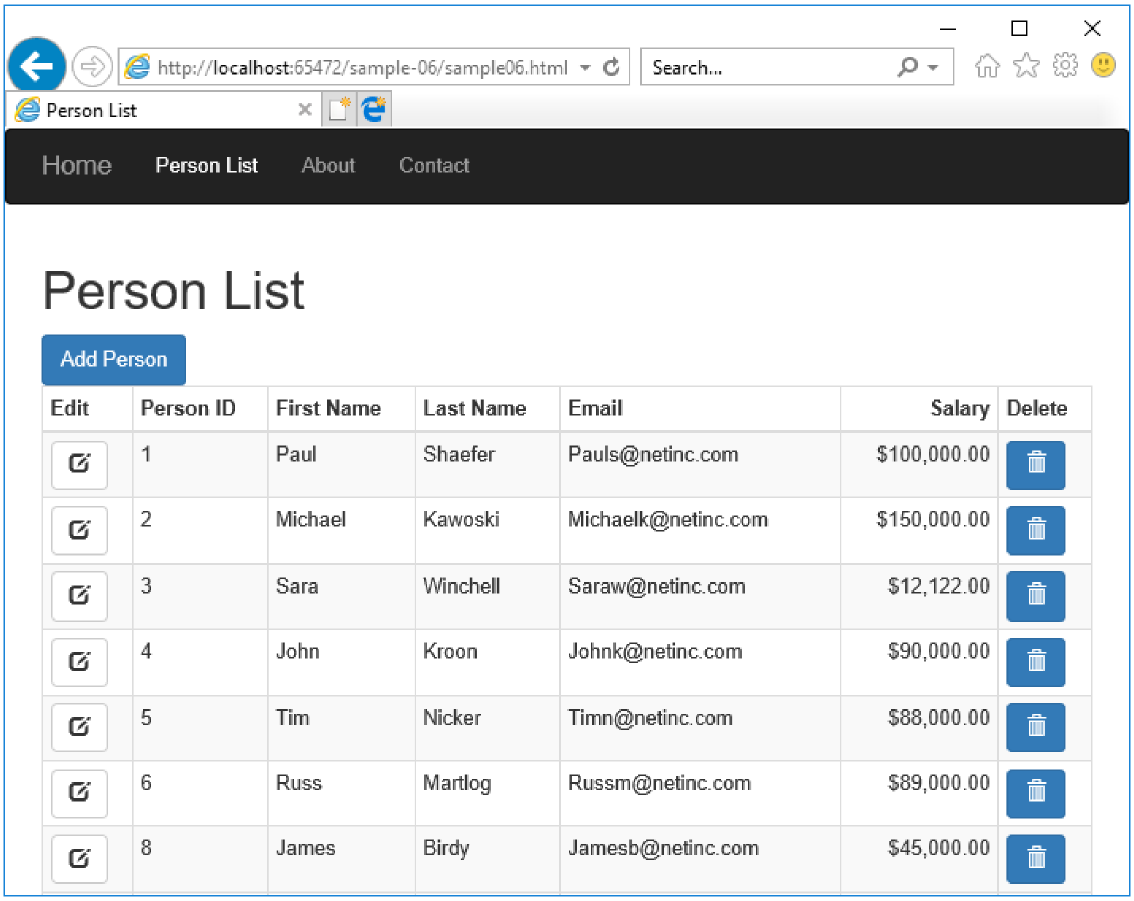
Apply Angular Techniques to jQuery Applications – Part 2

