPage variable on the controller scope and sets it equal to “Lookup”. So, when the website loads and the constructor function executes, the Pick A Bug page will be displayed.

Further down in the constructor, we define the goToLookUp() function:

$scope.goToLookUp = function() {

$scope.currentPage = "Lookup";

};

This could hardly be more simple. When the user clicks on the “Look up a bug” button at the top of the website, goToLookUp() is called, which sets currentPage to “Lookup”. The two ng-show attributes discussed earlier do the rest.

The function definition for goToAdd() is very similar – it sets currentPage to “AddBug” in the last line – but it also sets all the variables associated with the new textboxes equal to the empty string. This means that all the textboxes on the Add A Bug page will be cleared every time the user clicks “Add a bug” to display the page.

$scope.goToAdd = function() {

// Make sure all these variables are initialized to the empty string

$scope.commonName2 = "";

$scope.className2 = "";

$scope.orderName2 = "";

$scope.familyName2 = "";

scope.genusName2 = "";

scope.speciesName2 = "";

$scope.description2 = "";

$scope.filename2 = "";

// Display the Add A Bug page

$scope.currentPage = "AddBug";

};

Finally, we define the addTheBug() function, which will be called when the user clicks the “Add it to the list" button. This function uses the data that the user entered into the textboxes to create a new bug object and then adds the object to bugArray. The last line of the function writes the entire bugArray object to the console, so that you can see that the new bug has been added. (If you’re using the Google Chrome web browser, you can display the console by holding down the function key and pressing F12. If you’re using a different browser, this may still work. Otherwise, google how to display the console for your particular browser.)

$scope.addTheBug = function() {

scope.bugArray.push(

{

CommonName: $scope.commonName2,

Class: $scope.className2,

Order: $scope.orderName2,

Family: $scope.familyName2,

Genus: $scope.genusName2,

Species: $scope.speciesName2,

Description: $scope.description2,

Filename: $scope.filename2

}

);

console.log($scope.bugArray);

};

In our next example, we’ll see how to automatically populate the drop-down menu from our bug array.

POPULATING THE DROP-DOWN

The code for this step is in Chapter 1, Example 6.

Up to now, we’ve been hard-coding the options for the drop-down list. Since the list of bugs is going to change, we need to come with some way of populating this drop-down list programmatically. We can do this using the ng-options directive. Here’s our new <select> element:

<select ng-model="selectedBug"

ng-options="bug as bug.CommonName for bug in bugArray">

<option value=""></option>

</select>

We’ve kept the first <option> element, which adds an empty item to the drop-down, but we’ve gotten rid of the other <option> elements entirely. Instead, we’ve added an ng-options directive to the <select> element:

ng-options="bug as bug.CommonName for bug in bugArray"

This tells Angular to add one <option> element for each object in bugArray. We’ve arbitrarily assigned each object to a variable named “bug”. Angular will create and inject a new scope for each <option> element, which can be accessed through that variable. The “bug as bug.CommonName” part says that we want to use the common name as the text for the <option> element, but we want to set the value of the option equal to the object itself.

We’ve made a couple of other changes to the <select> element. ng-model is now set to “selectedBug”, instead of “selectedName”. When the user selects one of the options, the associated bug object will be assigned to the “selectedBug” variable. The second change is that we’ve gotten rid of the ng-change directive, which was set to “setBug()”. This is no longer necessary, because simply selecting one of the options from the drop-down now results in the associated bug object being assigned to “selectedBug”, which is what setBug() was doing for us in the old version. We can also delete the definition for setBug() from our controller. We won’t be needing it anymore.

Take a look at the HTML code for the Add A Bug page. Just after the textbox for the description, there’s a label and textbox for the filename of the image:

<label>Filename:</label>

<input type="text" placeholder="Please enter a filename for the image" ng-model="filename2"><br/><br/>

When you enter the data for a new bug on this form, you should enter the filename of the associated image file into this textbox. Don’t enter the extension or the path. The program adds that part automatically. Naturally, it would be better if the user could browse for the image file that she wanted, instead of having to type in the filename. We’ll make that change later, but this will be good enough for now.

