Chapter 3Build a Credit Card Entry Page using Angular – Part 3

In the last two articles, you built an HTML page (Figure 1) to enter credit card information. You have the drop-down lists loaded with data coming from a Web API service. Your last tasks for this page are to validate the data entered is correct, both on the client and the server, display any validation messages, and finally, save the credit card data into the CreditCard table in your SQL Server database.

You are going to build custom directives in Angular to validate the input on the client side. A new method is going to be added to the view model class to save the credit card data. An additional method is added to the EF classes to validate the data on the server-side. If validation errors are detected, those errors are serialized and sent back to the client for display in an unordered list. This article will show you how to accomplish each of these tasks.

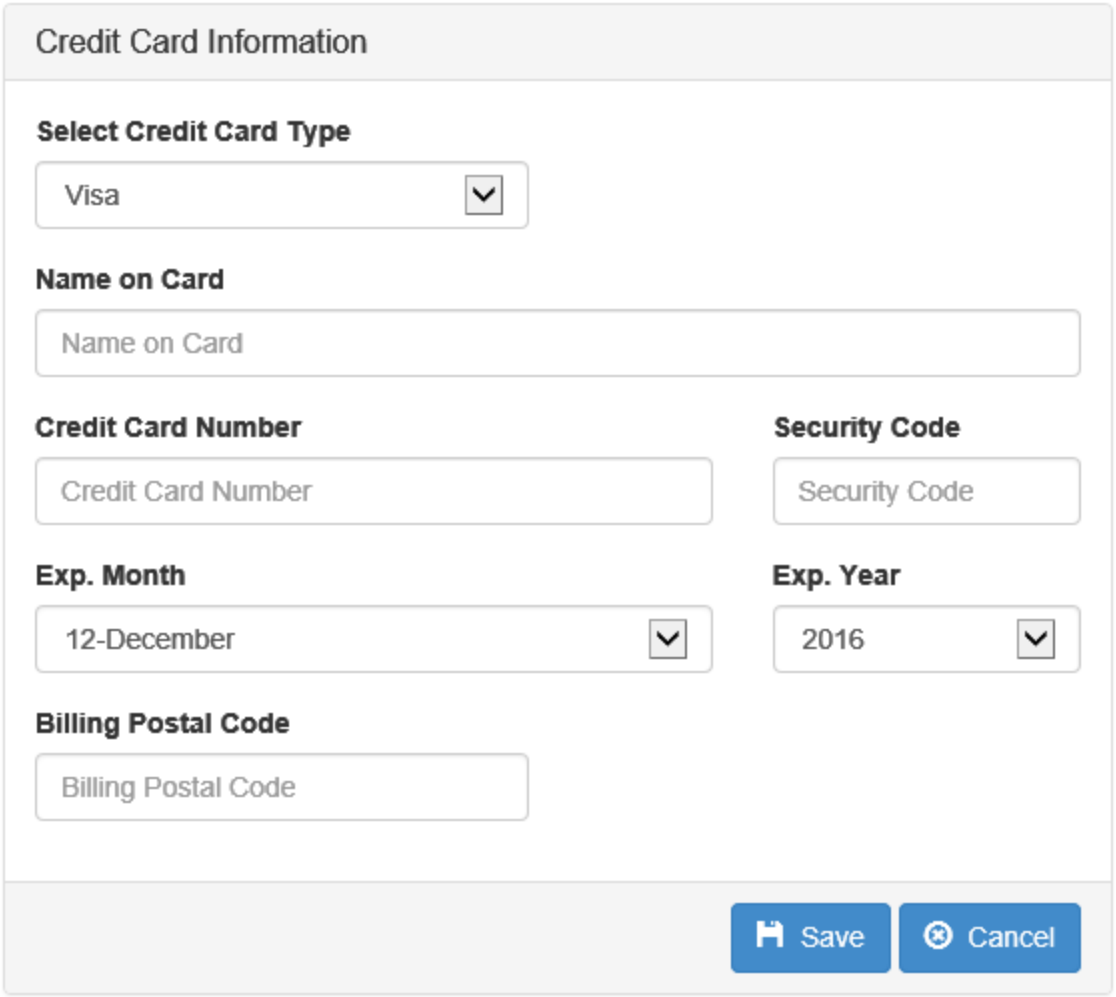

Figure 1: The Credit Card data entry page

Save Credit Card Data

The credit card table has a primary key field named CreditCardId. This field is a property in the CreditCard class generated by the Entity Framework. In the first article in this series, a hidden input field was created within the <form> tag to hold the CreditCardId field. If this value is null or an empty Guid, you know you need to insert the entered credit card data. If the value is a real Guid, then you know you just want to update the entered credit card data.

