

Build a Credit Card Entry Page using Angular – Part 1

A common page on many public websites is a page that asks a user to submit their credit card information. This seemingly simple little page has quite a few moving pieces in it. This series of articles illustrates how to build the HTML, Web API calls, a view model class, the Entity Framework objects, and the appropriate AngularJS controller to create a credit card entry page. Yes, I am still using AngularJS (or Angular 1) as opposed to Angular 2. The reason for this is I am finding that many developers are more familiar with JavaScript than with TypeScript and wish to stay with a language they know. There is nothing wrong with Angular 1, and thus no compelling reason to upgrade to Angular 2 if you don't want to.

In the first part of this article series you build the basic HTML for the credit card entry page. You also load the credit card types, months and years into drop-down lists on the page using hard-coded data. Succeeding articles will show how to build a Web API, Entity Framework classes, and a view model to support getting and storing credit card data from a set of SQL Server tables.

Overview of SPA Architecture

In this first article, you are going to layout the overall SPA pages needed to support the credit card data entry page. Figure 1 shows the two pages you are going to create for this system. The index.html page is like a Master Page in Web Forms or a Shared Layout page in MVC in that it contains the “chrome” for all the pages you route to. The only page in this article you are going to route to is the creditcard.html page, but the **ng-view** directive can be used for as many pages as you need in your application.

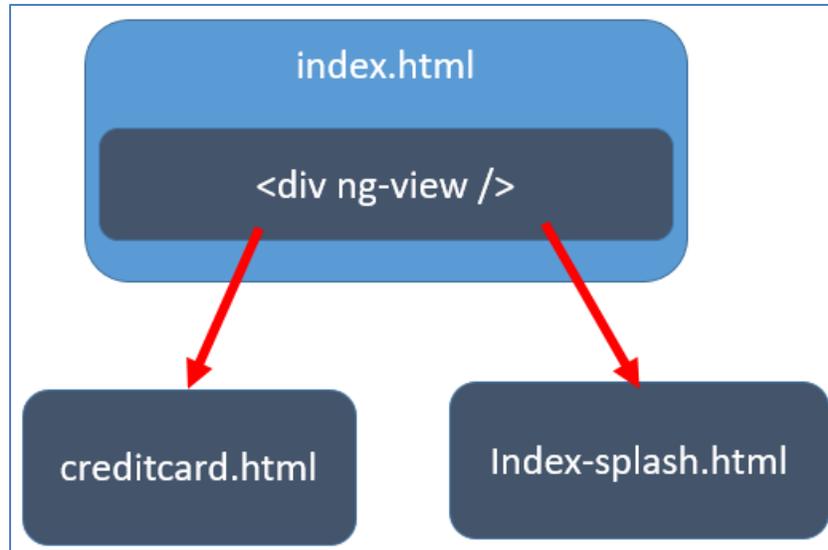


Figure 1: Overall architecture of SPA architecture

The creditcard.html page (shown in Figure 2) is where the user enters their credit card information. The index-splash.html page is displayed when the user first enters the system. If you have a ng-view directive, you must always have something displayed in it. If not, Angular will go into a recursive loop and eventually error out.

The image shows a web form titled "Credit Card Information". It contains several input fields and dropdown menus. At the top, there is a dropdown menu for "Select Credit Card Type" with "Visa" selected. Below that is a text input field for "Name on Card". The form is then split into two columns: "Credit Card Number" and "Security Code", each with a corresponding text input field. Below these are two more dropdown menus: "Exp. Month" (set to "12-December") and "Exp. Year" (set to "2016"). At the bottom, there is a text input field for "Billing Postal Code". The form concludes with two blue buttons: "Save" and "Cancel".

Figure 2: The Credit Card data entry page

Build the Main SPA Page

You are going to build a Single Page Application (SPA) to illustrate how to build the credit card page. This means you build a single `index.html` page from which you call the credit card page in your application. This sample will just contain the one credit card HTML page, but it is good to build your application using the SPA design pattern so you can add additional pages later.

Open Visual Studio 2015 and select **File | New | Project**. Choose Web under the Visual C# templates, then select **ASP.NET Web Application (.NET Framework)**. Set the **Name** of the project to **CreditCardEntry** and click the

OK button. Select the **Empty** template, and check the Web API check box before pressing the OK button as shown in Figure 3.