At this point, everything should be working. When the user fills in the text boxes on the Add A Bug page and clicks the "Add it to the list" button, a new object will be created and added to the bug array. If the user then clicks "Look up a bug" to display the Pick A Bug page, she will be able to select the new bug from the drop-down menu and it should be displayed correctly.

Note that we haven’t included any data validation here. If you click “Add it to the list” without entering any data, an empty object will be added to the array. There’s also nothing to keep you from adding the same bug multiple times. Data validation is important and we will be adding it later, but, for now, I just want you to understand how we can use the various directives to do what we need to do.

There’s just one more little detail. When the user clicks the "Add it to the list" button, the only thing that lets him know that the bug has been successfully added is the line that’s displayed in the console. Obviously, this will not do. I’ve replaced the console.log() line with the following:

$window.alert("You added a bug!");

This, of course, just displays an alert dialog box telling the user that the bug has been added. But why do we need to prefix the alert command with $window? In JavaScript, alert() is a method in the global window object. Unfortunately, the AngularJS root scope does not give us access to this object. Instead, we have to pass in a built-in Angular service called $window to our controller. This service contains an alert() method we can use.

If you look at the beginning of the script file, where the controller is added to the Angular module, you’ll see where we’re passing in the service:

<script>

angular.module('bugApp', [])

.controller('BugController', function ($scope, $window) {

. . .

Now we’re free to user alert() to notify the user that the new bug was successfully added.

TESTING IT OUT

Let’s test out our new code. Open the index.html file in your web browser and click “Add A Bug” to display the Add A Bug page. Type the following information into the text boxes and then click the "Add it to the list" button. Make sure you type in the filename exactly as shown – flower crab spider. There should already be a file with this name in the images folder.

Common Name: Flower Crab Spider

Class: Arachnida

Order: Araneae

Family: Thomisidae

Genus: Misumena

Species:

Description: The color of these spiders often matches the color of the flowers that they hang out on. In fact, the goldenrod crab spider can slowly change its color from white to yellow or vice versa over a period of a few days.

Filename: flower crab spider

Now if you go to the Pick A Bug page and click on the drop-down, you should see that Flower Crab Spider is one of the selections. Click on it and the data and image for it should be displayed.

Of course, if you refresh the webpage, the bug array will revert to its hard-coded form – the change you made will be lost. Eventually, we’ll have to add some code to save the data in the array to a database, but that’s yet another thing we’ll get to later.

ADDING A CATALOG

Chapter 1, Example 7

The drop-down control works great if you only have a few bugs in your list, but eventually we’ll want our program to be able to handle hundreds of bugs. For that, we’ll need to have some sort of catalog. Let’s begin by adding a new page to our website:

<div ng-show="currentPage == 'Catalog'">

<br/>

<div ng-repeat="bug in bugArray">

{{ bug }}

</div>

</div>

As usual, we use ng-show to ensure that only one page gets displayed at a time:

ng-show="currentPage == 'Catalog'"

On the Catalog page, we want to display the common name for each bug in bugArray, along with a thumbnail of the associated image. We would also like each common name and thumbnail to act as a link to a page that displays the data for the bug and the full image. So, the user will be able to scroll thru the thumbnails and click on the bug that he’s interest in.

First, let’s just figure out how to step through bugArray, displaying the data for each object in the array. To do this, we use the built-in ng-repeat directive. ng-repeat is pretty simple in its basic form. You add it as an attribute to an element that you want repeated and set it equal to “[item] in [collection]”. This tells AngularJS to create one element for each item in the collection. In our case, we want to create a <div> element for each bug in bugArray, so we use:

<div ng-repeat="bug in bugArray">

. . .

</div>

If there are four bugs in bugArray, then Angular will create four <div> elements. Angular will also create a new scope for each element and inject it into the element. The “bug” variable is how we access that scope. For now, all we’re going to put inside the repeated <div> element is {{ bug }}, which just displays the contents of the bug object.

