

Design and Develop a website using ASP.NET MVC 4, EF, Knockoutjs and Bootstrap

<https://ddmvc4.codeplex.com/>

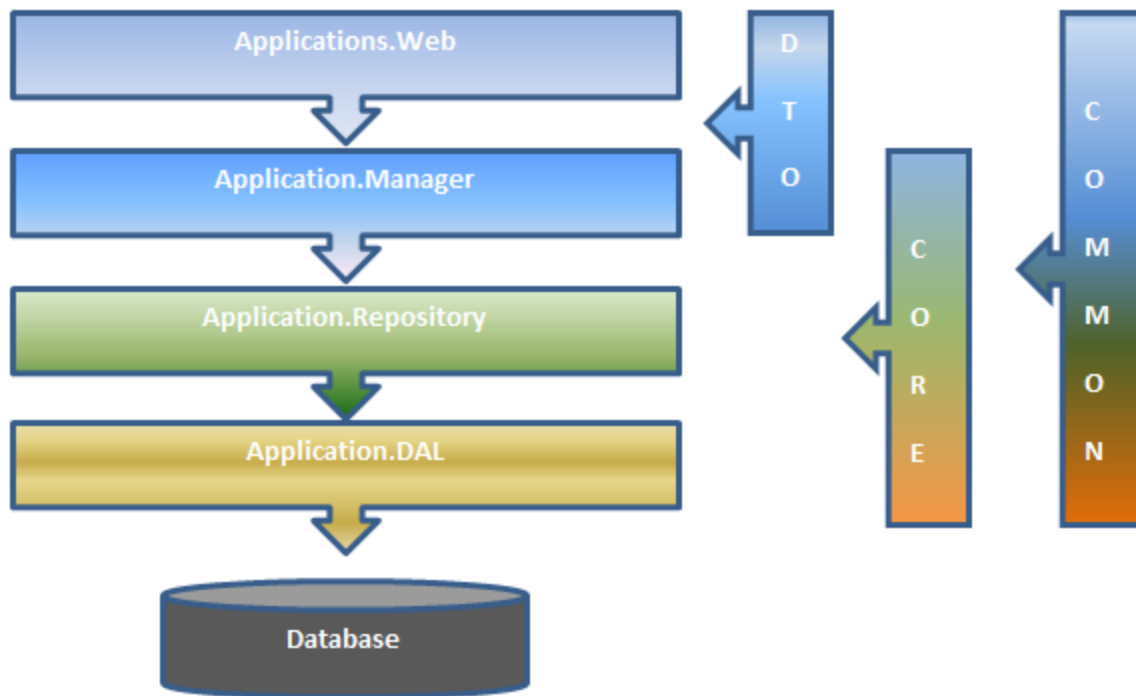
Another MVC Application: Introduction

All websites are growing faster these days, and once it grows, it is very hard to write, organize and maintain. As we add new functionality or developer to a project, any large web applications with poor design may be go out of control. So the idea behind this article is to design a website architecture that must be simple, easily understandable by any web designer (beginner to intermediate) and Search Engines. For this article I am trying to design a website for any individuals to maintain their contact details online. However, in future, the same application could be used by large community all over the world with added functionality and modules. So, the design should be easily adaptable in order to cope with the future growth of business.

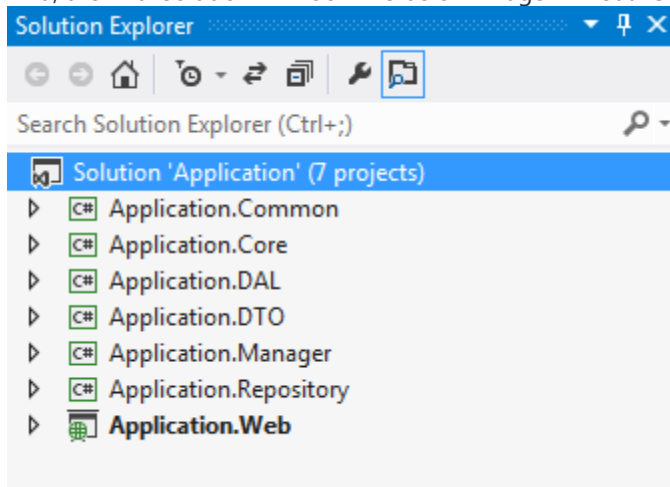
In this article I will talk about creating and designing User Interface (UI) in such a manner so that UI will be separated from the business logic, and can be created independently by any designer/developer. For this part we will use ASP.Net MVC, Knockout JQuery and Bootstrap. later in this article we will discuss more about database design and implementing business logic using structured layers using SQL Server 2008, Entity Framework, and Castle Windsor for Dependency Injection.

Separation of Concern: Primary Objective

The key concept is stripping out most or all logic. Logic should not be bound to a page. What if we need to re-use the logic from one page in another? In that case we will be tempted to copy-and-paste. If we are doing this then our project will become maintainable. Another important concept is to separate data access layer from any business logic, as we are planning to use Entity Framework this is less of a problem as EF should already have this end separate. We should be able to easily move all our EF files to another project and simply add a reference to the projects that need it. Below is the high level design:



And, the final solution will look like below image in Visual Studio :



Seven Projects in One Solution : Is it required ?

It is all about your decision...9 The proposed designed will offer some rather relevant benefits, which include:

- Separation of Concern: Design should allow clear and defined layers; means segregate application into distinct areas whose functionality does not overlap. such that UI-designers can focus on their job without dealing with the business logic (Application.Web), and the core developer can only work on main business logic's (Application.DTO or Application.Manager).

- Productivity: It is easier to add new features to existing software, since the structure is already in place, and the location for every new piece of code is known beforehand, so any issue can be easily identified and separated to cope with complexity, and to achieve the required engineering quality factors such as * * robustness, adaptability, maintainability, and re-usability.

Maintainability: It is easier to maintain application, due to clear and defined structure of the code is visible and known, so it's easier to find bugs and anomalies, and modify them with minimal risk.

- Adaptability: New technical features, such a different front ends, or adding a business rule engine is easier to achieve, as your software architecture creates a clear separation of concerns.

Re-usability: Re-usability is another concern on designing any application, because it is one of the main factors to decrease the total cost of ownership, and our design should consider to what extent we can reuse the created Web application and different layers.

In last section of this article, we will discuss the functionality of each individual layer's in details.

Tools & Technology

To achieve the final solution, we need below tools/dll:

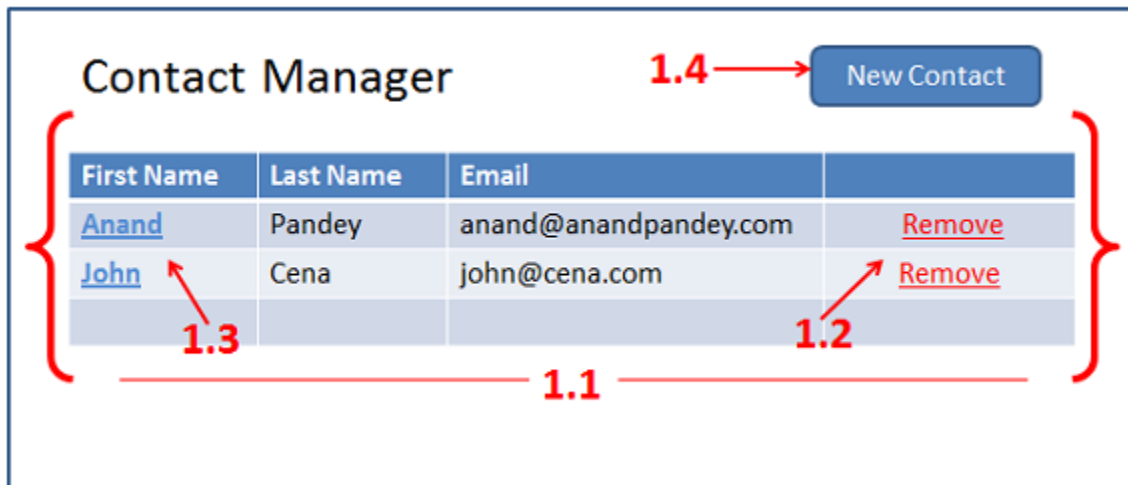
- Visual Studio 2012
- ASP.NET MVC 4 with Razor View Engine
- Entity Framework 5.0
- Castle Windsor for DI
- SQL Server 2008/2012
- Knockout.js & JQuery
- Castle Windsor for DI
- Bootstrap CSS

What we are trying to achieve: Requirement

Screen 1: Contact List - View all contacts

- 1.1 This screen should display all the contacts available in Database.
- 1.2 User should be able to delete any contact.
- 1.3 User should able to edit any contact details.
- 1.4 User should be able to create a new contact.

Initial sketch:



Screen 2: Create New Contact

This screen should display one blank screen to provide functionality as.

- 2.1 User should be able to Enter his/her First name, Last Name and Email Address.
- 2.2 User should be able to add any number of Phone numbers by clicking on Add numbers.
- 2.3 User should be able to remove any phone number.
- 2.4 User should be able to add any number of Addresses by clicking on Add new address.
- 2.5 User should be able to remove any address.
- 2.6 Click on save button should save Contact details in Database and user will return back in Contact List page.
- 2.7 Click on Back to Profile button should return back the user to Contact List page.

Initial sketch:

The image shows a wireframe of a 'Create New Contact' form. It is divided into four main sections: Profile Information, Phone Information, Address Information, and a final Save Profile button. Red arrows and numbers (2.1 through 2.7) point to specific UI elements:

- 2.1** points to the 'First Name', 'Last Name', and 'Email' input fields in the Profile Information section.
- 2.2** points to the 'Add New Phone' button in the Phone Information section.
- 2.3** points to the 'Remove' button next to the 'Number' input field in the Phone Information section.
- 2.4** points to the 'Add New Address' button in the Address Information section.
- 2.5** points to the 'Remove' button next to the 'Select Address Type' dropdown in the Address Information section.
- 2.6** points to the 'Save Profile' button at the bottom right.
- 2.7** points to the 'Back to Profile List' button at the top right.