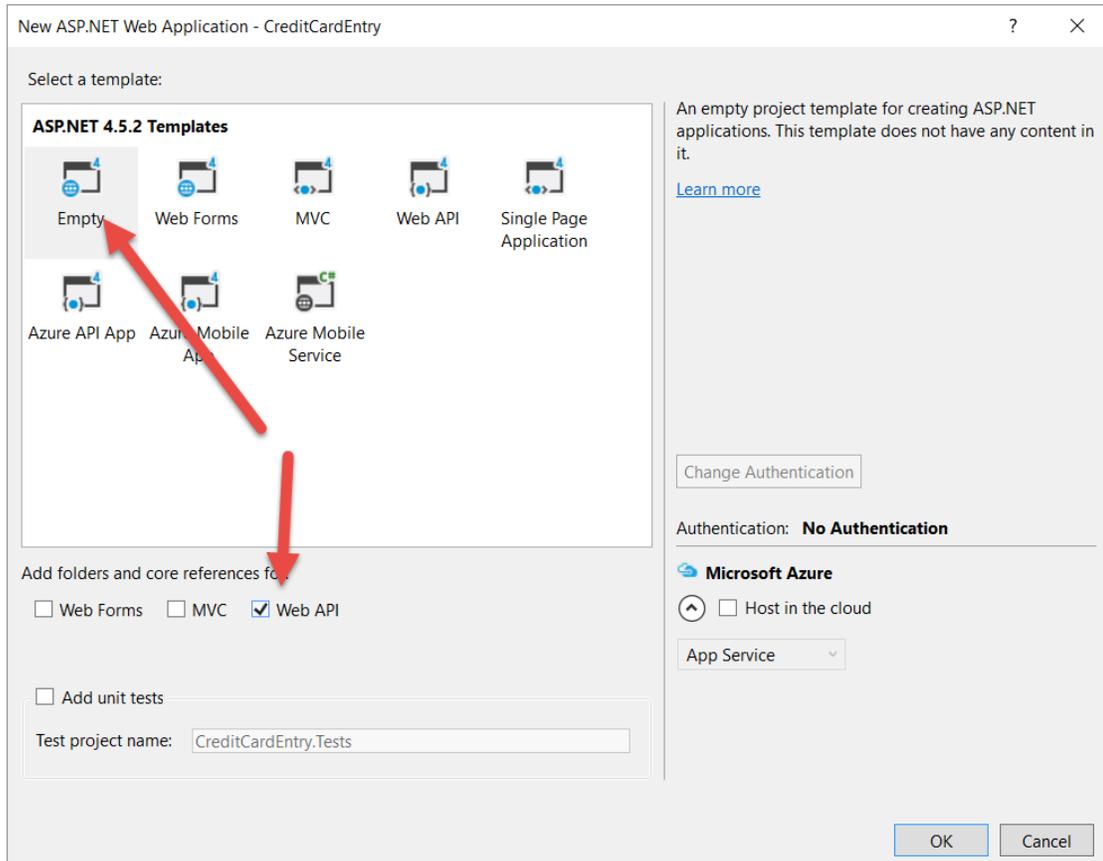


Figure 3: Select and Empty project template using the Web API

Using the NuGet Package Manager tool, install the packages needed for the complete sample. You are not going to use the Entity Framework yet, but in the next articles you will need this, so you might as well add it now.

- EntityFramework
- AngularJS.Core
- AngularJS.Route
- Bootstrap

Add a new HTML page in the root of your project called **index.html**. Modify the page to look like Figure 4 using the HTML shown in the listing below:

```

<!DOCTYPE html>
<html>
<head>
  <title>Credit Card Entry Sample</title>
  <meta charset="utf-8" />

  <link href="Content/bootstrap.min.css"
        rel="stylesheet" />
</head>
<body>
  <div class="container"
        ng-app="app"
        ng-controller="IndexController as vm">
    <div class="row">
      <div class="col-sm-10">
        <h1>Credit Card Entry System in Angular
        </h1>
      </div>
    </div>

    <div class="row">
      <div class="col-sm-10">
        <a href="#/creditcard"
           class="btn btn-primary">
          Credit Card Entry
        </a>
      </div>
    </div>

    <!-- Separator Line -->
    <div class="row">
      <div class="col-sm-10">
        &nbsp;
      </div>
    </div>

    <!-- ** BEGIN PARTIAL VIEWS AREA -->
    <div ng-view></div>
    <!-- ** END PARTIAL VIEWS AREA -->
  </div>

  <script src="scripts/angular.js"></script>
  <script src="scripts/angular-route.js"></script>
</body>
</html>

```

Towards the bottom of the page you see a `<div>` tag with an attribute of `ng-view`. This attribute tells Angular where you wish to insert partial pages within this page. The `#` sign as the first character in the `<a>` tag, followed by the `/creditcard` informs Angular you want to find an Angular route named "creditcard" and to load that partial page within the `<div>` tag with the `ng-view` attribute.