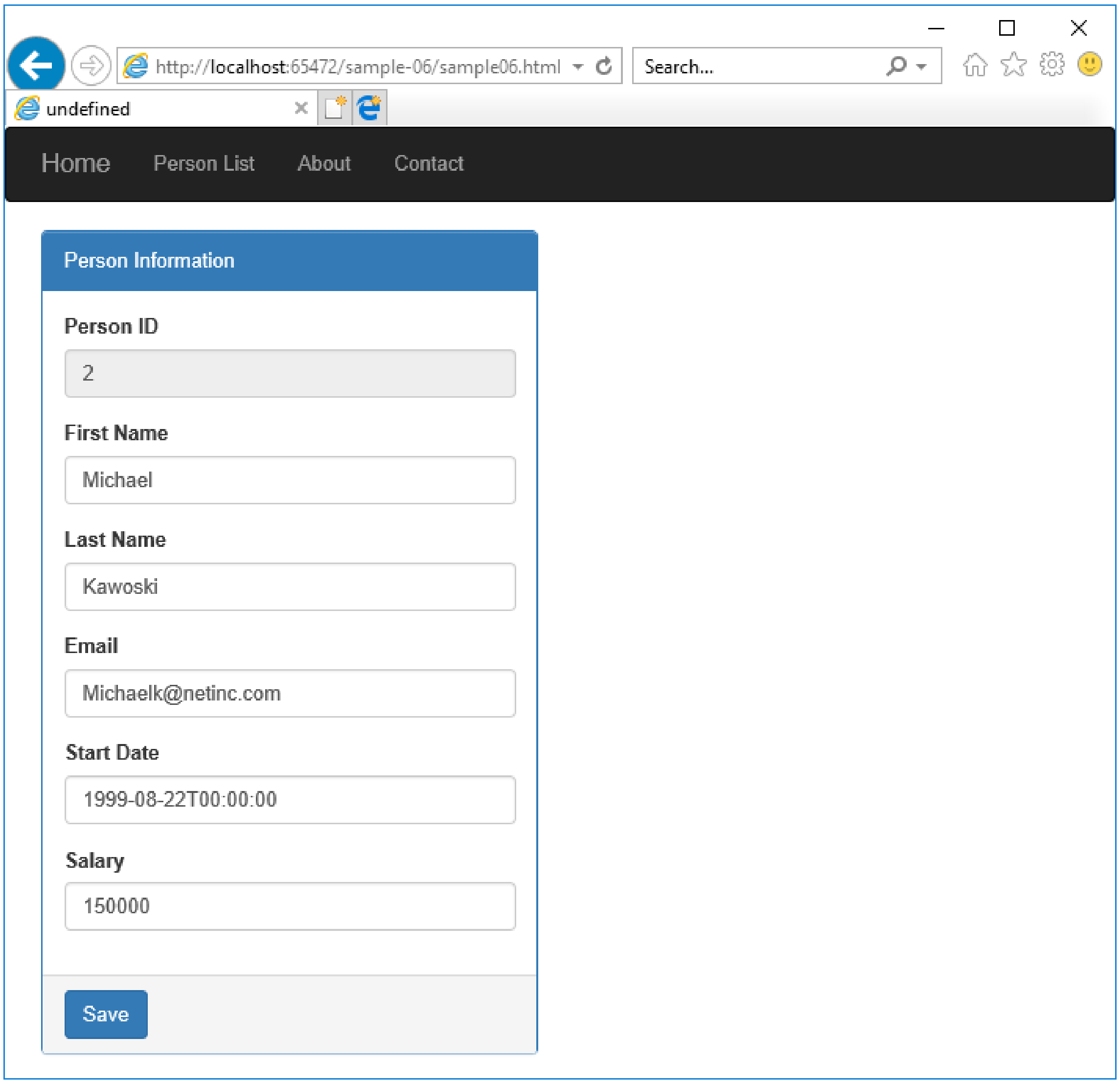
In the last blog post you learned how to structure your jQuery applications like Angular applications. You created a single page on which to host all your other pages. In this post you are going to put those techniques to work by building a complete list, add, edit and delete page as shown in Figure 1 and Figure 2. You are going to use a Person table full of data such as First Name, Last Name, Email and Salary data for a set of people.



Person List

Edit	Person ID	First Name	Last Name	Email	Salary	Delete
	1	Paul	Shaefer	Pauls@netinc.com	$100,000.00	
	2	Michael	Kawoski	Michaelk@netinc.com	$150,000.00	
	3	Sara	Winchell	Saraw@netinc.com	$12,122.00	
	4	John	Kroon	Johnk@netinc.com	$90,000.00	
	5	Tim	Nicker	Timn@netinc.com	$88,000.00	
	6	Russ	Martlog	Russm@netinc.com	$89,000.00	
	8	James	Birdy	Jamesb@netinc.com	$45,000.00	

Figure 1: Person List Page



http://localhost:65472/sample-06/sample06.html

undefined

Person Information

Person ID: 2

First Name: Michael

Last Name: Kawoski

Email: Michaelk@netinc.com

Start Date: 1999-08-22T00:00:00

Salary: 150000

Figure 2: Person Detail Page

Create Server-Side Code

There are several pieces you need to create on the server-side to prepare for calling a Web API from your client-side jQuery CRUD page. You need to do the following.

1. Create a person table
2. Create a web project in Visual Studio 2017
3. Create Entity Framework classes to access your person table

4. Create base Web API controller class
5. Create Person Web API controller
6. Change JSON formatter to return camel-case property names

Person Table

To have some data for this page, create a table in a SQL Server database. Below is the script to create the Person table.

```
CREATE TABLE Person (
  PersonId int PRIMARY KEY NONCLUSTERED
               IDENTITY(1,1) NOT NULL,
  FirstName varchar(50) NOT NULL,
  LastName varchar(50) NOT NULL,
  EmailAddress varchar(250) NULL,
  StartDate datetime NULL,
  Salary money NULL
);
```

Once you have this table created, add some data. You may download the samples for this blog post and use the Person.sql script provided to create this table and the data. See the **Summary** section at the end of this blog post for information on how to download the sample.

Web Project

For this blog post, I am going build the sample project using Visual Studio 2017. Start Visual Studio 2017 and select **File** | **New** | **Project…** from the menu system. From the New Project dialog, select **Visual C#** | **Web** | **ASP.NET Web Application (.NET Framework)**. Set the Name to **jQueryCRUD** and click the OK button. On the **New ASP.NET Web Application** dialog, select the **Web API** template and click the OK button to create your web project.