Eventually, we’re going to replace {{ bug }} with a custom element that displays the common name and thumbnail as links, but that gets a bit involved, so I decided to save it for the next example. In the meantime, let’s go ahead and add a page to display the data for the bug and the full image, so it’ll be ready when we finish creating our new element. Here’s the HTML code for the Bug Viewer page:

<div ng-show="currentPage == 'BugViewer'">

<h3>You selected: {{ selectedBug.CommonName }}</h3>

Class: {{ selectedBug.Class }} <br/>

Order: {{ selectedBug.Order }} <br/>

Family: {{ selectedBug.Family }} <br/>

Genus: {{ selectedBug.Genus }} <br/>

Species: {{ selectedBug.Species }} <br/><br/>

{{ selectedBug.Description }} <br/>

<img ng-src="{{ getFilename(selectedBug.Filename) }}" />

</div>

There’s nothing new here. You should be able to see what I’ve done. Finally, we’ll add a function to the controller that will be used to display the bug when the user clicks on one of the links in the catalog:

$scope.displayBug = function(name) {

$scope.bug = $scope.getBug(name);

$scope.currentPage = "BugViewer";

}

The function calls getBug() to assign the “bug” variable to the bug that the user selected. The next line sets currentPage to “BugView”, which causes the Bug Viewer page to be displayed.

That’s as far as I’m going to take it in this example. You can open the webpage in a browser and check that when you click on the “Catalog” button, the new page is displayed, showing the contents of the bug array. You won’t be able to test the Bug Viewer page yet, because the program isn’t actually calling displayBug() anywhere. We’ll take care of this in Example 8.

ADDING A CUSTOM DIRECTIVE

Chapter 1, Example 8

IMPORTANT: For all the examples so far, you could just open the index.html file directly in a browser and everything would work as expected. For our next example, we’ll be creating a separate .html file whose contents will be loaded when index.html is displayed. If you open the new index.html directly in your browser and click on the “Catalog” button, you’ll get an error that says something about cross origin requests not being supported and the catalog page will not be displayed. (You’ll have to open the console to see the error message.) I’m not going to get into what a cross origin request is or why you get this error. You can google it, if you’re interested. The important thing is, it means that you have to use a web server to test the changes.

To do this, first you need to make sure that Microsoft’s web server, Internet Information Services (IIS), is enabled. If you don’t know how to tell or, if you need to enable it but don’t know how, just google “enable IIS”. It might also help to specify the version of Windows that you’re using (e.g. “enable ISS for Windows 8).

Once ISS is enabled, open Windows Explorer and go to C:\inetpub\wwwroot. Create a new subdirectory for the web page. Call it “mybugs” or whatever you want. Then copy the files from the Example8 directory into the new folder. Now, you should be able to navigate to the web page by entering the following url into the address bar of your web browser:

<http://localhost/mybugs/index.html>

Clicking the Catalog button should now display the list of thumbnails.

Now, let’s finally get into the nitty gritty of how to create a custom element. We do this by creating a custom directive. Open index.html in your text editor and scroll down to the end of the code for the controller. You’ll see that I’ve chained a call to the directive() method:

.directive('bugEntry', function() {

return {

restrict: 'E',

templateUrl: 'bug.html',

scope: {

bug: "=aBug",

displayBug: '&displayBug',

getFilename: '&getFilename'

},

link: function(scope, elem, attr) {

scope.displayBug(bug.Filename);

}

};

});

This is the code that creates the custom directive. The directive() method takes two arguments – a name for the custom element and a callback function that returns an object. There’s a lot going on here, but we’ll take it a step at a time.

The first member of the object returned by the callback function is “restrict” and it’s set to ‘E’:

restrict: 'E',

This tells AngularJS that we’re going to be using our new directive as an element. That is, we’ll be adding it like this:

<bug-entry [attributes…]></bug-entry>

Why does our custom element begin with “bug-entry”, instead of “bugEntry”? By convention, JavaScript variable names are written in camel case. As you probably know, this just means that each word in the variable has the first letter capitalized, except for the first one. For example:

var thisIsAnExampleVariable = 1;

The problem is that HTML is case insensitive. Angular’s solution is to go ahead and specify that the name for the custom element should be written in camel case when it’s referred to from within AngularJS, but the name should be converted to all lowercase words separated by dashes when used in the actual HTML. For example, “thisIsAnExampleVariable” becomes “this-is-an-example-variable”. Thus, when we use our new bugEntry element in our HTML code, we write its name as bug-entry. This convention applies to built-in AngularJS directives as well. Directives like ng-app and ng-repeat are referred to in the documentation as ngApp and ngRepeat.