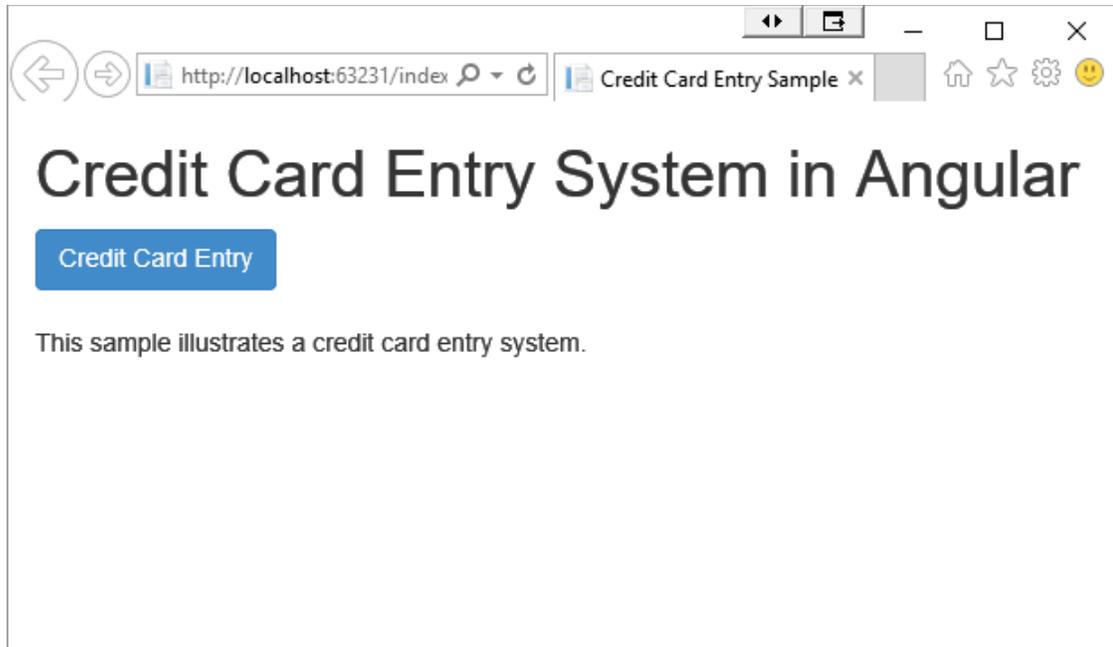


Figure 4: The main HTML page for your SPA

Modify the `Global.asax.cs` file to handle self-referencing in the Entity Framework and convert all pascal case property names to camel case. There is no self-referencing in this sample, but I find in many projects, this helps avoid some hard-to-track-down bugs.

```
protected void Application_Start() {
    GlobalConfiguration.Configure(WebApiConfig.Register);

    // Handle self-referencing in Entity Framework
    HttpConfiguration config =
        GlobalConfiguration.Configuration;
    config.Formatters.JsonFormatter
        .SerializerSettings.ReferenceLoopHandling =
            Newtonsoft.Json.ReferenceLoopHandling.Ignore;

    // Convert to camelCase
    var jsonFormatter = config.Formatters
        .OfType<JsonMediaTypeFormatter>().FirstOrDefault();
    jsonFormatter.SerializerSettings.ContractResolver =
        new CamelCasePropertyNamesContractResolver();
}
```

Build Angular Folder and JavaScript Files

A popular style for laying out Angular applications is to create a `\app` folder in your project and place all your HTML and JavaScript files in this folder. Group your files into folders with the name of the page you are working on. There are two Angular-controlled pages in this sample; the index page and the credit card page. Create a folder structure like the one shown in Figure 5.