After the project has been created, right mouse-click on the project and select **Manage NuGet Packages…** from the context-sensitive menu. Click on the Updates tab and if there are any updates, select all packages and update all of them.

Entity Framework Classes

You need a data layer to be able to retrieve and modify the data in the Person table. Use the Entity Framework code generator that is built-into Visual Studio to create this data layer. This code generator generates code to allow us to retrieve, add, edit and delete data within the Person table.

Right mouse-click on the \Models folder and select **Add | New Item…** Click on the **Data** node, then select **ADO.NET Entity Data Model** from the list of templates. When prompted for the item name, type in **PersonDB**. Click the Add button to move to the next step in this process

Select **Code First from database** and click the Next button to create a server connection. Create a new connection to the database where you installed the Person table. After creating the connection, leave everything else the same in this step of the wizard and click the Next button.

Expand the tree view and locate the Person table you previously created. Click the Finish button to have Visual Studio create the Entity Framework data model for the Person table.

Base Web API Controller Class

All Web API controllers you create in Visual Studio inherit from the ApiController class. It is a good practice to create your own base class that inherits from ApiController, then have all your Web API controllers inherit from your base class. By doing this, you can add methods and properties to your base controller class that you can use among all your Web API controllers.

Add a new folder named **Components** to your project. Right mouse-click on the Components folder and add a new class called **BaseApiController**. Into this new class you are going to add two new methods. The first method is going to help each controller deal with exceptions. The second method is going to convert any validation exceptions returned from the Entity Framework into a ModelStateDictionary to be returned to the client application. Add some using statements to the top of this new class.

```
using System;
using System.Data.Entity.Validation;
using System.Diagnostics;
using System.Web.Http;
using System.Web.Http.ModelBinding;
```

Write the rest of the BaseApiController class by typing in the following code:

```
public class BaseApiController : ApiController
{
  protected IHttpActionResult HandleException(Exception ex,
                                              string msg) {
    IHttpActionResult ret;

    // TODO: Add exception publishing here
    Debug.WriteLine(ex.ToString());

    // Create new exception with generic message
    ret = InternalServerError(new Exception(msg, ex));

    return ret;
  }

  protected ModelStateDictionary
   ConvertToModelState(DbEntityValidationException ex)
  {
    ModelStateDictionary ret = new ModelStateDictionary();

    foreach (var list in ex.EntityValidationErrors) {
      foreach (var item in list.ValidationErrors) {
        ret.AddModelError(item.PropertyName,
                          item.ErrorMessage);
      }
    }

    return ret;
  }
}
```

The HandleException method is used in the catch block of any method in your controllers. If an exception, other than a validation exception, is raised by code in your method, you pass the exception object to this method. It creates a status code of 500 by returning an IHttpActionResult created from the InternalServerError() method.

The method ConvertToModelState is used when you attempt to add, or update a person and a business rule fails. The Entity Framework raises a DbEntityValidationException exception when business rules fail. You are going to pass that exception object to this method. The validation errors in this exception are extracted and bundled into a ModelStateDictionary object. This dictionary object is passed back to the client by returning the BadRequest method with the dictionary object as the payload.

Person Controller Class

Open the Controllers folder and delete the ValuesController.cs file. This is just a sample and is not needed. Right mouse-click on the Controllers folder and select **Add | Web API Controller Class (v2.1)** from the menu. Set the name to **PersonController** and click the OK button. Add a few using statements at the top of this file.

```
using System;
using System.Data.Entity.Validation;
using System.Linq;
using System.Web.Http;
using jQueryCRUD.Components;
using jQueryCRUD.Models;
```

Delete all the code within this new controller class as you are going to write your Web API methods using a more updated approach than what is generated by Visual Studio. There are five methods you are going to add to this controller; Get(), Get(id), Post(), Put() and Delete(). Modify the PersonController class to inherit from BaseApiController instead of ApiController.

The code in each of these methods has some commonalities. Each defines an instance of a PersonDB object and creates a new instance of that object within a try…catch block. The catch block calls the HandleException() method you previously defined within the BaseApiController.