Screen 3: Update Existing Contact

This screen should display screen with selected contact information details.

- 3.1 User should be able to modify his/her First name, Last Name and Email Address.
- 3.2 User should be able to modify /delete/Add any number of Phone numbers by clicking on Add numbers or delete link.
- 3.3 User should be able to modify /delete/Add any number of Addresses by clicking on Add new address or delete link.
- 3.4 Click on save button should update Contact details in Database and user will return back in Contact List page.
- 3.5 Click on Back to Profile button should return back the user to Contact List page.

Initial Sketch:

The image shows a wireframe of an 'Edit Contact' form. The form is divided into several sections: 'Profile Information', 'Phone Information', and 'Address Information'. Red arrows and numbers (3.1, 3.2, 3.3, 3.4, 3.5) point to specific UI elements. 3.1 points to the 'Profile Information' section header. 3.2 points to the 'Remove' button for the second phone entry. 3.3 points to the 'Add New Address' button. 3.4 points to the 'Save Profile' button. 3.5 points to the 'Back to Profile List' button. The form contains input fields for name, email, phone numbers, address, and city, as well as dropdown menus for phone type and address type. There are also 'Remove' buttons for each phone and address entry.

Edit Contact

3.1 → **Profile Information**

Anand Pandey anand@anandpandey.com

Phone Information

Work Phone 1-832-832-8328 Remove

Personal Phone 1-238-238-2382 Remove

Add New Phone

Address Information

Billing Address Remove

10000 Richmond Ave Apt # 1000 USA

Texas Houston 70000

Add New Address 3.3

3.4 → Save Profile

3.5 → Back to Profile List

Part 1: Create Web Application (Knockout.js, Asp.Net MVC and Bootstrap): For Designers

Before kick-off the UI part, let us see what benefits we are getting using Knockoutjs and Bootstrap along with ASP.NET MVC 4.

Why Knockoutjs: Knockout is an MVVM pattern that works with a javascript ViewModel. The reason this works well with MVC is that serialization to and from javascript models in JSON is very simple, and it is also included with MVC 4. It allows us to develop rich UI's with a lot less coding and whenever we modify our data, Immediate it reflect in the user interface.

Why Bootstrap: Twitter Bootstrap is a simple and flexible HTML, CSS, and Javascript for popular user interface components and interactions. It comes with bundles of CSS styles, Components and Javascript plugins. It provides Cross platform support, which eliminates major layout inconsistencies. Everything just works! Good documentation and the Twitter Bootstrap's website itself is a very good reference for real life example. And,

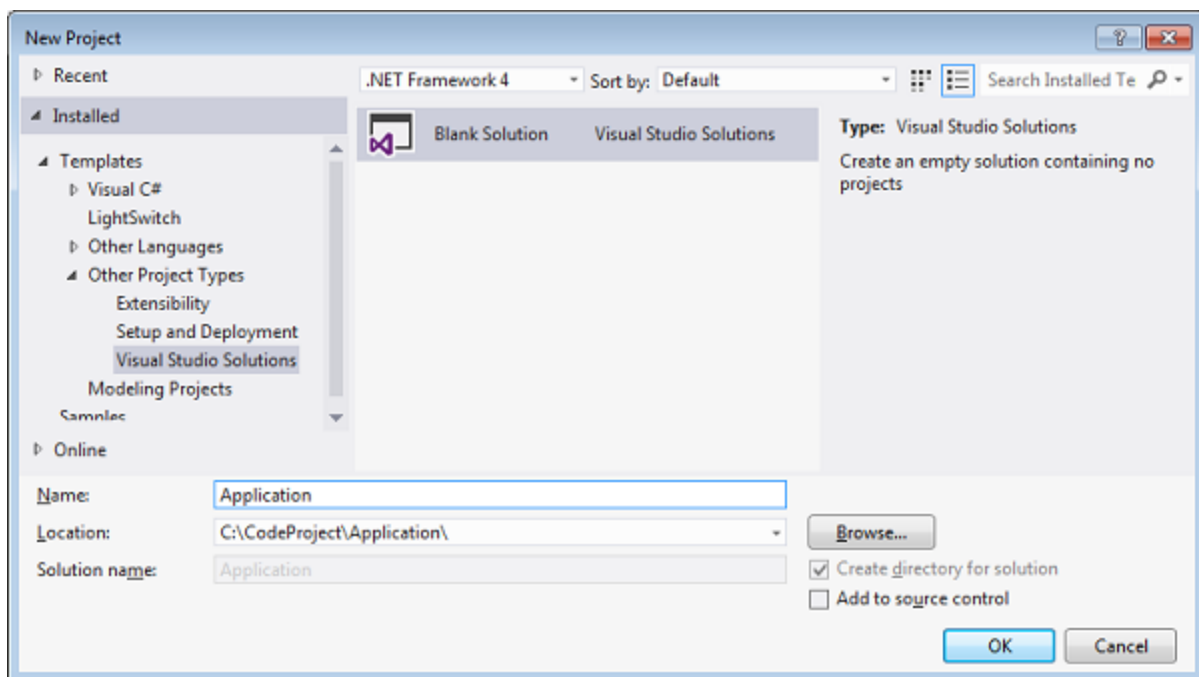
finally, it saves me plenty of times, as it cut the development time to half with very less testing and almost zero browser issues. Some other benefits we can get by using this framework are:

- 12-Column Grid, fixed layout, fluid or responsive layout.
- Base CSS for Typography, code (syntax highlighting with Google prettify), Tables, Forms, Buttons and uses Icons by Glyphicon .
- Web UI Components like Buttons, Navigation menu, Labels, Thumbnails, Alerts, Progress bars and misc.
- Javascript plugins for Modal, Dropdown, Scrollspy, Tab, Tooltip, Popover, Alert, Button, Collapse, Carousel and Typehead.

In below steps, we will work through the layout and design to build UI for above requirement by using dummy javascript data.

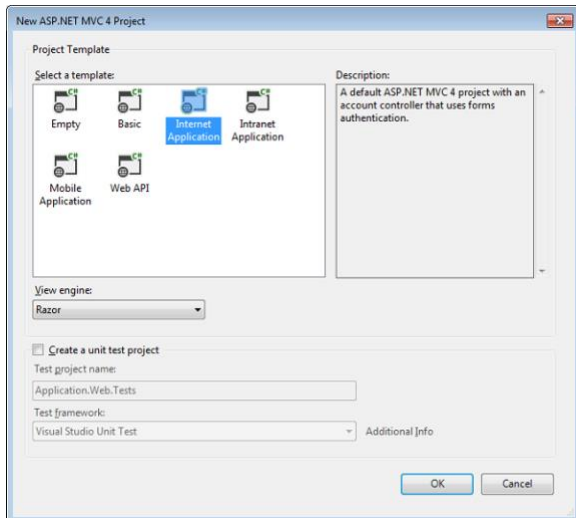
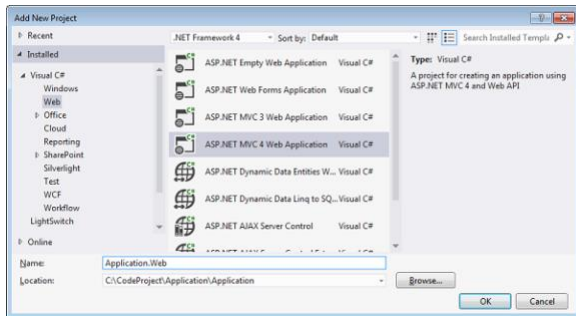
Step 1:

Create a new project as Blank Solution; name it as "Application"

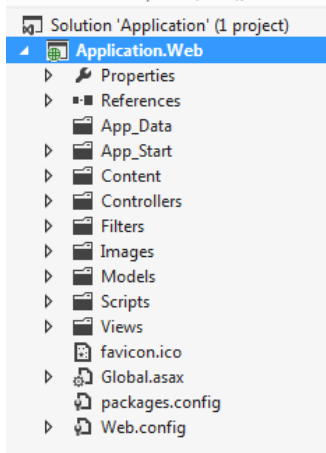


Step 2:

Right Click on Solution folder and add new Project of type ASP.NET MVC 4 as an Internet Application Template with View engine as Razor.

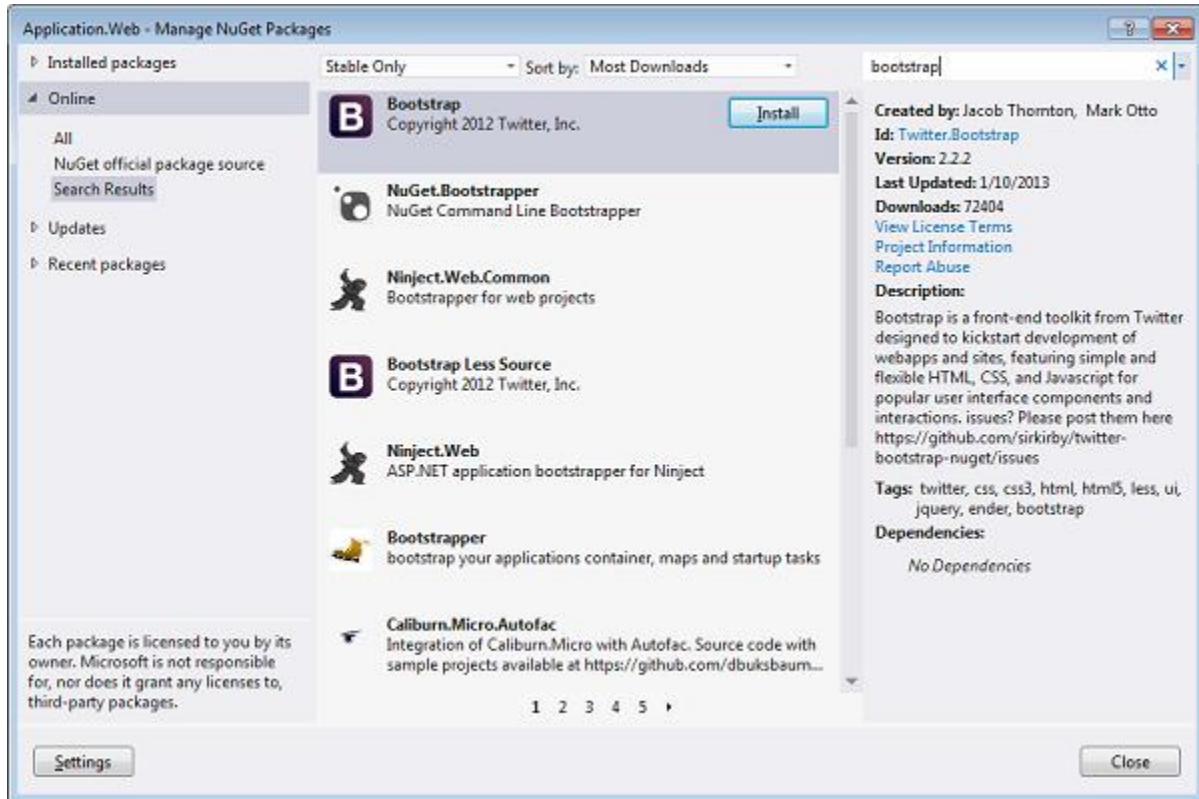


After Step 2 - the project structure should look like the below image



Step 3:

Right click on References and click on Manage NuGet Packages. Type Bootstrap on search bar then click on Install button.



