Build a Credit Card Entry Page using Angular – Part 2

In the last article, you created an HTML page (Figure 1) to enter credit card information using Angular. You created some hard-coded functions in your Angular controller to populate the three drop-down lists. In this article, you create Web API calls to gather the data for these three drop-down lists from a SQL Server table. These Web API calls request the information for these drop-down lists from a view model class. The view model class uses the Entity Framework (EF) to build a collection credit card types from a SQL Server table, a collection of language-specific month names, and a collection of years. Once you have this built, you call the Web API from your Angular controller to load the drop-down lists from this data instead of the hard-coded data you used in the last article.

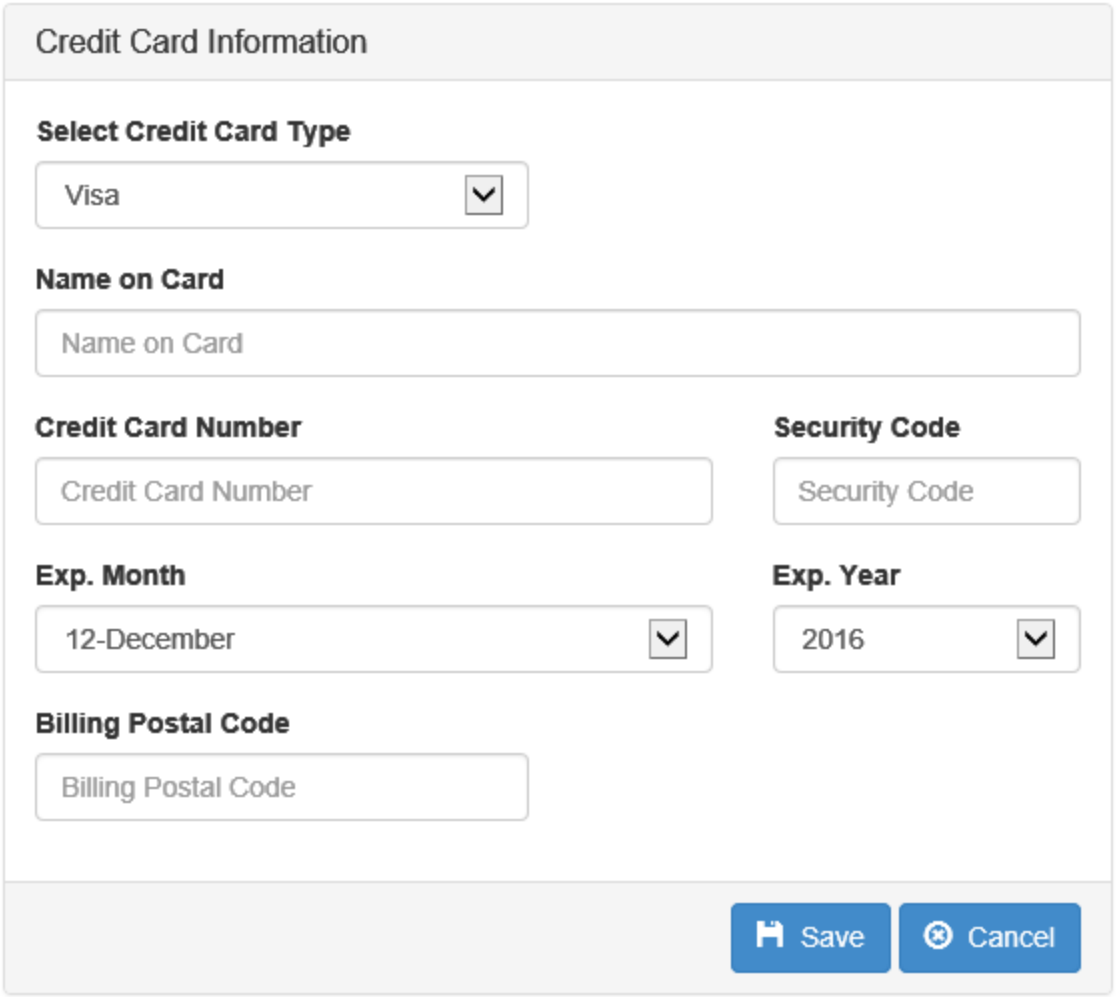

Figure 1: The Credit Card data entry page

Database Design

There are two tables (shown in Figure 2) needed for the credit card page. The first table, named CreditCardType, holds the list of credit card types (Visa, MasterCard, etc.) to be loaded into the drop-down on the web page. The second table, named CreditCard, holds the credit card information entered by the user, and will be used in the next article.