The “restrict” attribute has three other possible settings:

restrict: ‘A’ – The directive will be used as an HTML attribute within an element. For example:

<div bug-entry><bug-entry>

or

<div bug-entry=”[some expression]”></bug-entry>

restrict: ’C’ – The directive will be used in a class attribute:

<div class="bug-entry"></div>

or

<div class=”bug-entry: [some expression]”></bug-entry>

restrict: ‘M’ – The directive will be used in an HTML comment:

<!-- directive: bug-entry expression -->

(I honestly don’t know what good that last one is, but I guess somebody thought it was important.)

The settings can be combined. For example:

restrict: ‘AE’

means that the directive can be used as an attribute or an element. If the restrict member is left out of the directive definition, it defaults to ‘A’.

Some older browsers don’t understand custom elements, so if you want to make sure that your HTML works for the absolute largest number of users, you should use the ‘A’ setting exclusively. Personally, I doubt that very many people are still using browsers that old, so I don’t worry about it, but it’s up to you. If you would like to try rewriting this example to use the directive as an attribute, I’ll let you google the details. I’ll probably be using the ‘E’ attribute exclusively in this book.

Notice that templateUrl is set to “bug.html”. This file contains the HTML code that will actually make up the new element. bug.html is in the same folder as index.html, so the controller should have no problem finding it. Here’s the contents of the file:

<br/>

<a href ng-click="displayBug( '{{bug.CommonName)}' )"><h2>{{ bug.CommonName }}</h2></a>

<a href ng-click="displayBug( '{{bug.CommonName)}' )"><img ng-src="{{ getFilename() }}" width="300px"></img></a>

This code does two things. It uses the bug object to display the common name of the bug – {{ bug.CommonName }} – and it displays a thumbnail for the bug:

<img ng-src="{{ getFilename() }}" width="300px"></img>

The common name and the thumbnail are each wrapped in an <a> link element. When Angular adds our new element to the webpage, it’s going to have to supply it with a new scope that includes the bug object and the two functions that the code in bug.html references – displaybug() and getFilename(). We tell it how to do this in the directive() call.

Take a look at the Catalog page in index.html:

<div ng-show="currentPage == 'Catalog'">

<br/>

<span ng-repeat="item in bugArray">

<bug-entry a-bug="item" display-bug="displayBug(item.CommonName)" get-filename="getFilename(item.Filename)"></bug-entry>

</span>

</div>

We’ve replaced {{ bug }} with the following line:

<bug-entry a-bug="item" display-bug="displayBug(item.CommonName)" get-filename="getFilename(item.Filename)"></bug-entry>

This is the custom element that we just defined. The element has an attribute name “a-bug” that is set to “item”. “item” refers to the item object in the ng-repeat directive. That is, it refers to a bug object in the array:

ng-repeat="item in bugArray"

Our <bug-entry> element is wrapped in <span ng-repeat="item in bugArray"> … </span>, so Angular will insert one copy of the element for each object in bugArray. For each copy, AngularJS will create a new scope and pass in the relevant bug object. That bug object is assigned to the a-bug attribute.

Now, look at the directive() call again. The object that is being returned by the callback function includes this member:

scope: {

bug: "=aBug",

displayBug: '&displayBug',

getFilename: '&getFilename'

}

If you don’t include a scope member in your directive definition, then the scope for the directive will be the controller scope (since it’s enclosed in the <body> element, which is assigned to the controller). This can cause some serious problems. For example, let’s say we define the following directive:

.directive(“test”, function() {

return {

restrict: ‘E’,

template: ‘<input type=”text” model=”test\_value /> Our test value: {{ test-value }}’

};

});

(Here, we’ve used template, instead of templateUrl, so that we can write the HTML for the template directly in the directive definition.)