Step 4:

add below line of code into BundleConfig.cs file under App_Start folder to add Knockoutjs and Bootstrap for every page

```
bundles.Add(new ScriptBundle("~/bundles/knockout").Include(
    "~/Scripts/knockout-{version}.js"));
bundles.Add(new StyleBundle("~/Content/css").Include("~/Content/bootstrap.css"));
```

Also in _Layout.cshtml file under Views/Shared folder add below line to register knockout files as :

```
@Scripts.Render("~/bundles/knockout")
```

Step 5:

Add a new folder name as Contact inside Views, and then add Index.cshtml as new View page. Then add a new Controller name it ContactController.cs inside Controller folder, and add a new Contact.js file under Scripts folder. Refer to below image.



Step 6:

Finally modify the default map route in Route.config to point to Contact controller as:

```
routes.MapRoute(
    name: "Default",
    url: "{controller}/{action}/{id}",
    defaults: new { controller = "Contact", action = "Index", id =
UrlParameter.Optional }
);
```

And also modify the _Layout.cshtml file inside View/Shared as per the Bootstrap Syntax. Below is the modified code:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <title>@ViewBag.Title - Contact manager</title>
    <link href="~/favicon.ico" rel="shortcut icon" type="image/x-icon" />
    <meta name="viewport" content="width=device-width" />
    @Scripts.Render("~/bundles/jquery")
    @Scripts.Render("~/bundles/knockout")
    @Styles.Render("~/Content/css")
    @Scripts.Render("~/bundles/modernizr")
    @RenderSection("scripts", required: false)
</head>
<body>

    <div class="container-narrow">
        <div class="masthead">
            <ul class="nav nav-pills pull-right">

            </ul>
            <h3 class="muted">Contact Manager</h3>
        </div>
        <div id="body" class="container">
            @RenderSection("featured", required: false)
```

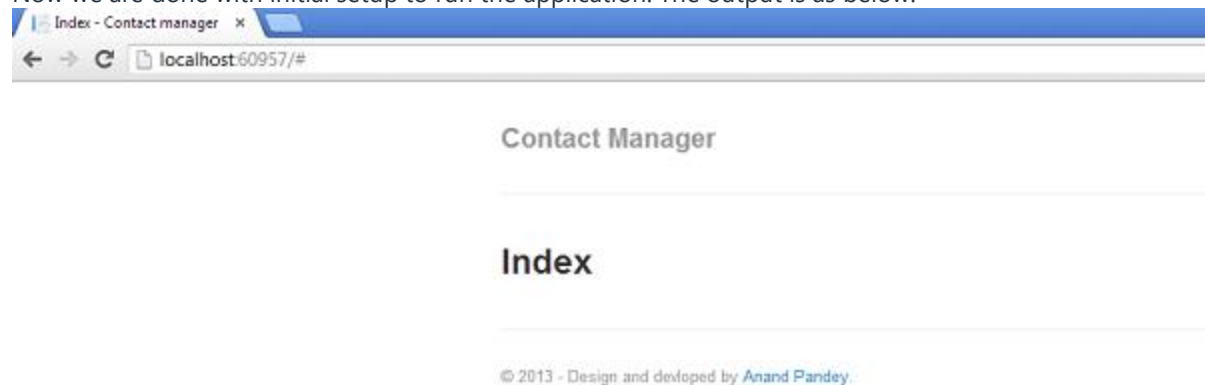
```

        <section>
            @RenderBody()
        </section>
    </div>
    <hr />
    <div id="footer">
        <div class="container">
            <p class="muted credit">&copy; @DateTime.Now.Year - Design and devloped
by <a href="http://www.anandpandey.com">Anand Pandey</a>.</p>
            </div>
        </div>
    </div>
</body>
</html>

```

Step 7:

Now we are done with initial setup to run the application. The output is as below:



We will use this page to display the requirement for Screen 1 i.e. Contact List - View all contacts

Step 8:

First we will create a dummy profile data as array in Contact.js (later we will fetch it from database), and then we will use this data to populate Grid.

```
var DummyProfile = [
  {
    "ProfileId": 1,
    "FirstName": "Anand",
    "LastName": "Pandey",
    "Email": "anand@anandpandey.com"
  },
  {
    "ProfileId": 2,
    "FirstName": "John",
    "LastName": "Cena",
    "Email": "john@cena.com"
  }
]
```

Next, we will create ProfilesViewModel, a viewmodel class which hold Profiles, an array holding an initial collection of DummyProfile data. Note that it's a ko.observableArray, and it is the observable equivalent of a regular array, which means it can automatically trigger UI updates whenever items are added or removed.

And finally we need to activate Knockout using ko.applyBindings().

```
var ProfilesViewModel = function () {
  var self = this;
  var refresh = function () {
    self.Profiles(DummyProfile);
  };

  // Public data properties
  self.Profiles = ko.observableArray([]);
  refresh();
};
ko.applyBindings(new ProfilesViewModel());
```

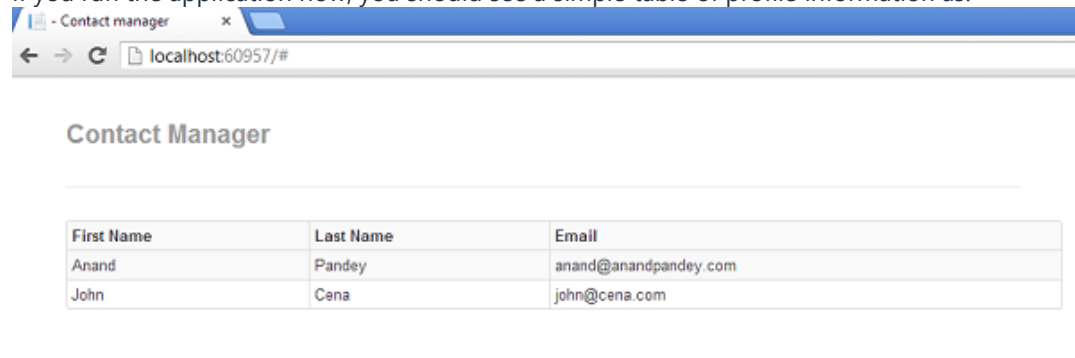
Step 9:

Next we will write code in Index.cshtml page, that's supposed to display the Profile List. We need to use the foreach binding on <tbody> element, so that it will render a copy of its child elements for each entry in the profiles array, and then populate that <tbody> element with some markup to say that you want a table row (<tr>) for each entry.

```
<table class="table table-striped table-bordered table-condensed">
  <tr>
    <th>First Name</th>
    <th>Last Name</th>
    <th>Email</th>
  </tr>
  <tbody data-bind="foreach: Profiles">
    <tr>
      <td data-bind="text: FirstName"></td>
      <td data-bind="text: LastName"></td>
      <td data-bind="text: Email"></td>
    </tr>
  </tbody>
</table>

<script src="~/Scripts/Contact.js"></script>
```

If you run the application now, you should see a simple table of profile information as:



Remember for table style we are using Bootstrap's css class. In above example it is ;

```
<table class="table table-striped table-bordered table-condensed">
```

Step 10:

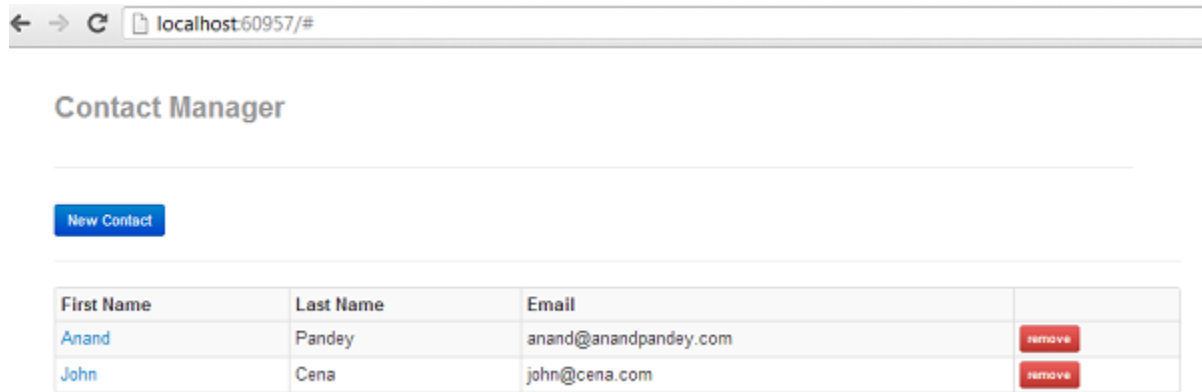
Now we need to add Edit and Remove functionality for each row, and one more button at top to create a new profile. So let us do:

- Add one more <th> and <td> in our table template and bind it's click event with removeProfile function in js.
- Modify First Name row to add link for Edit Profile and then bind its click event with editProfile function.
- Add one button for Create new profile and bind it with click event using createProfile function.

So our final code for Index.cshtml is :