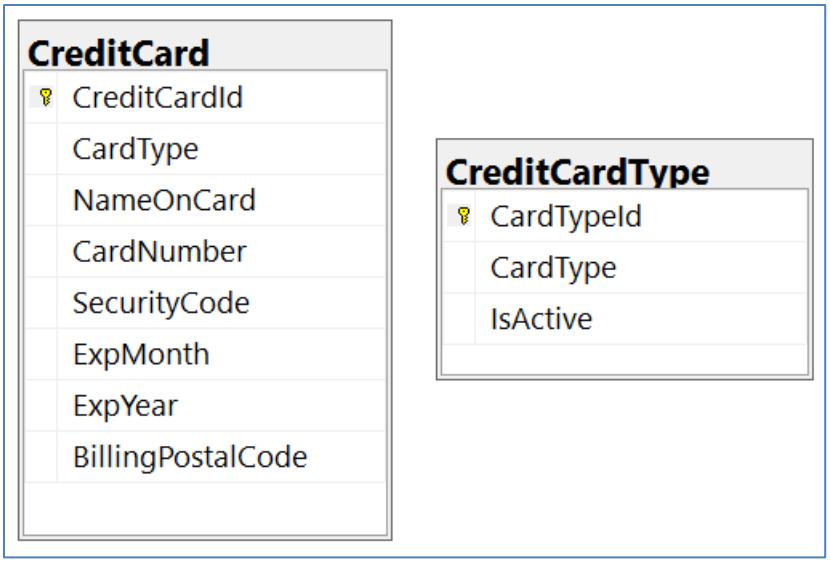

Figure 2: Two tables are needed for credit card information

To create these two tables, open SQL Server Management Studio and select an existing, or create a new database. Open a query window in your new database and type in the following script to create the CreditCard and CreditCardType table.

```
CREATE TABLE CreditCard (
  CreditCardId uniqueidentifier NOT NULL
     PRIMARY KEY NONCLUSTERED,
  CardType varchar(20) NOT NULL,
  NameOnCard varchar(100) NOT NULL,
  CardNumber varchar(25) NOT NULL,
  SecurityCode varchar(4) NOT NULL,
  ExpMonth smallint NOT NULL,
  ExpYear smallint NOT NULL,
  BillingPostalCode varchar(10) NOT NULL
);

CREATE TABLE CreditCardType (
  CardTypeId int IDENTITY(1,1) NOT NULL
    PRIMARY KEY NONCLUSTERED,
  CardType varchar(20) NOT NULL,
  IsActive bit NOT NULL DEFAULT ((1))
);

INSERT CreditCardType (CardType)
       VALUES ('Visa');
INSERT CreditCardType (CardType)
       VALUES ('Master Card');
INSERT CreditCardType (CardType)
       VALUES ('American Express');
INSERT CreditCardType (CardType)
       VALUES ('Discover');
INSERT CreditCardType (CardType)
       VALUES ('Diners Club');
```

Add the Entity Framework

I separate the classes for my data access into a separate DLL. This provides me with the ability to change my data access method later if I want. To do this, right mouse click on your solution and choose **Add | New Project** from the menu. Select Windows | Class Library from the list of templates. Set the Name to **DataLayer** and click the OK button. Delete the **Class1.cs** file from the project as this is not needed.