In the ViewModelLayer project, open the CreditCardViewModel class and add a SaveData() method. This class has an Entity property which is an instance of the CreditCard class. When the data is submitted by the user, the Entity

property of the view model class is filled in by the controller. If the CreditCardId property is filled in with a valid Guid, assume that the credit card data already exists in the database and set the state of the current entity in the EF data context to **modified**. When the SaveChanges method is called on the data context, an INSERT or an UPDATE statement is sent to the SQL Server.

```
public void SaveData() {
  PTC db = null;

  Messages = null;
  try {
    db = new PTC();

    // Determine whether to Insert or Update
    if (Entity.CreditCardId == null ||
        Entity.CreditCardId == Guid.Empty) {
      // Set new Guid for Primary Key
      Entity.CreditCardId = Guid.NewGuid();
      db.CreditCards.Add(Entity);
    }
    else {
      db.Entry(Entity).State =
         System.Data.Entity.EntityState.Modified;
    }
    db.SaveChanges();
  }
  catch (DbEntityValidationException ex) {
    IsValid = false;
    Messages = ex.EntityValidationErrors.ToList();
  }
}
```

Add CreditCardController

Go back to the CreditCardEntry project and locate the Controllers folder. Right mouse click on this folder and select **Add | Web API Controller Class (v2.1)**. If this option does not appear in your drop down menu, select Add | New Item and choose it under the Web | Web API templates. Set the name of this controller to **CreditCardController**. Delete all the code in this controller and add a Post method as shown below:

```
[HttpPost()]
public IHttpActionResult Post(
      CreditCard card) {
  IHttpActionResult ret = null;
  CreditCardViewModel vm =
    new CreditCardViewModel();
  ModelStateDictionary Messages =
    new ModelStateDictionary();

  // Assign client-side credit card object
  // to view model entity
  vm.Entity = card;

  // Attempt to save data
  vm.SaveData();
  if (vm.IsValid) {
    ret = Created<CreditCard>(
       Request.RequestUri +
         vm.Entity.CreditCardId.ToString(),
           vm.Entity);
  }
  else {
    if (vm.Messages.Count > 0) {
      // Validation errors
      foreach (var msgs in vm.Messages) {
        foreach (var item in msgs.ValidationErrors) {
          Messages.AddModelError(item.PropertyName,
             item.ErrorMessage);
        }
      }

      ret = BadRequest(Messages);
    }
  }

  return ret;
}
```

Add some using statements to the top of this controller file to resolve references to the various classes used in this method.

```
using DataLayer;
using ViewModelLayer;
using System.Web.Http.ModelBinding;
```

This Post method takes the CreditCard object passed from the UI and places it into the Entity property of the view model class. Next it calls the SaveData method you just wrote to either insert or update the data into the CreditCard table. If the view model's IsValid property is set to true, set the return value to the result of the Created method. This method returns the URI where the front end can retrieve the new entity. The Created method also returns the complete entity object back to the front end. You should use this method to

pass back any property data that may have changed because of the insert or update into the table.

If the IsValid property is set to a false value, then check to see if there are messages in the Messages property of the view model class. If there are you loop through the messages in the Messages property and add them as a model error in a ModelStateDictionary. This dictionary can be passed as a payload from the BadRequest method. In your Angular controller, loop through these model state dictionary objects and retrieve the error messages and display them on the UI to your user. Later in this article you add validation failure messages to this Messages property.

Save Data from Angular

Now that you have the back-end services written to save the credit card data to your database table, it is now time to write the Angular code to clean up the data on the front end and call the Post method in your CreditCard Web API controller.