The Get() and Get(id) methods return an Ok() or a NotFound() status depending on whether or not they found any data. The Put() and Post() methods can return a BadRequest() if a null person object is passed in, or if validation exceptions are generated from the Entity Framework. The complete PersonController class is presented below.

```
[RoutePrefix("api/Person")]
public class PersonController : BaseApiController
{
  [HttpGet()]
  public IHttpActionResult Get()
  {
    PersonDB db = null;
    IHttpActionResult ret = null;

    try {
      db = new PersonDB();

      if (db.People.Count() > 0) {
        ret = Ok(db.People);
      }
      else {
        ret = NotFound();
      }
    }
    catch (Exception ex) {
      ret = HandleException(ex,
        "Error attempting to retrieve a list of persons");
    }

    return ret;
  }

  [HttpGet()]
  public IHttpActionResult Get(int id)
  {
    PersonDB db = null;
    IHttpActionResult ret = null;
    Person person = null;

    try {
      db = new PersonDB();

      person = db.People.Find(id);
      if (person != null) {
        ret = Ok(person);
      }
      else {
        ret = NotFound();
      }
    }
    catch (Exception ex) {
      ret = HandleException(ex,
       "Error attempting to retrieve a single person");
    }

    return ret;
  }

  [HttpPost()]
  public IHttpActionResult Post([FromBody]Person person)
  {
    PersonDB db = null;
    IHttpActionResult ret = null;

    try {
      db = new PersonDB();

      if (person != null) {
        db.People.Add(person);
        db.SaveChanges();
        ret = Created<Person>(Request.RequestUri +
                              person.PersonId.ToString(),
                              person);
      }
      else {
        ret = BadRequest(
         "Invalid person object passed to POST method");
      }
    }
    catch (DbEntityValidationException ex) {
      ret = BadRequest(ConvertToModelState(ex));
    }
    catch (Exception ex) {
      ret = HandleException(ex,
        "Error attempting to insert person data");
    }

    return ret;
  }

  [HttpPut()]
  public IHttpActionResult Put([FromBody]Person person)
  {
    PersonDB db = null;
    IHttpActionResult ret = null;

    try {
      db = new PersonDB();

      if (person != null) {
        db.Entry(person).State =
           System.Data.Entity.EntityState.Modified;
        db.SaveChanges();
        ret = Ok(person);
      }
      else {
        ret = BadRequest(
          "Invalid person object passed to PUT method");
      }
    }
    catch (DbEntityValidationException ex) {
      ret = BadRequest(ConvertToModelState(ex));
    }
    catch (Exception ex) {
      ret = HandleException(ex,
        "Error attempting to update person data");
    }
```

```
      return ret;
    }

    [HttpDelete()]
    public IHttpActionResult Delete(int id)
    {
      PersonDB db = null;
      IHttpActionResult ret = null;
      Person person = null;

      try {
        db = new PersonDB();

        person = db.People.Find(id);
        if (person != null) {
          db.People.Remove(person);
          db.SaveChanges();
        }
        ret = Ok(true);
      }
      catch (Exception ex) {
        ret = HandleException(ex,
          "Error attempting to delete person data");
      }

      return ret;
    }
  }
```

Camel-Case Property Names

C# property names are generated from the Entity Framework using PascalCase. However, JavaScript programmers are used to camelCase property names. You can have .NET automatically convert the C# property names to camelCase by adding a little bit of code in the Register() method in the WebApiConfig class. Open the \App_Start\WebApiConfig.cs file and make sure you have the following using statements at the top of this file.

```
using System.Linq;
using System.Net.Http.Formatting;
using System.Web.Http;
using Newtonsoft.Json.Serialization;
```

At the bottom of the Register() method, add the following code to perform the camel-case conversion.

```
// Make return results camel case
var jsonFormatter =
 config.Formatters.OfType<JsonMediaTypeFormatter>()
        .FirstOrDefault();

jsonFormatter.SerializerSettings.
    ContractResolver = new
      CamelCasePropertyNamesContractResolver();
```

The above code queries the Formatters collection and retrieves the first instance of a JsonMediaTypeFormatter object it finds. Into the SerializerSettings.ContractResolver property create a new instance of a CamelCasePropertyNamesContractResolver. This property controls how the JSON objects are formatted and sent to the client-side caller.

Display a List of Person Data

If you have not already done so, please read the first part of this blog post. Or, at least download the sample from that blog post as you need the code from that post to continue. You can get the samples at www.pdsa.com/downloads. Choose “PDSA Blogs” from the Category, then select “Apply Angular Techniques to jQuery Applications – Part 1”.

With the server-side code in place, you are now ready to create the HTML and the JavaScript/jQuery for the client-side code. To do this, you are going to perform a few tasks.