Right mouse click on the new DataLayer project and select **Add | New Item** from the menu. Select **Data | ADO.NET Entity Data Model** from the list of templates. Set the Name to **PTC** and click the Add button. Choose **Code First from Database** from the list. Create a new connection to the database that contains the CreditCard and CreditCardType tables you created earlier. Select both tables from the list and click the **Finish** button.

After the generation is complete, you have three new classes in your class library project. An App.config file is also created with a <connectionString> element. Move the connection string from the App.config in this project into the Web.config file in your CreditCardEntry project. Open the App.config file and locate the <connectionStrings> element that looks like the following:

```
<connectionStrings>
  <add name="PTC"
     connectionString="data source=localhost;
         initial catalog=PTC;
         integrated security=True;
         MultipleActiveResultSets=True;
         App=EntityFramework"
    providerName="System.Data.SqlClient" />
</connectionStrings>
```

Cut this out of the App.config file, open the Web.config file in the CreditCardEntry project and paste this section into this config file. After you have moved the connection string, delete the App.config file from the DataLayer project. Your data layer is now complete and ready to be used.

Add a View Model

Sticking with our theme of “separation of concerns”, let’s build a view model class to use as the intermediary between the Web API controller and the data access layer. It is a good practice to keep as little code as possible in the controller. By creating a view model class in a separate project it allows you to reuse all the business and data access logic in any other project.

Right mouse click on your CreditCardEntry solution and choose **Add | New Project** from the menu. Select Windows | Class Library from the list of templates. Set the Name to **ViewModelLayer** and click the OK button. Rename the **Class1.cs** file to **CreditCardViewModel**.cs. Answer yes when prompted if you wish to rename the class as well.

Right mouse click on the References folder and select **Add Reference** from the menu. Click on the **Projects | Solution** tab and check the **DataLayer** from the list of projects. Click the OK button to add the reference.

Since you are going to be using Entity Framework generated classes in your view model class you need to add some EF references to this project. Right mouse click on the ViewModelLayer project and select **Manage NuGet Packages** from the menu. Click on the browse tab and type in **Entity Framework** into the search text box and hit the Enter key. Locate the

EntityFramework by Microsoft and click on it. Click the Install button to add all of the appropriate references to the entity framework for this project.

MonthInfo Class

When loading the months into the drop-down list, you are going to need to know the month number as well as the month name. Create a class called **MonthInfo** in the ViewModelLayer project that you can place these two values into. A collection of these objects will be serialized and sent to Angular for loading into the drop-down.

```
public class MonthInfo
{
  public MonthInfo(short number, string name) {
    MonthNumber = number;
    MonthName = name;
  }
  public short MonthNumber { get; set; }
  public string MonthName { get; set; }
}
```

Add using Statements

The CreditCardViewModel class is going to call the CreditCardType class to retrieve the various credit card types to display. This class will retrieve validation error messages from your EF generated classes and generate a list of month names based on the user's current browser language. With all of this functionality, and a few others, it will be necessary to add the following list of using statements at the top of the CreditCardViewModel file.

```
using DataLayer;
using System;
using System.Collections.Generic;
using System.Data.Entity.Validation;
using System.Globalization;
using System.Linq;
```

Create Properties for View Model

Like any view model class, a set of properties are needed to hold the state of the object. Add the following properties to the CreditCardViewModel class.

```
public bool IsValid { get; set; }
public List<DbEntityValidationResult> Messages { get; set; }

public string DefaultLanguage { get; set; }
public string Language { get; set; }

public CreditCard Entity { get; set; }

public List<CreditCardType> CardTypes { get; set; }
public List<MonthInfo> Months { get; set; }
public List<int> Years { get; set; }
```

The **Entity** property holds the current CreditCard object the user is attempting to insert. An **IsValid** property is used to report back if there were validation errors in the current Entity. The **Messages** property is a list of **DbEntityValidationResult** objects. Each of these objects contains a validation error for a field that failed.