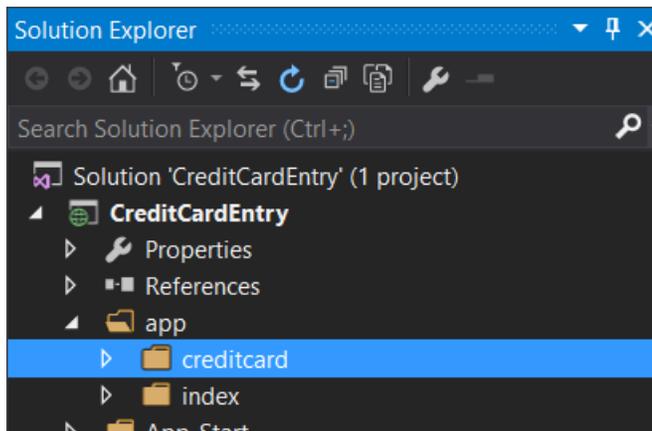


Figure 5: Create your folder structure.

Build Index Page JavaScript Files

In the `\app\index` folder add a new JavaScript file named `index.module.js`. This is where you create your Angular module that matches the name in the `ng-app` attribute of the `<div>` tag in your `index.html` page. As you are going to be using Angular routing include the `'ngRoute'` module dependency.

```
(function () {  
    'use strict';  
  
    angular.module('app', ['ngRoute']);  
})();
```

Create another JavaScript file named `index.controller.js` within the `\app\index` folder. In this file is where you define the Angular controller for this page. This page has no functionality other than to redirect to other pages, thus there is no code in this controller function.

```
(function () {
  'use strict';

  angular.module('app').controller('IndexController',
    IndexController);

  function IndexController() {
  }
}) ();
```

Add one more JavaScript file to the `\app\index` folder named **index.route.js**. This file defines the various routes that you use in anchor tags on your `index.html` page.

```
(function () {
  'use strict';

  // Create angular routing
  angular.module('app')
    .config(['$routeProvider', function ($routeProvider) {
      $routeProvider
        .when('/',
          {
            templateUrl: 'app/index/index-splash.html',
            controllerAs: 'vm',
            controller: 'IndexController'
          })
        .when('/creditcard',
          {
            templateUrl: 'app/creditcard/creditcard.html',
            controllerAs: 'vm',
            controller: 'CreditCardController'
          })
        .otherwise(
          {
            redirectTo: '/'
          });
    }]);
}) ();
```

Each route is defined using the **when()** function. You pass to the `when()` function the route defined in an anchor tag, or to call using Angular location services. You also pass in an object that defines either some inline HTML or an HTML template to display within the `ng-view` area. You may optionally define the controller name and a `controllerAs` property to name each controller reference.

The default route, defined as `when('/')`, tells Angular to redirect back to `index.html`. You must define either some HTML, or an HTML template to display within the `ng-view` area when someone requests a route or Angular gets caught in a loop which eventually causes an error. Personally, I like to

define a little “splash” page to display some opening remarks to the user. Create a new HTML page in the `\app\index` folder named **index-splash.html**. Strip all HTML from the page that is added, and just add this one line to the file.

```
<p>This sample illustrates a credit card entry system.</p>
```

Open the `index.html` page and add the following `<script>` tags below the angular script tags.

```
<script src="scripts/angular.js"></script>
<script src="scripts/angular-route.js"></script>

<script src="app/index/index.module.js"></script>
<script src="app/index/index.controller.js"></script>
<script src="app/index/index.route.js"></script>
</body>
</html>
```

You should be able to run the main index page. Don't click on the button as it is not hooked up yet.

Build the Credit Card HTML

Add an HTML page to the `\app\creditcard` folder named **creditcard.html**. This is a partial page, so we don't need any of the normal HTML tags, so go ahead and delete all HTML in this page. You only need to write the HTML to be loaded within the `<div>` tag with the `ng-view` attribute. The code shown below is not the final code, but is just the initial layout for the credit card HTML page.

```
<div class="row">
  <div class="col-sm-8">
    <form name="creditCardForm" novalidate>
      <div class="panel panel-default">
        <div class="panel-heading">
          <h3 class="panel-title">
            Credit Card Information
          </h3>
        </div>
        <div class="panel-body">

        </div>
        <div class="panel-footer">
          <div class="row">
            <div class="col-xs-12">
              <div class="pull-right">
                <button type="button"
                  class="btn btn-primary"
                  ng-click=
                    "vm.saveClick(creditCardForm)">
                  <i class="glyphicon
                    glyphicon-floppy-disk">
                  </i>
                  &nbsp; Save
                </button>
                <a class="btn btn-primary"
                  formnovalidate="formnovalidate"
                  href="#">
                  <i class="glyphicon
                    glyphicon-remove-circle">
                  </i>
                  &nbsp; Cancel
                </a>
              </div>
            </div>
          </div>
        </div>
      </div>
    </form>
  </div>
</div>
```

The credit card page is created within a Bootstrap row and a column. Add a `<form>` tag to encapsulate the input fields for the credit card data. Next, build the structure of the Bootstrap panel control into which all the messages and input fields will reside. The messages and input fields will be placed into the panel body. In the footer of the panel control you define a save and cancel button.