Now, suppose we include the custom element twice in our index.html file:

[Some stuff…]

<test></test>

[Some more stuff…]

<test></test>

If we load the page, we will see the two input boxes, as expected, but if we type something into either of them, that something appears in the other box, as well. It also appears after both instances of “Our test value:” This is because both elements are using the same scope. Probably not what we want!

To prevent this, all we have to do is add the scope member to the directive definition, as follows:

.directive(“test”, function() {

return {

restrict: ‘E’,

scope: {},

template: ‘<input type=”text” model=”test\_value /> Our test value: {{ test-value }}’

};

});

This tells AngularJS to create an “isolate” scope for each instance of the custom element. Any value passed into the element, such as through the input box, will be passed only to that particular scope. Problem solved!

Since we’re passing values into our directive via attributes, we need some way to get these values into the isolate scope. That’s what the object assigned to the scope member does.

Look at the following line in that object:

bug: "=aBug",

“aBug” is the camel case version of “a-bug” – the first attribute in the <bug-entry> element. It’s being assigned to a variable named “bug”. Remember that our template file – bug.html – references a bug variable. This is where that variable is being created and assigned. The value of the a-bug attribute gets passed to the isolate scope, where it’s assigned to a local variable named “bug”. I know… it’s pretty confusing. Don’t worry, though – it gets worse.

The “=” symbol at the beginning of “=aBug” tells AngularJS that the binding between the new “bug” variable and the value of the a-bug attribute should be a two-way binding. That is, Angular can get the value of the attribute from the “bug” variable and it can also change the value of the attribute by changing the value of the “bug” variable. If Angular doesn’t need to change the value of the attribute, you can use “@”, instead of “=”. In fact, we could have probably used “@” here, but I used “=” when I wrote the code and I don’t feel like changing it.

If we were only going to use one attribute with our custom element, we could have defined the scope member like this:

scope: {

bug: "=”

}

You could then call the attribute anything you wanted when using the custom element. For instance:

<bug-entry whatever="bug”></bug-entry>

Angular would know to assign the value of the “whatever” attribute to the “bug” variable in the scope, because it’s the only attribute in the element. In our case, though, we’re using several attributes, so we have to specify a name for the attribute.

Now, let’s examine the line that passes the getFilename() function into the isolate scope:

getFilename: '&getFilename'

The ‘&’ symbol is used to tell Angular that we want to get a reference the getFilename() function in the controller. We assign it to a local variable, which also happens to be named “getFilename”. So, now, the Angular code in the <img> element in bug.html can use this local variable to call getFilename():

<img ng-src="{{ getFilename() }}" width="300px"></img>

We do the same to hook up displayBug() with our directive. This completes the directive definition. The ng-repeat directive causes the <span> element containing our <bug-entry> element to be repeated once for every bug in the bug array. At every repetition, one of the bug objects is passed into that instance of <bug-entry>, which uses it to display the common name and image as links. Clicking on one of the links displays the Bug View page and populates that page with the data and image for the associated bug.

BEST PRACTISES

Chapter 1, Example 9

Let’s end this chapter with a couple of recommended practices that may help us to avoid trouble in the future.

When we called our controller() method in the previous examples, we passed in a callback function with two parameters -- $scope and $window:

.controller('BugController', function ($scope, $window) {

As I explained earlier in the chapter, AngularJS figures out from our function definition that it needs to inject $scope and $window objects into the function. This process is called dependency injection. More specifically, it’s called *implicit* dependency injection, because Angular is using the parameter names to guess what dependencies it should inject. Where we run into trouble is if our code gets minified in the build process.

If you’re an experienced JavaScript programmer, then you’re probably familiar with the process of *minification*. This is a way of simultaneously shortening code while making it harder for other (competing) programmers to figure out what it’s doing. The problem is that, by default, the minification process changes the names of function parameters. So, if we leave our controller definition like it is now and we later decide to minify the project, then the $scope and $windows parameters will be renamed to something shorter and Angular will no longer be able to infer what dependencies we want injected. In other words, our code breaks.