Add Clean Up Data Function

Before I pass any data from the client-side to the server-side, I always have a little routine I call named cleanUpData. This function can clean up dates (which can be an issue in Internet Explorer), it can take selected values from drop-downs and move it into the actual entity object to pass to the server. It can also do any other clean up that you may find necessary. Open the creditcard.controller.js file and add a function named cleanUpData.

```
function cleanUpData() {
  // Get card type
  vm.creditCard.cardType =
      vm.selectedCardType.cardType;
  // Get expiration month
  vm.creditCard.expMonth =
      vm.selectedMonth.monthNumber;
}
```

In the cleanUpData function for the credit card page, take the selected card type from the vm.selectedCardType.cardType property and put it into the vm.creditCard.cardType property. Next, take the selected month number from the vm.selectedMonth.monthNumber property and put it into the vm.creditCard.expMonth property. You are going to pass the vm.creditCard

object to the server and this methods retrieves the data from the appropriate selected objects and puts that data into the appropriate properties in the vm.creditCard object.

Add Insert Data Function

Add a function named insertData to your creditcard.controller.js. Call the cleanUpData function you just created. Call the dataService.post function to call the Post method on your server. Pass to the post function the URL where your post method is located and the vm.creditCard object which contains the data your user entered on the credit card page. If the call to the Post method is successful, get the result.data and set it back into the vm.creditCard object. In case the server updates any of the fields during the insert or update statement, you want to be able to reflect that data back on the page for the user. However, in this function, the user won't see it as you are going to navigate back to the home page using $location service. However, if you were going to stay on this page for some reason, it would be good to display the updated values to the user.

```
function insertData() {
  // Clean up object before sending to server
  cleanUpData();

  // Post credit card info to server
  dataService.post(
      "/api/CreditCard", vm.creditCard)
    .then(function (result) {
      // Update credit card object
      vm.creditCard = result.data;

      // Redirect back to home page
      $location.path("/");

    }, function (error) {
      handleException(error);
    });
}
```

Add Save Click Function

The save button on your credit card HTML page has a ng-click event which calls a function named saveClick. You pass in the name of the <form> to this function to test the validity of validation you created on the form. You are going to learn about validation a little later in this article. For now, just write the saveClick as shown below.

```
function saveClick(creditCardForm) {
  if (creditCardForm.$valid) {
    vm.uiState.isMessageAreaHidden = true;
    creditCardForm.$setPristine();
    insertData();
  }
  else {
    vm.uiState.isMessageAreaHidden = false;
  }
}
```

If all the controls pass Angular validation the $valid property is set to true on the creditCardForm object. To make sure no further messages are displayed on the web page, call the $setPristine function. The $setPristine function sets the internal flags on the Angular credit card model to a value which is consistent with the first time you entered the page. Call the insertData function to post the data entered to the Web API controller.

All functions you need to call from the HTML page must be registered on the $scope object. Add a blank line just below your variable declarations and before the call to the loadCardTypes function and add the following lines of code.

```
// Expose public functions
vm.saveClick = saveClick;
```

Run the credit card page and enter some data, click on the save button, and you should be able to look in the CreditCard table and see the data you entered.

Add Angular Validation

For each field on your web page, decide which fields to perform validation upon. Determine the type of validation you can accomplish with the attributes available in HTML/HTML5 and Angular. In the table below is a list of the attributes you can use with Angular validation.

Attribute	Type	Description
required	HTML	The field must contain a value.
min	HTML	A minimum value for a numeric input field.
max	HTML	A maximum value for a numeric input field.

ng-minlength	Angular	The minimum number of characters for a field.
ng-maxlength	Angular	The maximum number of characters for a field.
ng-required	Angular	The field must contain a value. Same as 'required'.
ng-pattern	Angular	A regular expression the input value must match.

Table 1: Validation attributes you may add to any input field.

Each input field must have the **name** attribute in addition to the **id** attribute if you wish to use Angular validation. The name attribute, combined with one or more of the attributes listed in Table 1, is what Angular uses to determine the set of fields that need to be validated.

Add Attributes to Input Fields

Open the creditcard.html page and start adding validation to each of the input controls that require it. You are going to add a combination of HTML attributes, custom validation directives, and Angular validation. Start by locating the nameOnCard field and add the **pdsa-validatenotlowercase** and the **required** attributes. The custom directive **pdsa-validatenotlowercase** will check to make sure that the name on the credit card is not entered as all lower-case characters. You will write this custom directive later in this article, for now, just add the attribute.

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

A credit card number needs to be of a certain length, and you can validate the number entered by using the Luhn algorithm. The custom directive **pdsa-validatenotlower** case is simply going to check the length, but you can add the Luhn algorithm if you wish. Yes, we could use the ng-minlength and ng-maxlength attributes for verifying length, but I want to show you how to create a custom directive. Locate the cardNumber control and add the pdsa-validatecreditcard and required attributes.