As there are three drop-down lists to load on your page, create three properties to hold a collection for each of these; **CardTypes**, **Months** and **Years**. For the months drop-down list, you should strive to display them in the user's language. You can attempt to retrieve the user's language from the browser and pass that to the **Language** property in the view model so it can use the Globalization classes in .NET to retrieve month names in the language of choice. However, if you are unable to retrieve the user's language setting, then you should have a **DefaultLanguage** property you can fill in with a default language setting.

Initialize the Public Properties

It is always a good idea to initialize your properties in your view model class to a valid start state. Build a constructor in your view model class to set each property to an initialized state.

```
public CreditCardViewModel() {
  IsValid = true;
  Messages = new List<DbEntityValidationResult>();

  DefaultLanguage = string.Empty;
  Language = string.Empty;

  Entity = new CreditCard();

  CardTypes = new List<CreditCardType>();
  Months = new List<MonthInfo>();
  Years = new List<int>();
}
```

Load Card Types

In the last article, you hard-coded credit card types in the Angular controller. Now that you have a database table, and the appropriate EF classes, you can retrieve these credit card types from your table and populate your **CardTypes** property. Create a method called LoadCardTypes() and write the code shown below.

```
public void LoadCardTypes() {
  PTC db = new PTC();

  CardTypes = db.CreditCardTypes
              .Where(c => (c.IsActive))
              .OrderBy(c => c.CardType).ToList();
}
```

Load Months

Create a method named **LoadMonths** to build this collection of MonthInfo objects. Using the System.Globalization namespace you attempt to get the month names from the **CultureInfo** class in .NET. The DateTimeFormat property of the CultureInfo class contains a MonthNames collection with the localized month names for the culture specified by the language passed to the constructor of the CultureInfo class. If you pass a bad language specifier, an exception is thrown. You then use the value in the DefaultLanguage property. This DefaultLanguage property is filled in by the Web API controller with data retrieved from the Web.config file.

```
public void LoadMonths() {
  string[] monthNames = null;

  try {
    // Try to get month names
    monthNames = (new CultureInfo(Language))
                   .DateTimeFormat.MonthNames;
  }
  catch (CultureNotFoundException) {
    // Default to a known language
    monthNames = (new System.Globalization
                   .CultureInfo(DefaultLanguage))
                     .DateTimeFormat.MonthNames;
  }

  // Create Months Array
  for (int index = 0; index < monthNames.Length; index++) {
    // NOTE: Month array is 13 entries long
    if (!string.IsNullOrEmpty(monthNames[index])) {
      Months.Add(new MonthInfo(Convert.ToInt16(index + 1),
                               monthNames[index]));
    }
  }

  if (Entity.ExpMonth == 0) {
    // Figure out which month to select
    // Make it next month by default
    Entity.ExpMonth = Convert.ToInt16(DateTime.Now.Month + 1);
    Entity.ExpYear = Convert.ToInt16(DateTime.Now.Year);
    // If past December, then make it January of the next year
    if (Entity.ExpMonth > 12) {
      Entity.ExpMonth = 1;
      Entity.ExpYear += 1;
    }
  }
}
```

Once you have the array of month names, you need to turn these into a collection of MonthInfo objects. Loop through the array and each time through create a new instance of the MonthInfo class, setting the MonthNumber and the MonthName properties from the month names array.

The last thing to do is to default the ExpMonth and ExpYear to some default values. As you do not want your user to put in a month and year less than the current month and year, you add one to the current month and set that into the ExpMonth property. If you add one to the current month and it comes out to be 13, then increment the year by one and place that value into the ExpYear property. Otherwise, you just set the ExpYear to be the current year.

Load Years

The last method to create is to load the years into the Years property. The LoadYears method accepts a specified number of years in the future to load a generic list of integers. If you do not pass in any number, the default of 20 years is used.

```
public void LoadYears(int yearsInFuture = 20) {
  List<int> ret = new List<int>();

  Years = new List<int>();
  for (int i = DateTime.Now.Year;
       i <= (DateTime.Now.Year + yearsInFuture); i++) {
    Years.Add(i);
  }
}
```

Build Web API Controllers

Now that you have a database design, some Entity Framework classes, and a view model to encapsulate the data you need for your credit card page, you are ready to expose that data through a web service. Right mouse click on the **References** folder in the CreditCardEntry project and select **Add Reference**. Add references to your two new projects; **DataLayer** and **ViewModelLayer**.