```
<input type="button" class="btn btn-small btn-primary" value="New Contact" data-  
bind="click:$root.createProfile" />  
<hr />  
<table class="table table-striped table-bordered table-condensed">  
  <tr>  
    <th>First Name</th>  
    <th>Last Name</th>  
    <th>Email</th>  
    <th></th>  
  </tr>  
  <tbody data-bind="foreach: Profiles">  
    <tr>  
      <td class="name"><a data-bind="text: FirstName, click:  
$parent.editProfile"></a></td>  
      <td data-bind="text: LastName"></td>  
      <td data-bind="text: Email"></td>  
      <td><button class="btn btn-mini btn-danger" data-bind="click:  
$parent.removeProfile">remove</button></td>  
    </tr>  
  </tbody>  
</table>  
  
<script src="~/Scripts/Contact.js"></script>
```

And output is:



None of the added button and link will work, because we have not written any code for that, so let's fix that in next step.

Step 11:

Add events for createProfile, editProfile and removeProfile in Contact.js

```
self.createProfile = function () {  
    alert("Create a new profile");  
};  
  
self.editProfile = function (profile) {  
    alert("Edit tis profile with profile id as :" + profile.ProfileId);  
};  
  
self.removeProfile = function (profile) {  
    if (confirm("Are you sure you want to delete this profile?")) {  
        self.Profiles.remove(profile);  
    }  
};
```

Now when we run our application and click on Remove button, then it will remove that profile from current array. As we define this array as observable, the UI will be kept in sync with changes to that array. Try it by clicking on Remove button. Edit link and Create Profile button will simply display the alert. So, let us implemented this functionality in next steps:

Step 12:

Next we will add:

- A new Razor View inside Views/Contact as CreateEdit.cshtml, and register it in ContactController.cs class.

```
public ActionResult CreateEdit()
{
    return View();
}
```

- A new js file named as CreateEdit.js inside Scripts folder.
- Modify createProfile and editProfile method of Contact.js, so that it will point to the CreateEdit page.

```
self.createProfile = function () {
    window.location.href = '/Contact/CreateEdit/0';
};

self.editProfile = function (profile) {
    window.location.href = '/Contact/CreateEdit/' + profile.ProfileId;
};
```

Running the application will now work for all the events in Contact List (screen -1). And create and Edit should redirect to CreateEdit page with required parameters.

Step 13:

First we will start with adding Profile Information to this CreateEdit page. For that we need to do:

We need to get profileId from url so, add below two lines on top of the CreateEdit.js page

```
var url = window.location.pathname;
var profileId = url.substring(url.lastIndexOf('/') + 1);
Define a dummy Profile data as array in CreateEdit.js
Collapse | Copy Code
var DummyProfile = [
    {
        "ProfileId": 1,
```

```

        "FirstName": "Anand",
        "LastName": "Pandey",
        "Email": "anand@anandpandey.com"
    },
    {
        "ProfileId": 2,
        "FirstName": "John",
        "LastName": "Cena",
        "Email": "john@cena.com"
    }
]

```

Profile, a simple JavaScript class constructor that stores a profile's FirstName, LastName and Email selection.

```

var Profile = function (profile) {
    var self = this;

    self.ProfileId = ko.observable(profile ? profile.ProfileId : 0);
    self.FirstName = ko.observable(profile ? profile.FirstName : '');
    self.LastName = ko.observable(profile ? profile.LastName : '');
    self.Email = ko.observable(profile ? profile.Email : '');
};

```

ProfileCollection, a viewmodel class that holds profile (a JavaScript object providing Profile data) along with binding for saveProfile and backToProfileList events.

```

var ProfileCollection = function () {
    var self = this;

    //if ProfileId is 0, It means Create new Profile
    if (profileId == 0) {
        self.profile = ko.observable(new Profile());
    }
    else {
        var currentProfile = $.grep(DummyProfile, function (e) { return e.ProfileId ==
profileId; });
    }
};

```

```

        self.profile = ko.observable(new Profile(currentProfile[0]));
    }

    self.backToProfileList = function () { window.location.href = '/contact'; };

    self.saveProfile = function () {
        alert("Date to save is : " + JSON.stringify(ko.toJSON(self.profile())));
    };
};

```

And finally, activate Knockout using `ko.applyBindings()`.

```
ko.applyBindings(new ProfileCollection());
```

Step 14:

Next we will write code in `CreateEdit.cshtml` page, that's supposed to display the Profile information. We need to use the "with" binding for profile data, so that it will render a copy of its child elements for a particular profile, and then assign the appropriate values. The code for `CreateEdit.cshtml` is as below:

```

<table class="table">
    <tr>
        <th colspan="3">Profile Information</th>
    </tr>
    <tr></tr>
    <tbody data-bind='with: profile'>
        <tr>
            <td>
                <input class="input-large" data-bind='value: FirstName'
placeholder="First Name"/>
            </td>
            <td>
                <input class="input-large" data-bind='value: LastName' placeholder="Last
Name"/>
            </td>
            <td>

```

```

        <input class="input-large" data-bind='value: Email' placeholder="Email"
/>
    </td>
</tr>
</tbody>
</table>

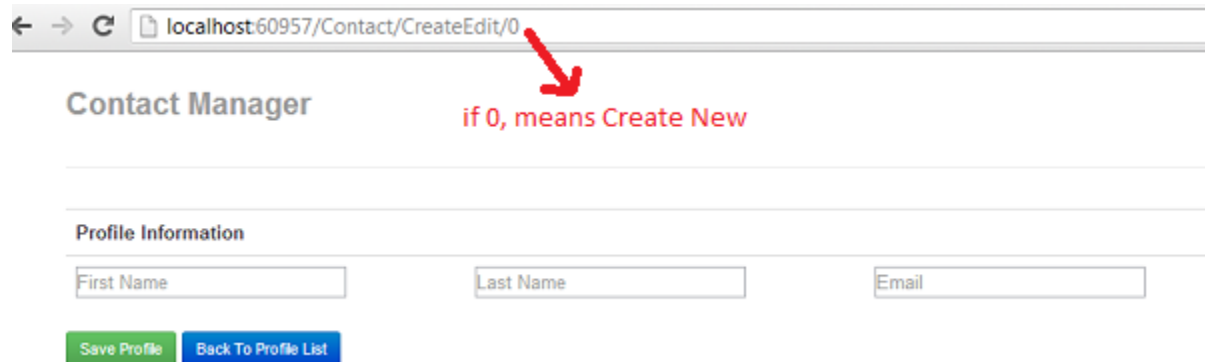
<button class="btn btn-small btn-success" data-bind='click: saveProfile'>Save
Profile</button>
<input class="btn btn-small btn-primary" type="button" value="Back To Profile List" data-
bind="click:$root.backToProfileList" />

<script src="~/Scripts/CreateEdit.js"></script>

```

Running the application will display the below screens :

For create New:



← → ↻

Contact Manager

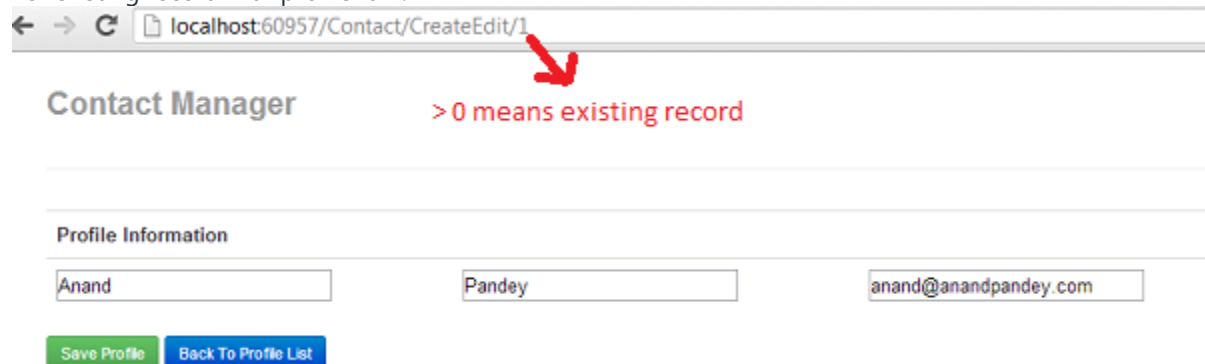
if 0, means Create New

Profile Information

First Name Last Name Email

Save Profile Back To Profile List

For existing record with profile Id 1:



← → ↻

Contact Manager

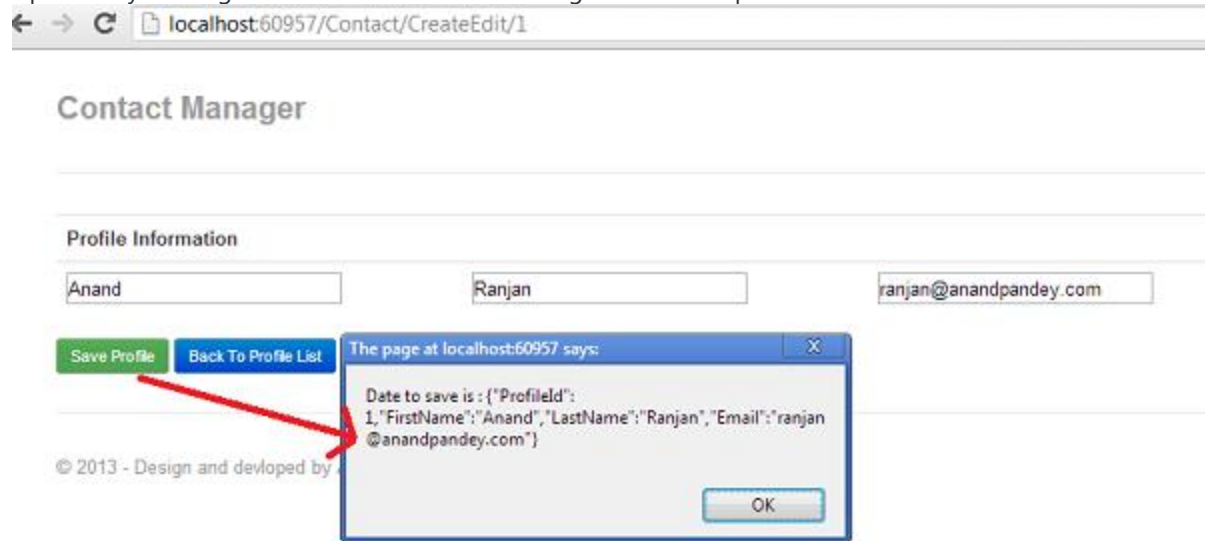
> 0 means existing record

Profile Information

Anand Pandey anand@anandpandey.com

Save Profile Back To Profile List

Update any existing record and click on save with give below output:



As per requirement for this screen, we already have done:

2.1 User should be able to Enter his/her First name, Last Name and Email Address

2.6 Click on save button should save Contact details in Database and user will return back in Contact List page.

2.7 Click on Back to Profile button should return back the user to Contact List page.

In next step we will try to achieve below requirement:

2.2 User should able to add any number of Phone numbers by clicking on Add numbers.

2.3 User should able to remove any phone number

Step 15:

To achieve requirement no. 2.2 and 2.3, we need to do:

Define a dummy PhoneType and PhoneDTO data as an array in CreateEdit.js. phoneTypeData will be used to bind dropdown to define the Phone Type for a particular Phone number.