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

A credit card security code is the 3 or 4-digit code found on the back of a credit card. Again, you are going to write a custom directive called pdsa-validatesecuritycode to check the length of the input. Find the securityCode control and add the **pdsa-validatesecuritycode** and **required** attributes.

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

The billing postal code for the credit card can be just about any combination of letters, spaces and numbers up to 18 characters. You can add another custom directive to test to ensure only letters, spaces and numbers and no other characters, or you can use the **ng-pattern** directive and a regular expresssion. Locate the billingPostalCode control and add the **ng-maxlength** and **required** attributes.

```
<input id="billingPostalCode"
       name="billingPostalCode"
       ng-model="vm.creditCard.billingPostalCode"
       ng-maxlength="18"
       required
       class="form-control"
       placeholder="Billing Postal Code"
       title="Billing Postal Code"
       type="text" />
```

Add Custom Directives

You added some custom attributes on the input fields above, so you now need to register the appropriate directives on your Angular application module. Directives are registered to the application module using the directive API. You may chain multiple directive functions together if you have more than one.

Add a new JavaScript file in the \creditcard folder named **creditcard.directives.js**. In this file, add the following code to define the overall structure for each of the custom directives you added to the input fields.

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

  angular.module("app")
   .directive('pdsaValidatenotlowercase', function () {
    return { };
  })
  .directive('pdsaValidatecreditcard', function () {
    return { };
  })
  .directive('pdsaValidatesecuritycode', function () {
    return { };
  });
})();
```

Each directive is defined with two parameters, the directive name (expressed in camel case) and a function that returns an object. The first parameter is the directive name you wish to use for the HTML attribute. This name must start with a lower-case letter, then must have one, and only one, upper case letter. It is at the break between the lower-case letters and upper case letter where you use a hyphen in your HTML attribute. It is recommended that you come up with your own unique prefix to avoid name collisions with other libraries you might add. In this case I used ‘pdsa’ as my prefix.

There are many properties that can be returned from the object supplied by the second parameter to the directive function. For a simple validation directive, the object returned from the function simply needs two properties; **require** and **link**. As an example of this object, below is the code you return from the **pdsaValidatenotlowercase** directive.

```
return {
  require: 'ngModel',
  link:
    function (scope, element, attributes, ngModel) {
      ngModel.$validators.pdsavalidatenotlowercase =
        function (value) {
          if (value) {
            return value.toLowerCase() != value;
          }

          return true;
        }
    }
};
```

Starting with the second property, **link**, you see that it is a function that receives four parameters. The only one you are interested in for a validation directive is the last one, **ngModel**. Now you see why the **require** property is needed. When Angular sees that you are requiring the use of ngModel, it knows to pass in the four parameters to the function specified in the link property.

Within the function in the link parameter you add your own function, named pdsavalidatenotlowercase in this sample, to the $validators collection of the current ngModel. When Angular performs its validation, it loops through the $validators collection checking to see what Angular validators are connected, such as ng-minlength and ng-maxlength. By adding your own function into this collection, Angular will call your function. Your function must accept one parameter, the value from the control to which this directive is attached, and the function must return a true or false value. This true or false value can be tested in your HTML and can then display or hide appropriate validation messages. The complete code for all the custom directives you added to the HTML is listed below.

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

  angular.module("app")
   .directive('pdsaValidatenotlowercase',
    function () {
    return {
      require: 'ngModel',
      link: function (scope, element,
                      attributes, ngModel) {
        ngModel.$validators
          .pdsavalidatenotlowercase =
          function (value) {
          if (value) {
            return value.toLowerCase() != value;
          }

          return true;
        }
      }
    };
  })
  .directive('pdsaValidatecreditcard',
    function () {
    return {
      require: 'ngModel',
      link: function (scope, element,
                      attributes, ngModel) {
        ngModel.$validators
         .pdsavalidatecreditcard =
          function (value) {
          if (value) {
            value = value.toString().split("-").join("")
              .split(" ").join("");
            return value.length >= 13 &&
                   value.length <= 16;
          }

          return true;
        }
      }
    };
  })
  .directive('pdsaValidatesecuritycode',
    function () {
    return {
      require: 'ngModel',
      link: function (scope, element,
                      attributes, ngModel) {
        ngModel.$validators
          .pdsavalidatesecuritycode =
          function (value) {
          if (value) {
            return value.length === 3 ||
                   value.length === 4;
          }
```

```
            return true;
          }
        }
      };
  });
})();
```

After you have added all the directive code, open the **index.html** and add a reference to your new creditcard.directives.js file below your other script references.