Add Credit Card Controller

For each partial web page, you need an Angular controller that goes with it. Add a JavaScript file in the `\app\creditcard` folder named **creditcard.controller.js**. Add the following code to this script file.

```
(function () {
  "use strict";

  angular.module("app")
    .controller("CreditCardController", CreditCardController);

  function CreditCardController($http, $location) {
    var vm = this;
    var dataService = $http;

    // Create UI state object
    vm.uiState = {
      isMessageAreaHidden: true,
      isLoading: true,
      messages: []
    };
  }
})();
```

In the above function the `$http` and `$location` services are injected into our controller. A common practice is to assign each of the services to a local variable. Assign the `$scope` from Angular to the variable **vm**, and the `$http` service to a variable named **dataService**.

The object in the credit card controller name **uiState** is used to hold properties that affect the user interface in some manner. The **isMessageAreaHidden** property turns on and off a Bootstrap alert area used to display messages to the user. The **isLoading** property determines whether to display a "Please wait while loading" message to the user when they first enter the web page or not. The **messages** array holds a collection of message objects that are displayed in the message area on the screen.

Error Message Area

Within the body of the panel control create a `<div>` tag to display error and validation messages to the user. This functionality for displaying messages will be added in a future article. Messages you add to the messages array you create in the controller are displayed within an unordered list. You can see the **ng-repeat** attribute is used to loop through each message in the array and display any values set in a **message** property within an `` element. Within the `<div class="panel-body">` add the HTML shown below.

```
<!-- ** BEGIN MESSAGE AREA ** -->
<div ng-hide="vm.uiState.isMessageAreaHidden ||
            (creditCardForm.$valid &&
             !creditCardForm.$pristine)"
      class="row">
  <div class="col-xs-12">
    <div class="alert alert-danger
              alert-dismissible"
          role="alert">
      <button type="button" class="close"
              data-dismiss="alert">
        <span aria-hidden="true">
          &times;
        </span>
        <span class="sr-only">Close</span>
      </button>
      <ul>
        <li ng-repeat="msg in vm.uiState.messages">
          {{msg.message}}
        </li>
      </ul>
    </div>
  </div>
</div>
<!-- ** END MESSAGE AREA ** -->
```

The `<div>` within the panel body uses the Angular **ng-hide** attribute to hide this area based on a few different flags. The `isMessageAreaHidden` property in the `vm.uiState` object in your controller is set by you depending on whether or not there are validation messages to be displayed. You check two other properties on the `creditCardForm` in addition to the `isMessageAreaHidden` property. If the `creditCardForm` is valid is true and `pristine` property is false, then the message area will be hidden.

Loading Message Area

Add another `<div>` tag just after the message area you created. This `<div>` tag uses the Angular **ng-show** attribute so this area is only displayed when a property in your controller, `vm.uiState.isLoading`, is set to a true value. When this page is first loaded, all the drop-down lists need to be loaded. Calling the Web API to get this data can take a couple of seconds on the first load, so it is a good idea to display a “Please wait while loading” message to the user before you display any other user interface items to them.

```
<!-- ** BEGIN LOADING MESSAGE AREA ** -->
<div class="row" ng-show="vm.uiState.isLoading">
  <div class="col-sm-offset-1 col-sm-10
    alert alert-warning">
    <div class="text-center">
      Please wait while loading
      credit card information...
    </div>
  </div>
</div>
<!-- ** END LOADING MESSAGE AREA ** -->
```