```
var phoneTypeData = [  
  {  
    "PhoneTypeId": 1,  
    "Name": "Work Phone"  
  },  
  {
```

```

        "PhoneTypeId": 2,
        "Name": "Personal Phone"
    }
];

var PhoneDTO = [
    {
        "PhoneId":1,
        "PhoneTypeId": 1,
        "ProfileId":1,
        "Number": "111-222-3333"
    },
    {
        "PhoneId": 2,
        "PhoneTypeId": 2,
        "ProfileId": 1,
        "Number": "444-555-6666"
    }
];

```

Profile, a simple JavaScript class constructor that stores a PhoneLine information which include its Phone Type i.e. whether its "Work Phone" or "Home Phone" etc. along with phone number.

```

var PhoneLine = function (phone) {
    var self = this;
    self.PhoneId = ko.observable(phone ? phone.PhoneId : 0);
    self.PhoneTypeId = ko.observable(phone ? phone.PhoneTypeId : 0);
    self.Number = ko.observable(phone ? phone.Number : '');
};

```

Modify ProfileCollection viewmodel class to also holds phoneNumbers along with binding for addPhone and removePhone events.

```

var ProfileCollection = function () {
    var self = this;

    //if ProfileId is 0, It means Create new Profile

```

```

    if (profileId == 0) {
        self.profile = ko.observable(new Profile());
        self.phoneNumbers = ko.observableArray([new PhoneLine()]);
    }
    else {
        var currentProfile = $.grep(DummyProfile, function (e) { return e.ProfileId ==
profileId; });
        self.profile = ko.observable(new Profile(currentProfile[0]));
        var currentProfilePhone = $.grep(PhoneDTO, function (e) { return e.ProfileId ==
profileId; });
        self.phoneNumbers = ko.observableArray(ko.utils.arrayMap(currentProfilePhone,
function (phone) {
            return phone;
        }));
    }

    self.addPhone = function () {
        self.phoneNumbers.push(new PhoneLine())
    };

    self.removePhone = function (phone) { self.phoneNumbers.remove(phone) };

    self.backToProfileList = function () { window.location.href = '/contact'; };

    self.saveProfile = function () {
        alert("Date to save is : " + JSON.stringify(ko.toJS(self.profile())));
    };
};

```

Step 16:

Next we will add one more section to add Phone Information in CreateEdit.cshtml page, that's supposed to display the Phone information. As one profile can have multiple phone of different types, So, we will use the "foreach" binding for Phone numbers data, so that it will render a copy of its child elements for a particular profile, and then assign the appropriate values. Add below section in CreateEdit.cshtml just after Profile Information and before Save button.

```

<table class="table">
  <tr>
    <th colspan="3">Phone Information</th>
  </tr>
  <tr></tr>
  <tbody data-bind='foreach: phoneNumbers'>
    <tr>
      <td>
        <select data-bind="options: phoneTypeData, value: PhoneTypeId,
optionsValue: 'PhoneTypeId', optionsText: 'Name', optionsCaption: 'Select Phone
Type...'"></select>
      </td>
      <td>
        <input class="input-large" data-bind='value: Number' placeholder="Number"
/>
      </td>
      <td>
        <a class="btn btn-small btn-danger" href='#' data-bind=' click:
$parent.removePhone'>X</a>
      </td>
    </tr>
  </tbody>
</table>
<p>
<button class="btn btn-small btn-primary" data-bind='click: addPhone'>Add New
Phone</button>
</p>

```

Now the output to add new contact is:

← → ↻ localhost:60957/Contact/CreateEdit/0

Contact Manager

Create New Contact

Profile Information

First Name Last Name Email

Phone Information

Select Phone Type... Number X

And to edit existing Contact is:

← → ↻ localhost:60957/Contact/CreateEdit/1

Contact Manager

Edit Contact with Profile Id = 1

Profile Information

Anand Pandey anand@anandpandey.com

Phone Information

Work Phone 111-222-3333 X

Personal Phone 444-555-6666 X

Select Phone Type... Number X

So now we left only with below requirement:

2.4 User should able to add any number of Addresses by clicking on Add new address.

2.5 User should able to remove any address

Step 17: The requirement no. 2.4 and 2.5 are similar to Phone Information, below is the final code for

CreateEdit.js and CreateEdit.cshtml files:

For CreateEdit.js

```
var url = window.location.pathname;
var profileId = url.substring(url.lastIndexOf('/') + 1);

var DummyProfile = [
    {
        "ProfileId": 1,
        "FirstName": "Anand",
        "LastName": "Pandey",
        "Email": "anand@anandpandey.com"
    },
    {
        "ProfileId": 2,
        "FirstName": "John",
        "LastName": "Cena",
        "Email": "john@cena.com"
    }
];

var PhoneTypeData = [
    {
        "PhoneTypeId": 1,
        "Name": "Work Phone"
    },
    {
        "PhoneTypeId": 2,
        "Name": "Personal Phone"
    }
];

var PhoneDTO = [
    {
        "PhoneId": 1,
        "PhoneTypeId": 1,
```

```
        "ProfileId":1,
        "Number": "111-222-3333"
    },
    {
        "PhoneId": 2,
        "PhoneTypeId": 2,
        "ProfileId": 1,
        "Number": "444-555-6666"
    }
];
```

```
var AddressTypeData = [
    {
        "AddressTypeId": 1,
        "Name": "Shipping Address"
    },
    {
        "AddressTypeId": 2,
        "Name": "Billing Address"
    }
];
```

```
var AddressDTO = [
    {
        "AddressId": 1,
        "AddressTypeId": 1,
        "ProfileId": 1,
        "AddressLine1": "10000 Richmond Avenue",
        "AddressLine2": "Apt # 1000",
        "Country": "USA",
        "State": "Texas",
        "City": "Houston",
        "ZipCode": "70000"
    },
    {
        "AddressId": 2,
```

```

        "AddressTypeId": 2,
        "ProfileId": 1,
        "AddressLine1": "20000 Highway 6",
        "AddressLine2": "Suite # 2000",
        "Country": "USA",
        "State": "Texas",
        "City": "Houston",
        "ZipCode": "80000"
    }
];

var Profile = function (profile) {
    var self = this;

    self.ProfileId = ko.observable(profile ? profile.ProfileId : 0);
    self.FirstName = ko.observable(profile ? profile.FirstName : '');
    self.LastName = ko.observable(profile ? profile.LastName : '');
    self.Email = ko.observable(profile ? profile.Email : '');
    self.PhoneDTO = ko.observableArray(profile ? profile.PhoneDTO : []);
    self.AddressDTO = ko.observableArray(profile ? profile.AddressDTO : []);
};

var PhoneLine = function (phone) {
    var self = this;

    self.PhoneId = ko.observable(phone ? phone.PhoneId : 0);
    self.PhoneTypeId = ko.observable(phone ? phone.PhoneTypeId : 0);
    self.Number = ko.observable(phone ? phone.Number : '');
};

var AddressLine = function (address) {
    var self = this;

    self.AddressId = ko.observable(address ? address.AddressId : 0);
    self.AddressTypeId = ko.observable(address ? address.AddressTypeId : 0);
    self.AddressLine1 = ko.observable(address ? address.AddressLine1 : '');
    self.AddressLine2 = ko.observable(address ? address.AddressLine2 : '');
};

```

```

    self.Country = ko.observable(address ? address.Country : '');
    self.State = ko.observable(address ? address.State : '');
    self.City = ko.observable(address ? address.City : '');
    self.ZipCode = ko.observable(address ? address.ZipCode : '');
};

var ProfileCollection = function () {
    var self = this;

    //if ProfileId is 0, It means Create new Profile
    if (profileId == 0) {
        self.profile = ko.observable(new Profile());
        self.phoneNumbers = ko.observableArray([new Phoneline()]);
        self.addresses = ko.observableArray([new AddressLine()]);
    }
    else {
        //For Profile information
        var currentProfile = $.grep(DummyProfile, function (e) { return e.ProfileId ==
profileId; });
        self.profile = ko.observable(new Profile(currentProfile[0]));
        //For Phone number
        var currentProfilePhone = $.grep(PhoneDTO, function (e) { return e.ProfileId ==
profileId; });
        self.phoneNumbers = ko.observableArray(ko.utils.arrayMap(currentProfilePhone,
function (phone) {
            return phone;
        }));
        //For Address
        var currentProfileAddress = $.grep(AddressDTO, function (e) { return e.ProfileId
== profileId; });
        self.addresses = ko.observableArray(ko.utils.arrayMap(currentProfileAddress,
function (address) {
            return address;
        }));
    }
}

```