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

Display Validation Messages

In part one of this article series, you added a element in which you added an unordered list and a list item that uses the ng-repeat directive to display messages from the vm.uiState.messages array. Just below this list item you are going to add a series of additional list item elements to display validation messages coming from the Angular validation system.

Each new list item element uses the **ng-show** directive to display the item based on the result of the function call within the ng-show directive. For example, here is the list item to display a message if the user enters all lower-case letters in the Name on Card field.

```
<li ng-show="creditCardForm.nameOnCard.
             $error.pdsavalidatenotlowercase">
  Name on Card must not be all lower case.
</li>
```

Angular looks for a control in the creditCardForm with the name of **nameOnCard**. It then checks its $error property to see if the result of calling the custom directive function is true. If it is, then it knows that the value entered was in all lower-case letters, thus this list item is displayed in the unordered list. The listing below are the list items to add to the unordered list in the messages area to display all Angular validation messages.

```
<ul>
  <li ng-repeat="msg in vm.uiState.messages">
    {{msg.message}}
  </li>

  <li ng-show="creditCardForm.nameOnCard.
               $error.required">
    Name on Card must be filled in.
  </li>
  <li ng-show="creditCardForm.nameOnCard.
               $error.pdsavalidatenotlowercase">
    Name on Card must not be all lower case.
  </li>
  <li ng-show="creditCardForm.cardNumber.
               $error.required">
    Credit Card Number must be filled in.
  </li>
  <li ng-show="creditCardForm.cardNumber.
               $error.pdsavalidatecreditcard">
    Credit Card Number is invalid
  </li>
  <li ng-show="creditCardForm.securityCode.
               $error.required">
    Security Code must be filled in.
  </li>
  <li ng-show="creditCardForm.securityCode.
               $error.pdsavalidatesecuritycode">
    Security Code is invalid
  </li>
  <li ng-show="creditCardForm.billingPostalCode.
               $error.required">
    Billing Postal Code must be filled in.
  </li>
  <li ng-show="!creditCardForm.
               billingPostalCode.$valid">
    Billing Postal Code must have 18
    characters or less.
  </li>

</ul>
```

Run this sample and enter various combinations of bad data to test out each of the validation error messages.

Add Server-Side Validation

Now that you have all the client-side validation working, add similar functionality to the server-side code as well. As it is very simple for a hacker

to bypass client-side validation, you always check to ensure the data is validated on the server-side as well. To do this you add a new class to the DataLayer project named **PTC-Extension**. After the file is added, rename the class inside of the file to **PTC** and make it a partial class. This will allow us to add additional functionality to the PTC Entity Framework model created in part two of this article series.

```
public partial class PTC
{

}
```

When you attempt to insert or update data using the Entity Framework, it first calls a method named ValidateEntity to perform the validation on any data annotations added to each property. You may override this method to add your own custom validations. Add the following code to the PTC class in the PTC-Extension.cs file you just added.

```
protected override DbEntityValidationResult
  ValidateEntity(DbEntityEntry entityEntry,
  IDictionary<object, object> items) {

  return base.ValidateEntity(entityEntry, items);
}
```

Add a new method named ValidateCreditCard just after the ValidateEntity method you added. In this method is where you add your own custom validations. You return a list of DbValidationError objects for each validation that fails.

```
protected List<DbValidationError>
  ValidateCreditCard(CreditCard entity) {
  List<DbValidationError> list =
    new List<DbValidationError>();


  return list;
}
```

The ValidateEntity method is called once for each entity class in your model that you are trying to validate. In our example, you are only validating the CreditCard object since that is what the user is inputting. The **entityEntry** parameter passed into this method has an **Entity** property which contains the current entity being validated. Write code to check to see if that property is a CreditCard object. If it is, pass that object to the ValidateCreditCard method. The ValidateCreditCard method returns a list of additional DbValidationError

objects that need to be returned. If the list count is greater than zero, then return a new DbEntityValidationResult object by passing in the entityEntry property and your new list of DbValidationError objects.

```
protected override DbEntityValidationResult
  ValidateEntity(DbEntityEntry entityEntry,
    IDictionary<object, object> items) {
  List<DbValidationError> list =
    new List<DbValidationError>();

  if (entityEntry.Entity is CreditCard) {
    CreditCard entity = entityEntry.Entity as CreditCard;

    list = ValidateCreditCard(entity);

    if (list.Count > 0) {
      return new DbEntityValidationResult(entityEntry, list);
    }
  }

  return base.ValidateEntity(entityEntry, items);
}
```