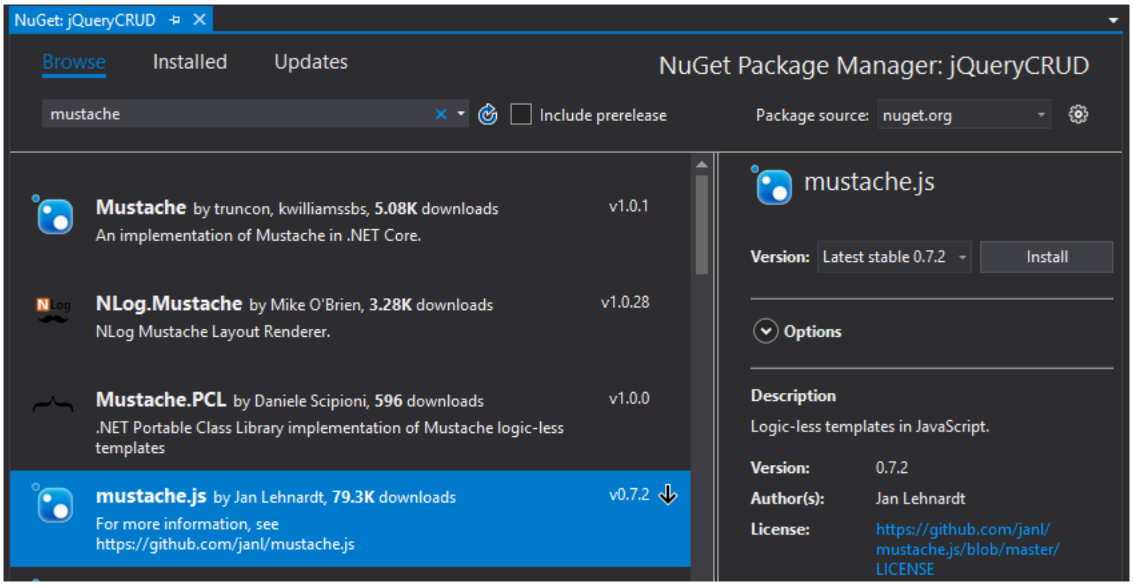
1. Copy files from previous blog post
2. Create a person.service.js file to call the Web API
3. Add mustache.js (or any templating framework) to your project
4. Create a person.list.html file to list all persons
5. Create a person.list.js file to call the person service and load the list of persons into an HTML table
6. Create a person.detail.html file to allow for inputting person information
7. Create a person.detail.js file to retrieve, add, and edit person data

Copy Files from Previous Post

Locate the \src folder in the samples from the previous blog post and copy the complete folder into your Visual Studio project. Copy the spa-common.js file from the \scripts folder and paste it into the \Scripts folder of your project. Right mouse-click on the \src\index.html file and choose **Set As Start Page** from the context-sensitive menu. Run the project and make sure your index.html page loads correctly.

Add Mustache.js to your Project

Instead of writing a bunch of code in JavaScript to load the person table, let's use a templating framework to build the table. There are many different templating frameworks you can utilize. I am going to use Mustache.js (https://github.com/janl/mustache.js/) for this article. Feel free to substitute your favorite templating framework. Open the NuGet Package Manager in your Visual Studio project and search for Mustache.js as shown in Figure 3. Install mustache into your project.



Mustache by truncon, kwilliamssbs, 5.08K downloads v1.0.1
An implementation of Mustache in .NET Core.

NLog.Mustache by Mike O'Brien, 3.28K downloads v1.0.28
NLog Mustache Layout Renderer.

Mustache.PCL by Daniele Scipioni, 596 downloads v1.0.0
.NET Portable Class Library implementation of Mustache logic-less templates

mustache.js by Jan Lehnardt, 79.3K downloads v0.7.2
For more information, see https://github.com/janl/mustache.js

mustache.js
Version: Latest stable 0.7.2

Description
Logic-less templates in JavaScript.

Version:	0.7.2
Author(s):	Jan Lehnardt
License:	https://github.com/janl/mustache.js/blob/master/LICENSE

Figure 3: Add mustache.js to your project

Add a link on the index.html page to the \Scripts\mustache.js file.

```
<script src="../Scripts/mustache.js"></script>
```

Create Person Service Closure

Let's start building our person files. Right mouse-click on the \src folder and add a new folder called **\person**. Add a new JavaScript file named **person.service.js**. Add a closure and assign it to a variable called personService. In this closure you are going to add all the methods required to retrieve all persons, get a single person, insert, update and delete a person, by calling the Person Web API you created earlier in this post.

Start by creating a method named getAll(). This method makes an ajax call to the Get() method in your Web API controller. To this method you pass two callback functions. The first function is called when person data is successfully retrieved from the ajax call. The second function is called when an exception occurs.

```
var personService = (function () {
  const API_URL = "/api/Person/";

  function getAll(success, failure) {
    // Get a list of data
    $.ajax({
      url: API_URL,
      type: 'GET',
      dataType: 'json'
    })
      .done(function (data) {
        success(data);
      })
      .fail(function (error) {
        if (failure) {
          failure(error);
        }
        else {
          console.error("Error Occurred: " + error);
        }
      });
  }

  // Public Functions
  return {
    getAll: function (success, failure) {
      getAll(success, failure);
    }
  };
})();
```

Add a link on the index.html page to the person.service.js file.

```
<script src="person/person.service.js"></script>
```