Fortunately, there’s an easy solution. We just have to use *explicit* dependency injection. We replace our callback function with an array whose last element is the callback function and whose first two elements are the names of the dependencies that we want Angular to inject:

.controller('BugController', ['$scope', '$window', function ($scope, $window) {

We also have to modify the end of the controller call to reflect this change:

. . .

}])

.directive('bugEntry', function() {

. . .

What we’ve done here is tell AngularJS that we want it to pass in the ‘$scope’ object for the first function parameter and the ‘$window’ object for the second parameter. Now that we’ve explicitly specified the dependencies, we no longer have to use the names $scope and $window for the parameters. For instance, we could rewrite the above line as follows:

.controller('BugController', ['$scope', '$window', function (s, w) {

Angular will still pass in the ‘$scope’ object for the first parameter (s) and the ‘$window’ object for the second parameter (w). We don’t have to change the parameters, though, and it’s probably less confusing if we just keep using the old names.

Of course, we aren’t limited to just two dependencies – we can pass in as many as we want. If we needed to use the ‘$http’ object, for instance, we could rewrite the line like this:

.controller('BugController', ['$scope', '$window', $http, function ($scope, $window, $http) {

That takes care of that problem. The second potential problem I want to look at is something that probably would not affect our code as it stands now. It could affect it in the future, though, depending on how it evolves, so we may as well do something now to guard against that possibility.

You may remember that when we first defined the controller, we did it like this:

var bugAppModule = angular.module('bugApp', []);

bugAppModule.controller('BugController', function ($scope) {

$scope.bugArray = bugs; // from bugList.js

. . .

In this code, the variable “bugAppModule” is a global variable. What if our index.html file loads another module (one that could have been written either by us or by a third party) and that module just happens to also define a global variable named “bugAppModule”? Chances are, our code would break. Okay, it’s unlikely that another module would have a variable with a name like that, but if we keep defining our variables globally then, sooner or later, we’ll probably run into this situation.

The solution is to wrap our code in an *Immediately-Invoded Function Expression* (IIFE). An IIFE (pronounced iffy) is simply an anonymous function that is defined and called in the same line. For example:

(function() {

var x = “This IIFE was just defined and executed.”;

Console.log(x);

})();

This first set of parentheses defines the function and the last set tells JavaScript to go ahead and execute it. It’s as if you did this:

var test = function() {

var x = “This IIFE was just defined and executed.”;

Console.log(x);

};

test();

You’re just leaving out the step of assigning the function to a variable, since it’s not necessary. Since variables defined inside of a function are local to that function, the variable x is local, not global. So we could do the same thing with our old code:

(function() {

var bugAppModule = angular.module('bugApp', []);

bugAppModule.controller('BugController', function ($scope) {

$scope.bugArray = bugs; // from bugList.js

. . .

})();

And that solves the global variables problem.

One last, picky little detail… We’re going to add a semicolon in front of the IIFE:

;(function() {

var bugAppModule = angular.module('bugApp', []);

bugAppModule.controller('BugController', function ($scope) {

$scope.bugArray = bugs; // from bugList.js

. . .

})();

What does this do? As real-world program evolves, it’s possible that your code will get concatenated with other code. What if JavaScript misinterprets the first set of parentheses as input to a function because it immediately follows some other code. For example:

. . .someWord(function() {

var bugAppModule = angular.module('bugApp', []);

bugAppModule.controller('BugController', function ($scope) {

$scope.bugArray = bugs; // from bugList.js

. . .

})();

JavaScript could interpret “someWord” as a function whose argument is our IIFE. Adding a semicolon before the IIFE ensures that JavaScript will know that we’re beginning a new command.

So, getting back to Example 9, we finish it up by enclosing our code in an IIFE preceded by a semicolon:

;(function() {

angular.module('bugApp', [])

.controller('BugController', ['$scope', '$window', function ($scope, $window) {

. . .

}])

CHAPTER 2 – INTEGRATING WITH VISUAL STUDIO

The rest of this book is under construction. Likewise, any code you find in the Chapter 2 folder is also under construction and subject to change.