Input Fields

After the loading message area above, add another `<div>` element which is hidden until the `vm.uiState.isLoading` property is set to a false value. In the controller, this property is initially set to a true value. After all the drop-down lists have been loaded, this property is set to false. At that time the loading message will be hidden and the user input fields within this `<div>` tag will be displayed. Below is the HTML for each of the input fields required for the credit card data. You have not added the Angular binding yet, but will do that later in this article.

```
<!-- ** BEGIN CREDIT CARD ENTRY AREA ** -->
<div ng-hide="vm.uiState.isLoading">
  <div class="row">
    <div class="form-group col-sm-6">
      <label for="types">Select Credit Card Type
      </label>
      <select id="types"
        name="types"
        class="form-control">
      </select>
    </div>
  </div>
</div>
<div class="form-group">
  <label for="nameOnCard">Name on Card</label>
  <input id="nameOnCard"
    name="nameOnCard"
    class="form-control"
    placeholder="Name on Card"
    title="Name on Card"
    type="text" />
</div>
<div class="row">
  <div class="form-group col-sm-8">
    <label for="cardNumber">Credit Card Number
    </label>
    <input id="cardNumber"
      name="cardNumber"
      class="form-control"
      placeholder="Credit Card Number"
      title="Credit Card Number"
      type="text" />
  </div>
  <div class="form-group col-sm-4">
    <label for="securityCode">Security Code
    </label>
    <input id="securityCode"
      name="securityCode"
      class="form-control"
      placeholder="Security Code"
      title="Security Code"
      type="text" />
  </div>
</div>
<div class="row">
  <div class="form-group col-sm-8">
    <label for="expMonths">Exp. Month</label>
    <select id="expMonths"
      name="expMonths"
      class="form-control">
    </select>
  </div>
  <div class="form-group col-sm-4">
    <label for="expYears">Exp. Year</label>
    <select id="expYears"
      name="expYears"
      class="form-control">
  </div>
</div>
```

```

        </select>
    </div>
</div>
<div class="row">
    <div class="form-group col-sm-6">
        <label for="billingPostalCode">
            Billing Postal Code
        </label>
        <input id="billingPostalCode"
            name="billingPostalCode"
            class="form-control"
            placeholder="Billing Postal Code"
            title="Billing Postal Code"
            type="text" />
    </div>
</div>
</div>
</div>
<!-- ** END CREDIT CARD ENTRY AREA ** -->

```

Open the index.html page and add the following `<script>` tag below the other script tags you added earlier.

```

<script src="scripts/angular.js"></script>
<script src="scripts/angular-route.js"></script>

<script src="app/index/index.module.js"></script>
<script src="app/index/index.controller.js"></script>
<script src="app/index/index.route.js"></script>

<script src="app/creditcard/creditcard.controller.js">
</script>
</body>
</html>

```

You should be able to run the main index page and click on the Credit Card Entry button to display your credit card page. The page will show the "Loading" message because the `isLoading` property is set to true. To see the input fields, go into the `creditcard.controller.js` file, change the `isLoading` property to false, and rerun the page. If you do this, be sure to set it back to true once you stop the web application.

Credit Card Controller

Let's add more of the properties to the `CreditCardController` function that you are going to require for the credit card page. You need array properties to

load the drop-down lists. You need objects for selecting a credit card type and a month, and you need an object to hold properties for each input field. In addition, you need properties to turn on and off the various message areas on the page. Add the additional code shown below to your `creditcard.controller.js` file.

```
function CreditCardController($http, $location) {
  var vm = this;
  var dataService = $http;

  // Expose public properties
  vm.cardTypes = [];
  vm.months = [];
  vm.years = [];

  vm.selectedCardType = {};
  vm.selectedMonth = {};

  vm.creditCard = {
    creditCardId: null,
    cardType: null,
    nameOnCard: null,
    cardNumber: null,
    securityCode: null,
    expMonth: null,
    expYear: null,
    billingPostalCode: null
  };

  vm.uiState = {
    isMessageAreaHidden: true,
    isLoading: true,
    messages: []
  };

  // Initialize Controller
  loadCardTypes();
  loadYears();
  loadMonths();

  // Load Credit Card Types
  function loadCardTypes() {

  }

  // Load years
  function loadYears() {

  }

  // Load months
  function loadMonths() {

  }
}
```

The three arrays in the scope for this controller hold the data to go into the three drop-down lists on the credit card page. The **cardTypes** array and **month** array are both object arrays. The **years** array is an array of integer values representing a year the user can select from.

```
vm.cardTypes = [];  
vm.months = [];  
vm.years = [];
```

When the user selects a value from a drop-down that is bound to an object, you can bind to another object in your controllers' scope. Create two additional properties to bind to for the month and card type.

```
vm.selectedMonth = {};  
vm.selectedCardType = {};
```

The next object you create is one that holds the data for each input element on the screen. This object is named **creditCard** and is added to the scope just like the others. Within the **creditCard** object define a property to map to each input element.

```
vm.creditCard = {  
  creditCardId: null,  
  cardType: null,  
  nameOnCard: null,  
  cardNumber: null,  
  securityCode: null,  
  expMonth: null,  
  expYear: null,  
  billingPostalCode: null  
};
```

Load Mock Data for Drop-Down Lists

Looking at Figure 2 you know you need to load the three drop-down lists for credit card types, months and years. Let's finish building the routines you added to the **creditCard** controller to load the data into the properties of our controller. For this first sample you are going to hard-code the various values. In a later article you learn to connect to a back-end through a Web API to retrieve credit card type data from a SQL Server database table, and to load the months and years.