```

self.addPhone = function () { self.phoneNumbers.push(new PhoneLine()) };

self.removePhone = function (phone) { self.phoneNumbers.remove(phone) };

self.addAddress = function () { self.addresses.push(new AddressLine()) };

self.removeAddress = function (address) { self.addresses.remove(address) };

self.backToProfileList = function () { window.location.href = '/contact'; };

self.saveProfile = function () {
    self.profile().AddressDTO = self.addresses;
    self.profile().PhoneDTO = self.phoneNumbers;
    alert("Date to save is : " + JSON.stringify(ko.toJS(self.profile())));
};
};

ko.applyBindings(new ProfileCollection());

```

and, for CreateEdit.cshtml

```

<table class="table">
    <tr>
        <th colspan="3">Profile Information</th>
    </tr>
    <tr></tr>
    <tbody data-bind='with: profile'>
        <tr>
            <td>
                <input class="input-large" data-bind='value: FirstName'
placeholder="First Name"/>
            </td>
            <td>
                <input class="input-large" data-bind='value: LastName' placeholder="Last
Name"/>
            </td>

```

```

        <td>
            <input class="input-large" data-bind='value: Email' placeholder="Email"
/>
        </td>
    </tr>
</tbody>
</table>

<table class="table">
    <tr>
        <th colspan="3">Phone Information</th>
    </tr>
    <tr></tr>
    <tbody data-bind='foreach: phoneNumbers'>
        <tr>
            <td>
                <select data-bind="options: PhoneTypeData, value: PhoneTypeId,
optionsValue: 'PhoneTypeId', optionsText: 'Name', optionsCaption: 'Select Phone
Type...'"></select>
            </td>
            <td>
                <input class="input-large" data-bind='value: Number' placeholder="Number"
/>
            </td>
            <td>
                <a class="btn btn-small btn-danger" href='#' data-bind=' click:
$parent.removePhone'>X</a>
            </td>
        </tr>
    </tbody>
</table>
<p>
<button class="btn btn-small btn-primary" data-bind='click: addPhone'>Add New
Phone</button>
</p>
<hr />

```

```

<table class="table">
  <tr><th colspan="5">Address Information</th></tr>
  <tbody data-bind="foreach: addresses">
    <tr>
      <td colspan="5">
        <select data-bind="options: AddressTypeData, value: AddressTypeId,
optionsValue: 'AddressTypeId', optionsText: 'Name', optionsCaption: 'Select Address
Type...'></select>
      </td>
    </tr>
    <tr>
      <td>
        <input class="input-large" data-bind='value: AddressLine1'
placeholder="Address Line1" />
        <p style="padding-top: 5px;"><input class="input-large" data-bind='value:
State' placeholder="State" /></p>
      </td>
      <td>
        <input class="input-large" data-bind=' value: AddressLine2'
placeholder="Address Line2" />
        <p style="padding-top: 5px;"><input class="input-large" data-bind='value:
Country' placeholder="Country" /></p>
      </td>
      <td>
        <input class="input-large" data-bind='value: City' placeholder="City" />
        <p style="padding-top: 5px;"><input class="input-large" data-bind='value:
ZipCode' placeholder="Zip Code" />
        <a class="btn btn-small btn-danger" href="#" data-bind='click:
$root.removeAddress'>X</a></p>
      </td>
    </tr>
  </tbody>
</table>
<p>
<button class="btn btn-small btn-primary" data-bind='click: addAddress'>Add New
Address</button>

```



```

</p>
<hr />
<button class="btn btn-small btn-success" data-bind='click: saveProfile'>Save
Profile</button>
<input class="btn btn-small btn-primary" type="button" value="Back To Profile List" data-
bind="click:$root.backToProfileList" />

<script src="~/Scripts/CreateEdit.js"></script>

```

So finally, Application will display screens as per the requirement:

Screen 1: Contact List - View all contacts

The screenshot shows a web browser at localhost:60957/contact. The page is titled "Contact Manager". Below the title is a "New Contact" button. A table lists two contacts:

First Name	Last Name	Email	
Anand	Pandey	anand@anandpandey.com	remove
John	Cena	john@cena.com	remove

Screen 2: Create New Contact - This screen should display one blank screen to provide functionality as.

The screenshot shows a web browser at localhost:60957/Contact/CreateEdit/0. The page is titled "Contact Manager". The form is divided into sections:

- Profile Information:** Fields for First Name, Last Name, and Email.
- Phone Information:** A dropdown for "Select Phone Type...", a "Number" field, and a red "X" icon. Below is an "Add New Phone" button.
- Address Information:** A dropdown for "Select Address Type...", and fields for Address Line1, Address Line2, City, State, Country, and Zip Code. A red "X" icon is next to the Zip Code field. Below is an "Add New Address" button.

At the bottom of the form are two buttons: "Save Profile" (green) and "Back To Profile List" (blue).

Screen 3: Update Existing Contact - This screen should display screen with selected contact information details.

[←](#) [→](#) [↺](#) [📄](#) localhost:60957/Contact/CreateEdit/1

Contact Manager

Profile Information

<input type="text" value="Anand"/>	<input type="text" value="Pandey"/>	<input type="text" value="anand@anandpandey.com"/>
------------------------------------	-------------------------------------	--

Phone Information

Work Phone <input type="text" value="111-222-3333"/>	<input type="button" value="X"/>
Personal Phone <input type="text" value="444-555-6666"/>	<input type="button" value="X"/>

Address Information

Shipping Address

<input type="text" value="10000 Richmond Avenue"/>	<input type="text" value="Apt # 1000"/>	<input type="text" value="Houston"/>
<input type="text" value="Texas"/>	<input type="text" value="USA"/>	<input type="text" value="70000"/> <input type="button" value="X"/>

Billing Address

<input type="text" value="20000 Highway 6"/>	<input type="text" value="Suite # 2000"/>	<input type="text" value="Houston"/>
<input type="text" value="Texas"/>	<input type="text" value="USA"/>	<input type="text" value="80000"/> <input type="button" value="X"/>

Validation:

We are almost done with designing part of our application .The only thing left now, is to manage validation when the user clicks on “Save” button. Validation is the major requirement and now a day’s most ignorant part for any web application. By proper validation, user can know what data needs to be entered. Next in this article, I am going to discuss KnockoutJS Validation library which can be downloaded using NuGet. Let us check some of the most common validation scenarios, and how to achieve it using knockout validation.

Scenario 1: First Name is a required field in form

```
this.FirstName = ko.observable().extend({ required: true });
```

Scenario 2: Maximum number of character for First Name should not exceed 50 and should not be less than 3 character

```
this.FirstName = ko.observable().extend({ maxLength: 50, minLength:3});
```

Scenario 4: Age is a required field in form, and should be always greater than 18 and less than 100

```
this.Age = ko.observable().extend({ required: true, max: 100, min:18 });
```

Scenario 5: Email is a required field and should be in proper email format

```
this.Email = ko.observable().extend({ required: true, email: true });
```

Scenario 6: Date of Birth is a required field and should be in proper date format

```
this.DateOfBirth = ko.observable().extend({ required: true, date: true });
```

Scenario 7: Price is a required field and should be in proper number or decimal format

```
this.Price = ko.observable().extend({ required: true, number: true });
```

Scenario 8: Phone Number is a required field and should be in proper US format

Note: A valid US phone number has the following format: 1-xdd-xdd-dddd

The "1-" at the beginning of the string is optional and so are the dashes. x is any digit between 2 and 9 while d can be any digit.

```
this.Phone = ko.observable().extend({ required: true, phoneUS: true });
```

Scenario 9: ToDate field must be greater than FromDate field

```
this.ToDate = ko.observable().extend({
    equal: function () { return (val > $('#FromDate').val()) ? val : val + "|" }
});
```

Scenario 10: Phone number should accept only +-()0-9 from users

```
var regex = /\(?([0-9]{3})\)?([ .-]?)([0-9]{3})\2([0-9]{4})/
this.PhoneNumber = ko.observable().extend({ pattern: regex });
```

So, far we have seen different type of validation scenarios and their syntax; now let us implement it in our application. For that first download the library knockout.validation.js using NuGet. Right now our validation script is fully completed and should look like this :

```
var Profile = function (profile) {
    var self = this;
    self.ProfileId = ko.observable(profile ? profile.ProfileId : 0).extend({ required:
true });
    self.FirstName = ko.observable(profile ? profile.FirstName : '').extend({ required:
true, maxLength: 50 });
    self.LastName = ko.observable(profile ? profile.LastName : '').extend({ required:
true, maxLength: 50 });
    self.Email = ko.observable(profile ? profile.Email : '').extend({ required: true,
maxLength: 50, email: true });
    self.PhoneDTO = ko.observableArray(profile ? profile.PhoneDTO : []);
    self.AddressDTO = ko.observableArray(profile ? profile.AddressDTO : []);
};

var PhoneLine = function (phone) {
    var self = this;
    self.PhoneId = ko.observable(phone ? phone.PhoneId : 0);
    self.PhoneTypeId = ko.observable(phone ? phone.PhoneTypeId : undefined).extend({
required: true });
    self.Number = ko.observable(phone ? phone.Number : '').extend({ required: true,
maxLength: 25, phoneUS: true });
};

var AddressLine = function (address) {
    var self = this;
    self.AddressId = ko.observable(address ? address.AddressId : 0);
```