Now write the ValidateCreditCard method to perform the various validations for your credit card data. Check the same validations you performed on the client-side. In this ValidateCreditCard method, you are going to retrieve a the YearsInFuture value from the <appSettings> section in your Web.config file. Add a reference in your DataLayer project to **System.Configuration.dll** so you can use the ConfigurationManager class.

```
protected List<DbValidationError>
   ValidateCreditCard(CreditCard entity) {
  List<DbValidationError> list =
    new List<DbValidationError>();

  // Check Name on Card
  if (entity.NameOnCard.ToLower() == entity.NameOnCard) {
    list.Add(new DbValidationError("NameOnCard",
      "Name on Card must not be all lower case."));
  }

  // Check Card Number
  entity.CardNumber = entity.CardNumber
    .Replace("-", "").Replace(" ", "");
  if (entity.CardNumber.Length < 13 ||
      entity.CardNumber.Length > 16) {
    list.Add(new DbValidationError("CardNumber",
      "Card Number is not valid"));
  }

  // Check Security Code
  if (entity.SecurityCode.Length < 3 ||
      entity.SecurityCode.Length > 4) {
    list.Add(new DbValidationError("SecurityCode",
      "Security Code is not valid"));
  }

  // Check Month and Year
  if (entity.ExpMonth < 1 ||
      entity.ExpMonth > 12) {
    list.Add(new DbValidationError("ExpMonth",
      "Invalid Month."));
  }
  else {
    if (entity.ExpYear < DateTime.Now.Year ||
        entity.ExpYear > DateTime.Now.Year +
            Convert.ToInt32(ConfigurationManager
              .AppSettings["YearsInFuture"])) {
      list.Add(new DbValidationError("ExpYear",
        string.Format("Expiration Year must be
          greater than {0} and less than {1}.",
            DateTime.Now.Year.ToString(),
              ConfigurationManager
                .AppSettings["YearsInFuture"])));
    }
    else {
      if (entity.ExpMonth < DateTime.Now.Month &&
          entity.ExpYear == DateTime.Now.Year) {
        list.Add(new DbValidationError("ExpYear",
          "Expiration Month/Year must be
           greater than this month."));
      }
    }
  }

  return list;
```

```
}
```

Modify handleException Function

The last thing you need to do is to modify the handleException() function located in the creditcard.controller.js file. Add a new case statement to handle a 400 which is a bad request. The BadRequest method is used as the return value in your CreditCardController.cs Post method when the view model property IsValid is set to false. This property is set to false when the Entity Framework raises an exception due to validation failing. Add the code shown below to the handleException function.

```
function handleException(error) {
  vm.uiState.isMessageAreaHidden = false;
  vm.uiState.isLoading = false;
  vm.uiState.messages = [];

  switch (error.status) {
    case 400:   // 'Bad Request'
      // Model state errors
      var errors = error.data.modelState;

      // Loop through and get all
      // validation errors
      for (var key in errors) {
        for (var i = 0; i < errors[key].length;
                i++) {
          vm.uiState.messages.push({
            property: key,
            message: errors[key][i]
          });
        }
      }

      break;

  // PREVIOUS EXCEPTION HANDLING CODE HERE

}
```

To try out the server-side validation, comment out the creditcard.directives.js <script> tag in the index.html file and run the project. Put in a lower-case name in the nameOnCard input field. Fill in valid values for all other fields. Click the Save button to call the Web API to post the data and an error

message should be returned and displayed telling you that you can't have a name that is all lower case.

Summary

In this article, you learned to save the credit card data to your database table using the Entity Framework. In addition, you learned to validate data both on the client-side and the server-side. Over these past three articles you have built a credit card data entry page using Angular, the Web API and C#. You learned how to load drop-down lists, create areas on your page that you can turn off and on, validate data and save data into a database. Hopefully this will give you a lot of ideas on how to build other pages using Angular and the Web API.

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 3**.