Load Credit Card Types

Open up the `\app\creditcard\creditcard.controller.js` file and locate the `loadCardTypes()` function. Modify the code to look like the code shown below:

```
function loadCardTypes() {
  vm.cardTypes.push({ cardType: 'Visa' });
  vm.cardTypes.push({ cardType: 'MasterCard' });
  vm.cardTypes.push({ cardType: 'American Express' });
  vm.cardTypes.push({ cardType: 'Discover' });

  vm.selectedCardType = vm.cardTypes[0];
}
```

In the above code push a new object with a single property onto the `cardTypes` array. The property is called `cardType` and you specify a value to display in the drop-down list on the HTML page. Feel free to add as many different card types as you wish to this list. Lastly, take the first `cardType` object and assign it to the `selectedCardType` property in your scope.

When you build the `<select>` HTML element you are going to use two Angular attributes; **ng-model** and **ng-options**. The **ng-model** attribute binds to the **vm.selectedCardType** property in the `CreditCardController`. The **ng-options** attribute specifies how to load the `<select>` element. Go to the `creditcard.html` page, locate the `types <select>` element and add these two attributes.

```
<select id="types"
  name="types"
  class="form-control"
  ng-model="vm.selectedCardType"
  ng-options="item.cardType for item in vm.cardTypes
    track by item.cardType">
</select>
```

You can think of the **ng-options** attribute like a **foreach** loop in C#. Here would be the equivalent pseudo-code in C# for what the `ng-options` attribute is doing.

```
foreach (item in vm.cardTypes) {
  <option value="item.cardType">item.cardType</option>
}
```

Let's break down each piece of the `ng-options` attribute value. The **"for item"** defines a local variable with the name of **"item"**. The **"in vm.cardTypes"** specifies the property in the scope of your controller to retrieve the collection of data from. The **"item.cardType"** before the **"for item"** is the name of the

property within each object in the `cardType` array you wish to display in the text portion of the drop-down. The “**track by item.cardType**” is the property from which to retrieve the value to put into the value portion of each `<option>` element within the `<select>`.

Load Years

I am seeing more and more credit card forms asking for a year up to 20 years in the future. So, for this sample, you load 20 years into the `vm.years` array. This array is simply an array of integer values. No object is necessary for each element in this array.

```
function loadYears() {
  var year = new Date().getFullYear();

  for (var i = 0; i < 20 ; i++) {
    vm.years.push((year + i));
  }

  vm.creditCard.expYear = year;
}
```

Go to the `creditcard.html` page, locate the `expYears` `<select>` element and add these two attributes.

```
<select id="expYears"
  name="expYears"
  class="form-control"
  ng-model="vm.creditCard.expYear"
  ng-options="item for item in vm.years track by item">
</select>
```

The **ng-model** attribute binds directly to the `vm.creditCard.expYear` property in the `CreditCardController`. Since each item in the `years` array is just an integer, that integer value is assigned to the `expYear` property.

The **ng-options** attribute uses just the variable name ‘item’ to bind to the text portion of the drop-down and to the value portion. Since each element is just an integer value, you do not specify any property name within ‘item’, you just use ‘item’ itself.

Load Months

The `loadMonths()` function is similar to the `loadCardTypes()` function in that you are creating an object to load into the `vm.months` array. The month object contains two properties; `monthNumber` and `monthName`. The `monthNumber`

property will be used in the value portion of the drop-down list, while the monthName property will be used in the text portion of the drop-down.