Create Person List HTML

Create a person.list.html file in the \person folder and add the appropriate HTML to build a person table. Leave the <tbody> element blank. This will be filled in by using the template defined in the <script> tag with the id of “dataTmpl”.

```
<div class="row">
  <div class="col-xs-12">
    <h1>Person List</h1>
  </div>
</div>

<div class="row">
  <div class="col-xs-12">
    <table id="people"
           class="table table-bordered
                  table-condensed table-striped">
      <thead>
        <tr>
          <th>Person ID</th>
          <th>First Name</th>
          <th>Last Name</th>
          <th>Email</th>
          <th class="text-right">Salary</th>
        </tr>
      </thead>
      <tbody></tbody>
    </table>
  </div>
</div>

<script src="./person/person.list.js"></script>
<script id="dataTmpl" type="text/html">
  {{#dataCollection}}
  <tr>
    <td>{{personId}}</td>
    <td>{{firstName}}</td>
    <td>{{lastName}}</td>
    <td>{{emailAddress}}</td>
    <td class="text-right">{{salaryAsCurrency}}</td>
  </tr>
  {{/dataCollection}}
</script>
```

Create Person List Closure

Now that you have the HTML for displaying a person table, build a person list closure to call the person service, and have mustache render the table. This person list closure contains methods to make the calls and render the data using mustache. Add a new JavaScript file named **person.list.js** to the \person folder. Add the following code in this file.

```
$(document).ready(function () {
  personListComponent.getAll();
});

// Closure for Person List Page
var personListComponent = (function () {
  // Private Variables
  var vm = {
    people: []
  };

  // Private Functions
  function renderData(templateId, insertInto) {
    // Get template from script element
    var template = $(templateId).html();
    // Call Mustache passing in the template and the
    // object with collection of data to display
    var html = Mustache.to_html(template,
                                personListComponent);
    // Insert the rendered HTML into the DOM
    $(insertInto).html(html);
  }

  // Callback function from Service
  function getAllSuccess(data) {
    // Assign data to array
    vm.people = data;
    // Create HTML table
    renderData("#dataTmpl", "#people tbody");
  }

  function getAll() {
    // Call Service to get list of data
    personService.getAll(getAllSuccess);
  }

  // Public Functions
  return {
    getAll: function () {
      getAll();
    },
    salaryAsCurrency: function () {
      return new Number(this.salary)
        .toLocaleString('en-US',
          { style: 'currency', currency: 'USD' });
    },
    dataCollection: function () {
      return vm.people;
    }
  };
})();
```

Add a new menu item to the index.html page to call the person list page.

```
<a href="#person.list"
   data-page-path="person/"
   title="Person List">
  Person List
</a>
```

Run the page and you should see your list of persons in the HTML table.

Display a Single Person

Create a new HTML page to display each of the columns in the Person table. You are going to use this page to add and edit the person data. Right mouse-click on the \person folder and create a new HTML page named **person.detail.html**. Make this HTML page look like the code below.

```
<div class="row">
  <div class="col-sm-6">
    <div class="panel panel-primary">
      <div class="panel-heading">
        Person Information
      </div>
      <div class="panel-body">
        <div class="form-group">
          <label for="personId">
            Person ID
          </label>
          <input type="number" id="personId"
                 readonly="readonly"
                 class="form-control" />
        </div>
        <div class="form-group">
          <label for="firstName">
            First Name
          </label>
          <input type="text" id="firstName"
                 class="form-control" />
        </div>
        <div class="form-group">
          <label for="lastName">
            Last Name
          </label>
          <input type="text" id="lastName"
                 class="form-control" />
        </div>
        <div class="form-group">
          <label for="emailAddress">Email</label>
          <input type="email" id="emailAddress"
                 class="form-control" />
        </div>
        <div class="form-group">
          <label for="startDate">Start Date</label>
          <input type="date" id="startDate"
                 class="form-control" />
        </div>
        <div class="form-group">
          <label for="salary">Salary</label>
          <input type="number" id="salary"
                 class="form-control" />
        </div>
      </div>
    </div>
  </div>
</div>
<script src="./person/person.detail.js"></script>
```

Create Person Detail Closure

Add a JavaScript file to the \person folder named **person.detail.js**. This is the closure with all the code for displaying the person data on the person.detail.html page.

```
$(document).ready(function () {
  personDetailComponent.get();
});

// Closure for person.detail page
var personDetailComponent = (function () {
  // Private Variables
  var vm = {
    person: {}
  };

  // Private Functions
  function getSuccess(data) {
    // Assign data to object
    vm.person = data;
    // Display fields in HTML inputs
    displayData(vm.person);
  }

  function get() {
    // Assuming the following url: #product.detail/n
    var id = window.location.hash;
    if (id.lastIndexOf("/") >= 0) {
      // Extract the ID portion from the hash
      id = id.substring(id.lastIndexOf("/") + 1);
    }
    else {
      id = null;
    }

    if (id) {
      // Call Service to get data
      personService.getPerson(id, getSuccess);
    }
  }

  function displayData(person) {
    $("#personId").val(person.personId);
    $("#firstName").val(person.firstName);
    $("#lastName").val(person.lastName);
    $("#emailAddress").val(person.emailAddress);
    $("#startDate").val(person.startDate);
    $("#salary").val(person.salary);
  }

  // Public Functions
  return {
    get: function () {
      get();
    }
  };
})();
```

Add Button to Call Person Detail

To get to the person detail page, you need to add a button to the HTML table you created on the person list page. Open the person.list.html page and add a new table header.

```
<th>Edit</th>
```