```

    self.AddressTypeId = ko.observable(address ? address.AddressTypeId :
undefined).extend({ required: true });
    self.AddressLine1 = ko.observable(address ? address.AddressLine1 : '').extend({
required: true, maxLength: 100 });
    self.AddressLine2 = ko.observable(address ? address.AddressLine2 : '').extend({
required: true, maxLength: 100 });
    self.Country = ko.observable(address ? address.Country : '').extend({ required: true,
maxLength: 50 });
    self.State = ko.observable(address ? address.State : '').extend({ required: true,
maxLength: 50 });
    self.City = ko.observable(address ? address.City : '').extend({ required: true,
maxLength: 50 });
    self.ZipCode = ko.observable(address ? address.ZipCode : '').extend({ required: true,
maxLength: 15 });
};

```

After validation our final solution looks like below screen after click on "Save" button:

← → ↻ 📄 localhost:60957/Contact/CreateEdit/0

Contact Manager

Profile Information

<input type="text" value="First Name"/>	<small>This field is required.</small>	<input type="text" value="Last Name"/>	<small>This field is required.</small>	<input type="text" value="88888"/>	<small>Please enter a proper email address</small>
---	--	--	--	------------------------------------	--

Phone Information

<input type="text" value="Work Phone"/>	<input type="text" value="111111111111111"/>	<small>Please specify a valid phone number</small>	<input type="button" value="X"/>
---	--	--	----------------------------------

Address Information

<input type="text" value="Select Address Type..."/>		<small>This field is required.</small>
---	--	--

<input type="text" value="Address Line1"/>	<small>This field is required.</small>	<input type="text" value="Address Line2"/>	<small>This field is required.</small>	<input type="text" value="City"/>	<small>This field is required.</small>
<input type="text" value="State"/>	<small>This field is required.</small>	<input type="text" value="Country"/>	<small>This field is required.</small>	<input type="text" value="Zip Code"/>	<small>This field is required.</small>

So far, we talked about achieving our UI without knowing any actual implementation (databases interaction) i.e. UI is created independently by any designer/developer without knowing the actual business logic. !!!That's great!!!

Next, I will talk about how to design database and how to implement business logic using structured layers.

Part 2: Create Database Design (SQL Server 2008 r2): For DBA

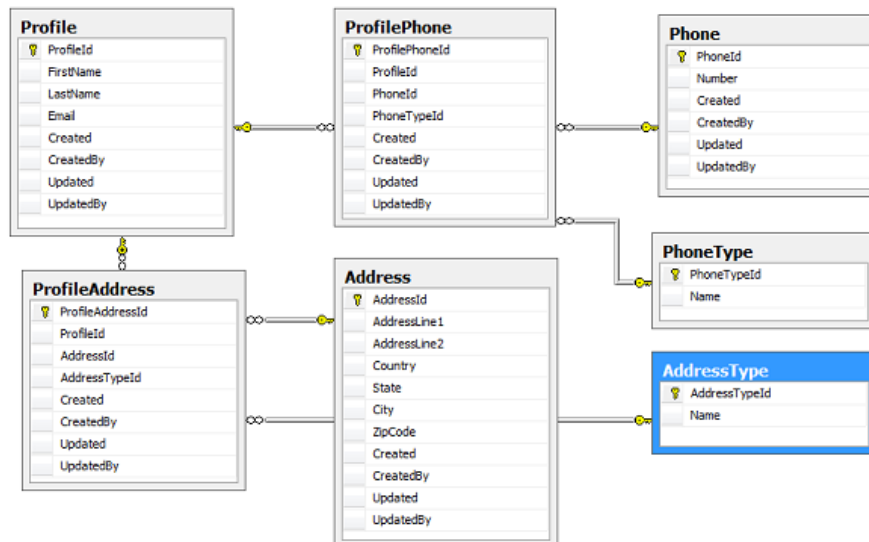
From database design prospective below are the main functionality which need to be achieved:

- A contact can have First Name, Last Name and Email.
- A Contact can have multiple addresses.
- A contact can have multiple phone numbers.

To achieve the contact manager functionality, below database design has been used.

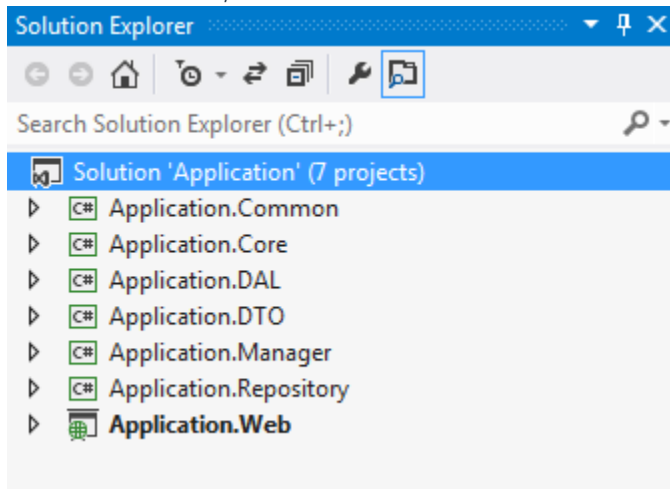
Table Name	Description
Profile	Basic information for any contact Like First Name, Last Name and Email
Address	Address information for any Profile
Phone	Phone information for any profile
AddressType	Master table for diff. Address Types. E.g. "Billing Address", Shipping Address"
PhoneType	Master table for defined diff. Phone Types. E.g. "Work Phone", Home Phone"
ProfileAddress	Link table for Profile, Address and AddressType
ProfilePhone	Link table for Profile, Phone and PhoneType

The relationship between tables are as per below database diagram:



Part 3: Design Logical Layers: For Core Developers

As discussed earlier, our final structure is:



Next we will discuss the overall structure for our application, in terms of the logical group of components into separate layers, which communicates with each other with/without any restrictions and each logic has its own goals. Layers are an architectural style, and it resolves the maintenance and enhancement problems in the long run.

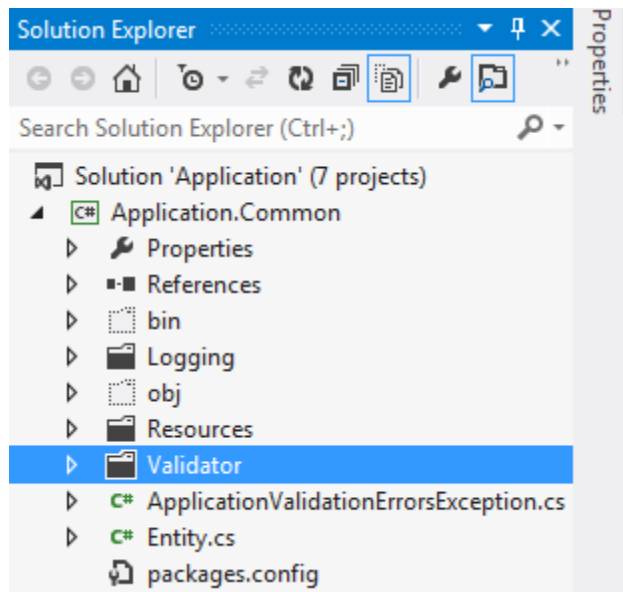
So let us proceed with adding a class library in our solution and name it as Application.Common.

Application.Common :

This is a class library application, with some common functionality and can be used by different logical layers. For e.g. Security, Logging, Tracking, Validation etc. The components defined in this layer can not only reuse by other layers in this solution, but can also be utilize by other applications. To make it easier to change in future, we can use Dependency Injection and abstraction, with minimal change in our application.

For example, In this layer we are going to use, a validator component to validate data entry, and custom Logger to log error or warning.

Below is the screen shot for Solution folder after adding common class library :



Next, we will add a class library in our solution and name it as Application.Core.

Application.Core:

The components of this layer implement the system core functionality and encapsulate all the relevant business logic. Basically, this layer usually contains classes which implement the domain logic within their methods. This layer also defined a Unit of Work contract within the Core Layer so it will be able to comply with the PI principle. The primary goal is to differentiate and clearly separate the behavior of the core domain/business and the infrastructure implementation details such as data access and specific repositories linked to a particular technology such as ORM, or simply data access libraries or even cross-cutting aspects of the architecture. Thus, by isolating the application Core functionality, we will drastically increase the maintainability of our system and we could even replace the lower layers (data access, ORM, and databases) with low impact to the rest of the application.

```

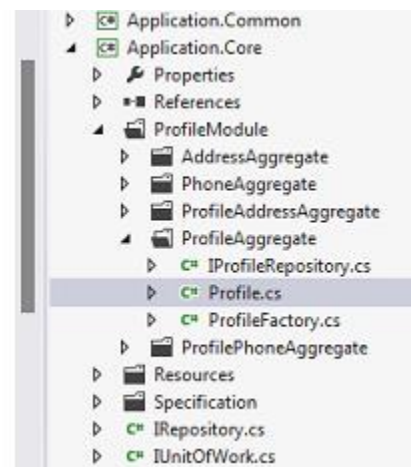
{
    public partial class Profile : Entity, IValidatableObject
    {
        #region Constructor

        public Profile()
        {
            this.ProfileAddresses = new HashSet<ProfileAddress>();
            this.ProfilePhones = new HashSet<ProfilePhone>();
        }

        #endregion Constructor

        #region Properties
        [Key]
        public int ProfileId { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public string Email { get; set; }
    }
}