After loading the objects, set the expYear and expMonth of the vm.creditCard object to the next month, or January if the current month is 12, and to either the current year, or the next year if the current month is 12. Below is the code to load the months and set the expYear and expMonth.

```
function loadMonths() {
  var today = new Date();

  vm.months.push({ monthNumber: 1, monthName: 'January' });
  vm.months.push({ monthNumber: 2, monthName: 'February' });
  vm.months.push({ monthNumber: 3, monthName: 'March' });
  vm.months.push({ monthNumber: 4, monthName: 'April' });
  vm.months.push({ monthNumber: 5, monthName: 'May' });
  vm.months.push({ monthNumber: 6, monthName: 'June' });
  vm.months.push({ monthNumber: 7, monthName: 'July' });
  vm.months.push({ monthNumber: 8, monthName: 'August' });
  vm.months.push({ monthNumber: 9, monthName: 'September' });
  vm.months.push({ monthNumber: 10, monthName: 'October' });
  vm.months.push({ monthNumber: 11, monthName: 'November' });
  vm.months.push({ monthNumber: 12, monthName: 'December' });

  // Figure out which month to select
  // Make it next month by default
  vm.creditCard.expMonth = today.getMonth() + 2;
  // If past December, then make it January of the next year
  if (vm.creditCard.expMonth > 12) {
    vm.creditCard.expMonth = 1;
    vm.creditCard.expYear = vm.creditCard.expYear + 1;
  }
  vm.selectedMonth = vm.months[vm.creditCard.expMonth - 1];

  // Set the page UI flag as not loading anymore
  vm.uiState.isLoading = false;
}
```

Go to the creditcard.html page, locate the expMonths <select> element and add these two attributes.

```
<select id="expMonths"
  name="expMonths"
  class="form-control"
  ng-model="vm.selectedMonth"
  ng-options="item.monthName for item
    in vm.months
    track by item.monthNumber">
</select>
```

Just like for the card types, this <select> element is binding to the selectedMonth object in your controller.

The last thing the `loadMonths` function does is to set the `isLoading` property of the `vm.uiState` object to `false`. If you were calling the Web API in each of the above functions, these calls might take a little time. In the HTML page display a “Please wait while loading...” message to the user until the `isLoading` property is set to `false`. This is where that property is set to `false`, which makes the message disappear.

Bind Credit Card Input Fields

With your controller built and a `creditCard` object with properties to bind to the input fields, the only thing left to do is to perform the binding. Add a hidden input field directly below the `<form>` tag in your HTML. This hidden field is for the primary key value for the credit card record you add to a SQL Server database table. Initially it will be null, but after you add a record this value will be filled in and passed back from the Web API.

```
<input type="hidden"
      ng-model="vm.creditCard.creditCardId" />
```

Locate the `nameOnCard` input field and bind to the `creditCard` object by adding the following `ng-model` attribute.

```
<input id="nameOnCard"
      name="nameOnCard"
      ng-model="vm.creditCard.nameOnCard"
      class="form-control"
      placeholder="Name on Card"
      title="Name on Card"
      type="text" />
```

Locate the `cardNumber` input field and bind to the `creditCard` object by adding the following `ng-model` attribute.

```
<input id="cardNumber"
      name="cardNumber"
      ng-model="vm.creditCard.cardNumber"
      class="form-control"
      placeholder="Credit Card Number"
      title="Credit Card Number"
      type="text" />
```

Locate the `securityCode` input field and bind to the `creditCard` object by adding the following `ng-model` attribute.

```
<input id="securityCode"
       name="securityCode"
       ng-model="vm.creditCard.securityCode"
       class="form-control"
       placeholder="Security Code"
       title="Security Code"
       type="text" />
```

Locate the `billingPostalCode` input field and bind to the `creditCard` object by adding the following `ng-model` attribute.

```
<input id="billingPostalCode"
       name="billingPostalCode"
       ng-model="vm.creditCard.billingPostalCode"
       class="form-control"
       placeholder="Billing Postal Code"
       title="Billing Postal Code"
       type="text" />
```

At this point you can run the sample and you should see the credit card page displayed with the appropriate data in the drop-down lists.

Summary

In this article you created a new empty web application in Visual Studio, added Angular and Bootstrap to display a credit card page. You built a few different JavaScript files to build an Angular module and routes for the SPA. You then built a credit card controller which loads data into arrays for displaying in drop-down lists on the credit card page. You also setup the appropriate HTML to display error messages, a loading message, and the various input fields for accepting credit card information from a user. In the next article you will learn to get data from the Web API instead of using hard-coded data.

Sample Code

You can download the code for this sample at www.pdsa.com/downloads.
Choose the category "PDSA Articles", then locate the sample **Code Magazine: Angular Credit Card Page – Part 1**.