Credit Card Types Controller

Right mouse click on the Controllers 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 **CreditCardTypeController**. Remove all code within the class and write the following method.

```
public IHttpActionResult Get() {
  IHttpActionResult ret;
  CreditCardViewModel vm = new CreditCardViewModel();

  vm.LoadCardTypes();
  if (vm.CardTypes.Count() > 0) {
    ret = Ok(vm.CardTypes);
  }
  else {
    ret = NotFound();
  }

  return ret;
}
```

This method creates an instance of your CreditCardViewModel class. Call the LoadCardTypes method to load the CardTypes collection in the view model. If card types are loaded, return the status code of 200, via the Ok method, passing in the card types collection as the payload. If no card types are found, then return a 404 using the NotFound method.

Month Names Controller

Right mouse click on the Controllers folder and select **Add | Web API Controller Class (v2.1)**. Set the name of this controller to **MonthNamesController**. Remove all code within the class and write the following method.

```
public IHttpActionResult Get(string id) {
  IHttpActionResult ret;
  CreditCardViewModel vm = new CreditCardViewModel();

  // Set default language
  vm.DefaultLanguage =
    ConfigurationManager.AppSettings["DefaultLanguage"];

  // Set the language passed in
  vm.Language = (string.IsNullOrEmpty(id) ?
                 vm.DefaultLanguage : id);

  vm.LoadMonths();
  if (vm.Months.Count() > 0) {
    ret = Ok(vm.Months);
  }
  else {
    ret = NotFound();
  }

  return ret;
}
```

This method is called from your Angular controller, but you need to pass in a parameter that is named **id**. You must use the parameter name of **id** since this is what the default route is expecting. The id parameter is the language code retrieved from the browser. By passing in the language you can let .NET return the month names in the appropriate language for your user. Call the LoadMonths method in your view model class and if months are loaded, return the months via the Ok method.

If no language is passed to this method, retrieve a default language from your <appSettings> section in the Web.config file and put it into the DefaultLanguage property of your view model. This default value will be used if no language is passed, or an unrecognized language is passed from the browser. Open your Web.Config file and add the DefaultLanguage key and a key for YearsInFuture. You will need the number of years in the next controller.

```
<appSettings>
  <add key="DefaultLanguage"
       value="en-US" />
  <add key="YearsInFuture"
       value="20" />
</appSettings>
```

Years Controller

Right mouse click on the Controllers folder and select **Add | Web API Controller Class (v2.1)**. Set the name of this controller to **YearsController**. Remove all code within the class and write the following method.

```
public IHttpActionResult Get() {
  IHttpActionResult ret;
  CreditCardViewModel vm = new CreditCardViewModel();

  vm.LoadYears(
    Convert.ToInt32(
      ConfigurationManager.AppSettings["YearsInFuture"]));

  if (vm.Years.Count() > 0) {
    ret = Ok(vm.Years);
  }
  else {
    ret = NotFound();
  }

  return ret;
}
```

This method retrieves the YearsInFuture setting from the Web.config file and passes that value to the LoadYears method in your view model class. The Years collection is filled with the number of years specified by the value passed in. If there are years in the Years collection, they are returned via the Ok method.

Call Web API to Load Drop-Down Lists

Now that you have the Web API calls built for loading the drop-down lists, you can now call these from your Angular controller. All the hard-coded functions you wrote in the previous article are going to be rewritten to call the appropriate Web API methods. You also need to add some exception handling to report any errors. Open the \app\creditcard\creditcard.controller.js file and start adding this new functionality.

Handle Exceptions

When you make calls to a Web API you should always make sure you are checking for exceptions. Write a generic handleException function to retrieve any error message information and place an object with a single property called **message** into the vm.uiState.messages array.