In the mustache template add a new <td> with an anchor tag that calls the person.detail page you just added.

```
<td>
  <a href="#person.detail/{{personId}}"
     data-page-path="./person/"
     title="Person Information"
     class='btn btn-default'>
    <span class='glyphicon glyphicon-edit' />
  </a>
</td>
```

One thing that was not accounted for in the previous blog post, was a forward slash after the page name in the href attribute. You added a person id after the page name of person.detail in the href. In order to get the page name, you must remove the forward slash and anything after it in order to get just page name. Open the spa-common.js file and locate the changePage() method. After the line of code that removes the # sign to create the page name, add code to remove data after the last forward slash from the page name.

```
function changePage(contentArea, hashValue) {
  // Get path for partial page
  var path = $("a[href='" + hashValue + "']")
    .data("page-path") || "";
  // Remove # to create the page file name
  var pageName = hashValue.substr(1);
  // Remove any trailing data after the slash
  if (pageName.lastIndexOf("/") >= 0) {
    pageName = pageName.substring(0,
     pageName.lastIndexOf("/"));
  }

  // Rest of the code
}
```

Update Person Service

Update the personService closure to add a method to retrieve a single person record from the Web API. Open the person.service.js file and add the following method.

```
function get(id, success, failure) {
  // Get a single row of data
  $.ajax({
    url: API_URL + id,
    type: 'GET',
    dataType: 'json'
  })
    .done(function (data) {
      success(data);
    })
    .fail(function (error) {
      if (failure) {
        failure(error);
      }
      else {
        console.error("Error Occurred: " + error);
      }
    });
}
```

Modify the return statement within this closure to expose this new method.

```
return {
  getPeople: function (success, failure) {
    getPeople(success, failure);
  },
  get: function (id, success, failure) {
    get(id, success, failure);
  }
};
```

Run the page and you should now be able to display the person detail page with a specific person’s data filled in.

Update a Person

After you have displayed a person in the appropriate input fields on the person detail page, you can now allow the user to change the data, click on a Save button and modify the data by calling the Web API PUT() method. Open the person.detail.js file and add the following methods in the closure.

```
function getDataFromInput() {
  return {
    personId: $("#personId").val(),
    firstName: $("#firstName").val(),
    lastName: $("#lastName").val(),
    emailAddress: $("#emailAddress").val(),
    startDate: $("#startDate").val().replace(/[^ -z]/g, ''),
    salary: $("#salary").val()
  }
}

function save() {
  // Gather data from HTML inputs
  vm.person = getDataFromInput();

  // Update data
  if (vm.person) {
    updateData();
  }
}

function updateDataSuccess(data) {
  // Return to list page
  window.history.back(-1);
}

function updateData() {
  // Get data from HTML
  vm.person = getDataFromInput();
  // Call Service to update data
  personService.updateData(vm.person, updateDataSuccess);
}
```

At the bottom of the closure, add a new public method, save(), to the return statement.

```
return {
  get: function () {
    // NO CHANGES HERE
  },
  save: function () {
    save();
  },
  cancel: function () {
    window.history.back(-1);
  }
};
```

Open the person.detail.html page and add a footer to the Bootstrap panel. Add a Save and a Cancel button in a row in this footer.

```
<div class="panel-footer">
  <div class="row">
    <div class="col-xs-12">
      <button type="button"
              id="btnSave"
              class="btn btn-primary"
              onclick="personDetailComponent.save();">
        Save
      </button>
      <button type="button"
              id="btnCancel"
              class="btn btn-primary"
              onclick="personDetailComponent.cancel();">
        Cancel
      </button>
    </div>
  </div>
</div>
```

Update Person Service

To call the Put() Web API method, add a new method to the person service closure. Open the person.service.js file and add the following method.

```
function updateData(person, success, failure) {
  // Update single row of data
  $.ajax({
    url: API_URL + person.personId,
    type: 'PUT',
    data: person,
    dataType: 'json'
  })
    .done(function (data) {
      success(data);
    })
    .fail(function (error) {
      if (failure) {
        failure(error);
      }
      else {
        console.error("Error Occurred: " + error);
      }
    });
}
```

Modify the return statement in this closure to expose this updateData() method. Add this new signature after the get() method in the return statement as shown below.

```
updateData: function (person, success, failure) {
  updateData(person, success, failure);
}
```

Run your application and click on one of the Edit buttons. Change some of the person data, and click the Save button. You should see the changes appear in the person list.

Add a Person

Open person.list.html and create an **Add Person** button above the HTML table.

```
<div class="row">
  <div class="col-xs-12">
    <a href="#person.detail"
       data-page-path="./person/"
       title="Person Information"
       id="btnAdd"
       disabled="disabled"
       class='btn btn-primary'>
      Add Person
    </a>
  </div>
</div>
```

The Add Person button is disabled until all items in the person list have been displayed. Open the person.list.js and add a new line of code as the last line in the getAllSuccess() method. This line of code enables the Add Person button after all line items have been successfully loaded.

```
// Enable Add button
$("#btnAdd").removeAttr('disabled');
```

Let's also add a function to handle exceptions in the person.list.js file. For now, let's just check for a 404 which means that no data was found. If there are no records in the person table, you still want to enable the Add Person button. Add the handleException() method shown below.

```
function handleException(error) {
  switch (error.status) {
    case 404:
      // Enable Add Person button
      $("#btnAdd").removeAttr('disabled');
      break;

    default:
      break;
  }
}
```

In the getAll() method add the handleException() method as the second parameter in the call to the personService.getAll() method.

```
function getAll() {
  // Call Service to get list of data
  personService.getAll(getAllSuccess, handleException);
}
```

If you run the page right now, you should see the Add Person button is disabled until the list of person data loads. Don't click on the Add Person button yet, as we still need to do a little more work.

Update Person Service

Add a new method to the person.service.js file to call the Post() method in the Web API.

```
function addData(person, success, failure) {
  // Add a single row of data
  $.ajax({
    url: API_URL,
    type: 'POST',
    data: person,
    dataType: 'json'
  })
    .done(function (data) {
      success(data);
    })
    .fail(function (error) {
      if (failure) {
        failure(error);
      }
      else {
        console.error("Error Occurred: " + error);
      }
    });
}
```

Modify the return statement in this closure to expose this addData() method. Add this new signature after the updateData() method in the return statement.

```
addData: function (person, success, failure) {
  addData(person, success, failure);
}
```

Modify get() Method

Open the person.detail.js file and modify the get() method to create an empty person object if no personId is passed in. You can set any defaults that you want. In the code below, I set the startDate to today's date and the salary to zero.

```
function get(id) {
  if (id) {
    // Call Service to get data
    personService.get(id, getSuccess);
  }
  else {
    vm.person = {
      personId: null,
      firstName: "",
      lastName: "",
      emailAddress: "",
      startDate: new Date().toLocaleString(),
      salary: 0
    };
    // Display fields in HTML inputs
    displayData(vm.person);
  }
}
```

Create an addData() Method

Add a new method called addData() to your person.detail.js file. This method is called when you need to add a new person. It is responsible for calling the addData() method in the person service.

```
function addDataSuccess(data) {
  // Return to list page
  window.history.back(-1);
}

function addData() {
  // Get data from HTML
  vm.person = getDataFromInput();
  // Call Service to update data
  personService.addData(vm.person, updateDataSuccess);
}
```

Modify save() Method

Locate the save() method in the person.detail.js file and modify the code to check to see if the personId property is not null. If not, then you call the updateData() method. Otherwise you call the new addData() method.

```
function save() {
  // Gather data from HTML inputs
  vm.person = getDataFromInput();

  // Save data
  if (vm.person.personId) {
    updateData();
  }
  else {
    addData();
  }
}
```

Go ahead and run the application and add a new person. Click on the Save button and you should see the person appear in the person list.

Delete a Person

To delete a person, add a new Delete button to the person HTML table. Open the person.list.html page and add a new table header.

```
<th>Delete</th>
```

In the data template, add a new table detail column.

```
<td>
  <button class='btn btn-primary'
          onclick="personListComponent
                   .deleteData({{personId}});">
    <span class='glyphicon glyphicon-trash' />
  </button>
</td>
```

Open the person.list.js file and add the following two methods.

```
function deleteDataSuccess(data) {
  // Refresh the List
  getAll();
}

function deleteData(id) {
  personService.deleteData(id, deleteDataSuccess);
}
```

In the return statement of the personListComponent closure, add the following method call.

```
deleteData: function (id) {
  if (confirm("Delete this Person?")) {
    deleteData(id);
  }
}
```

Update Person Service

Add a new method to the person service to delete a person record. Open the person.service.js file and add the following method.

```
function deleteData(id, success, failure) {
  // Delete a single row of data
  $.ajax({
    url: API_URL + id,
    type: 'DELETE',
    dataType: 'json'
  })
    .done(function (data) {
      success(data);
    })
    .fail(function (error) {
      if (failure) {
        failure(error);
      }
      else {
        console.error("Error Occurred: " + error);
      }
    });
}
```

Add a new method to the return statement at the bottom of this closure.

```
deleteData: function(id, success, failure) {
  deleteData(id, success, failure);
}
```

Run the page and delete one of the person records. After deleting the record, you should see the list refresh, and the record you deleted has been removed.

Summary

In this blog post you took the single page techniques you learned in the previous blog post and created a CRUD page. You learned to break up your HTML and your JavaScript into separate files. This structure keeps your application broken up into small, discrete, easily-modified and maintainable chunks. The design patterns presented in this article will help you move toward some of the frameworks in use today such as VueJS, Angular and React.

You can get the samples at www.pdsa.com/downloads. Choose “PDSA Blogs” from the Category, then select “Apply Angular Techniques to jQuery Applications – Part 2”.