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

  switch (error.status) {
    case 404:  // 'Not Found'
      vm.uiState.messages.push(
        {
          message: "The data you were " +
                    "requesting could not be found"
        });
      break;

    case 500:  // 'Internal Error'
      vm.uiState.messages.push(
        {
          message: error.data.exceptionMessage
        });
      break;

    default:
      vm.uiState.messages.push(
        {
          message: "Status: " +
                  error.status +
                  " - Error Message: " +
                  error.statusText
        });
      break;
  }
}
```

Load Card Types

In Part 1 of this article you hard coded a set of credit card types in the loadCardTypes function. Locate the loadCardTypes function in the creditcard.controller.js file and modify the code to look like the following.

```
function loadCardTypes() {
  dataService.get("/api/CreditCardType")
    .then(function (result) {
      vm.cardTypes = result.data;

      if (vm.cardTypes.length > 0) {
        vm.selectedCardType = vm.cardTypes[0];
      }
    },
    function (error) {
      handleException(error);
    });
}
```

In Part 1 of this article, the $http data service was passed into the controller. We assigned this service to the variable named **dataService**. Use the get() function of this data service to call the CreditCardType controller you created earlier to retrieve the credit card types from your SQL Server database. Once you retrieve the card type from the Web API, the **result.data** property is filled in with an array of JSON objects that represent each card type. Assign these values to the vm.cardTypes property because this is the property that is bound to the HTML <select> element that displays them to the user. Finally set the vm.selectedCardType property to the first element in the array in order to position the <select> element to that card type.

Load Month Names

The month names are still hard-coded in the loadMonths function. Locate the loadMonths function and replace the hard-coding with the code to call the Web API you created.

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

  // Get the language from the browser
  var language =
     navigator.languages &&
     navigator.languages[0] || // Chrome / Firefox
     navigator.language ||     // All browsers
     navigator.userLanguage;   // IE <= 10

  dataService.get("/api/MonthNames/" + language)
    .then(function (result) {
      // Transform the data to use nn - monthName format
      for (var index = 0;
               index < result.data.length;
               index++) {
        var month = {
          monthNumber: index + 1,
          monthName: (index + 1).toString()
            + "-" + result.data[index].monthName
        };
        vm.months.push(month);
      }

      // Figure out which month to select
      // Make it next month by default
      vm.creditCard.expMonth = today.getMonth() + 2;
      // If past December, make it January of 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];

      vm.uiState.isLoading = false;
    },
    function (error) {
      handleException(error);
    });
}
```

The call to the Web API is like the other calls you just wrote except you must pass in the current language the browser is running. The navigator object is queried to see which of the **languages**, **language** or the **userLanguage** properties contain a value. This is the value that you pass to the **id** parameter of the MonthNames controller. When the data is returned, loop through each object and convert the data to display in a nn-monthName format, for example; 1-January, 2-February, etc. The rest of the code is what you wrote in the first part of this article to set the default month and year of the drop-downs.

Load Years

The loadYears JavaScript function you wrote in the last article is almost the same as the code you wrote in the Years controller. Having the functionality to load years, months and credit card types in a web service gives us more flexibility rather than hard-coding everything in JavaScript files. Assign the return value from the Years API to the vm.years property as this is the property that is bound to the <select> element used to display the years.

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

  dataService.get("/api/Years")
    .then(function (result) {
      vm.years = result.data;

      vm.creditCard.expYear = year;
    },
    function (error) {
      handleException(error);
    });
}
```

You can now run the sample and see all of the data coming from the Web API calls.

Summary

In this article you created the appropriate Web API calls to load each of the drop-down lists on your credit card data entry page. You built a set of Entity Framework classes to read the credit card types from a SQL Server table. You built three Web API controllers to be called from your Angular functions to load each of the drop-down lists. Finally, you replaced the hard-coded functions you wrote in the first article, with calls to the Web API to get all data from the back-end server. In the next article, you are going to take the credit card data entered by the user, validate that data, and then save that data into a SQL Server table.

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 2**.