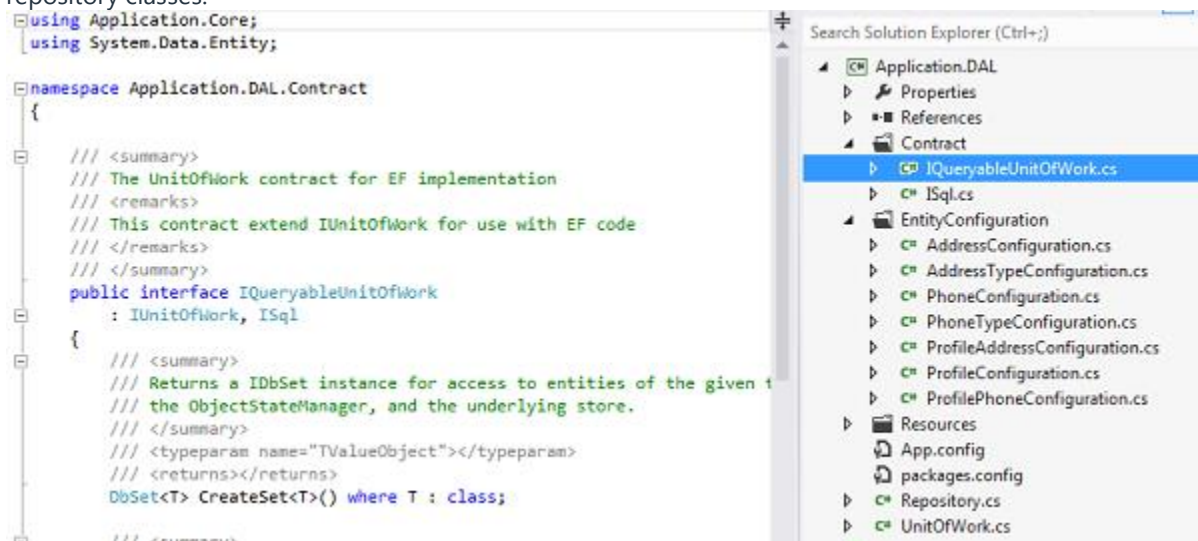
```



Next, we will add a class library in our solution and name it as Application.DAL.

Application.DAL:

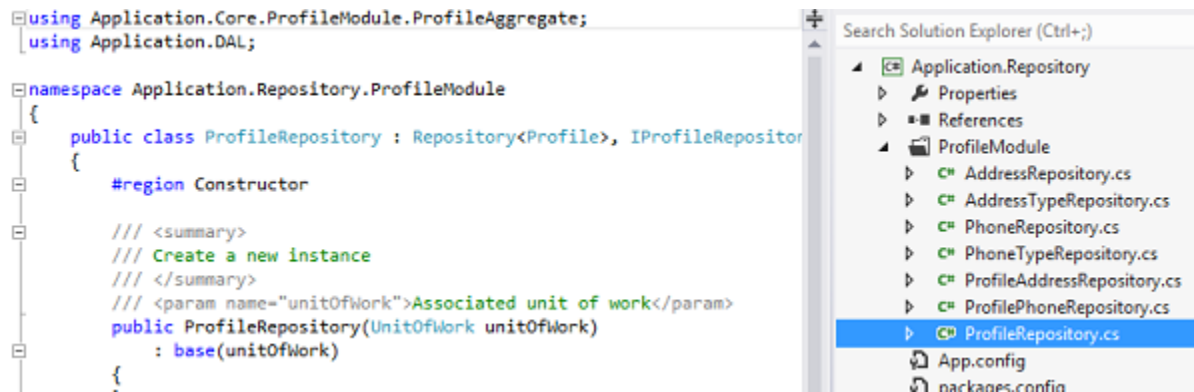
The responsibility of DAL is to and provides data access and persistence operations against the storage database; maintain multiple sessions and connection with multiple database, etc. The Primary goal here is to wrap the EF Context with an Interface/Contract so we can use it from the Manger and Core Layers with no direct dependencies to EF. The data persistence components provide access to the data hosted within the boundaries of our system (e.g., our main database which is within a specific BOUNDED CONTEXT), and also to the data exposed outside the boundaries of our system, such as Web services of external systems. Therefore, it has components like "Repositories" that provide such functionality to access the data hosted within the boundaries of our system, or "Service Agents" that will consume Web Services exposed by other external back-end systems. In addition, this layer will usually have base classes/components with reusable code for all the repository classes.



Next, we will add a class library in our solution and name it as Application. Repository.

Application.Repository:

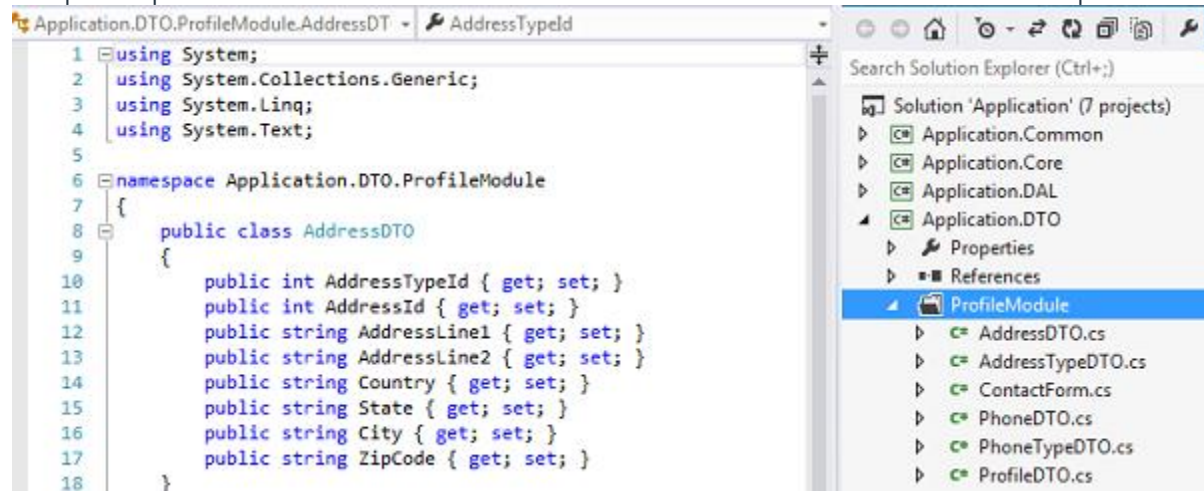
This is a Class Library and can be accessible only by Application.Manager. For each main root ENTITY in our domain, we need to create one repository. Basically, Repositories are classes/components that encapsulate the logic required to access the application data sources. Therefore, they centralize common data access functionality so the application can have a better maintainability and de-coupling between technology and logic owned by the "Manager" and "Core" layers.



Next, we will add a class library in our solution and name it as Application. DTO.

Application.DTO:

Again this is a class library, which contains different container classes that exposes properties but no methods and communicate between Presentation Layer (Application.Web) and Service Layer (Application.Manager). A Data Transfer Object is an object that is used to encapsulate data, and send it from one subsystem of an application to another. Here we are going to use DTOs by the Manager layer to transfer data between itself and the UI layer. The main benefit here is that it reduces the amount of data that needs to be sent across the wire in distributed applications. They also make great models in the MVC pattern. We can also use DTO's to encapsulate parameters for method calls. This can be useful if a method takes more than 4 or 5 parameters.



Next, we will add a class library in our solution and name it as Application. Manager.

Application.Manager :

This is a Class Library and can be accessible only by Application.Web. For each module we need to declare one Manager. The primary responsibilities of Manager are to accept request from UI/Web layer, then communicate with required repositories and manipulate data based on condition then return back the response. This layer is intermediate between UI and Repositories.

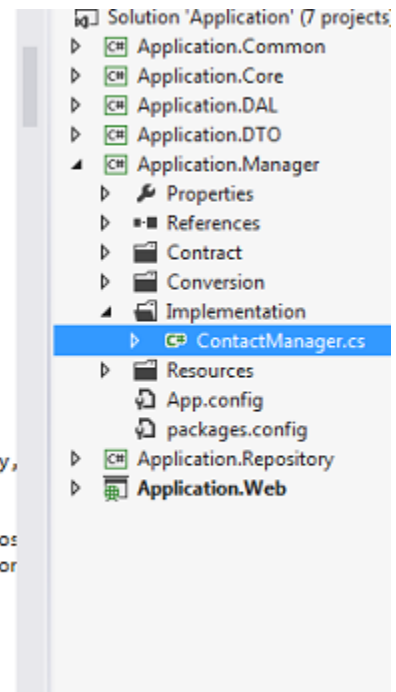
```
public class ContactManager : IContactManager
{
    #region Global Declaration

    AddressRepository _addressRepository;
    AddressTypeRepository _addressTypeRepository;
    PhoneRepository _phoneRepository;
    PhoneTypeRepository _phoneTypeRepository;
    ProfileAddressRepository _profileAddressRepository;
    ProfilePhoneRepository _profilePhoneRepository;
    ProfileRepository _profileRepository;

    #endregion Global Declaration

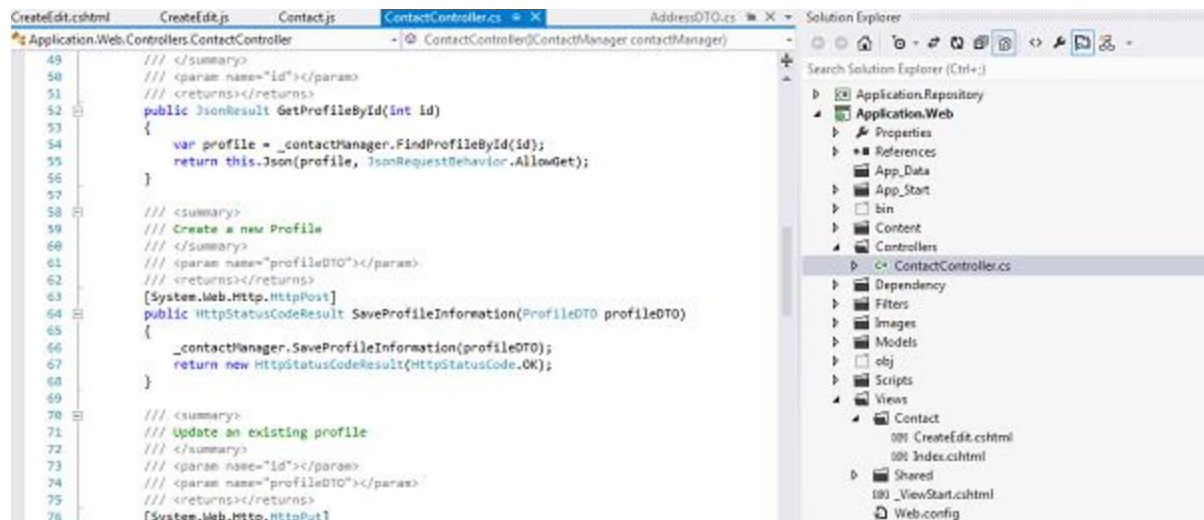
    #region Constructor

    public ContactManager(AddressRepository addressRepository,
        AddressTypeRepository addressTypeRepository,
        PhoneRepository phoneRepository,
        PhoneTypeRepository phoneTypeRepository,
        ProfileAddressRepository profileAddressRepository,
        ProfilePhoneRepository profilePhoneRepository,
        ProfileRepository profileRepository)
    {
        if (addressRepository == null)
            throw new ArgumentNullException("addressRepository");
    }
}
```



Application.Web:

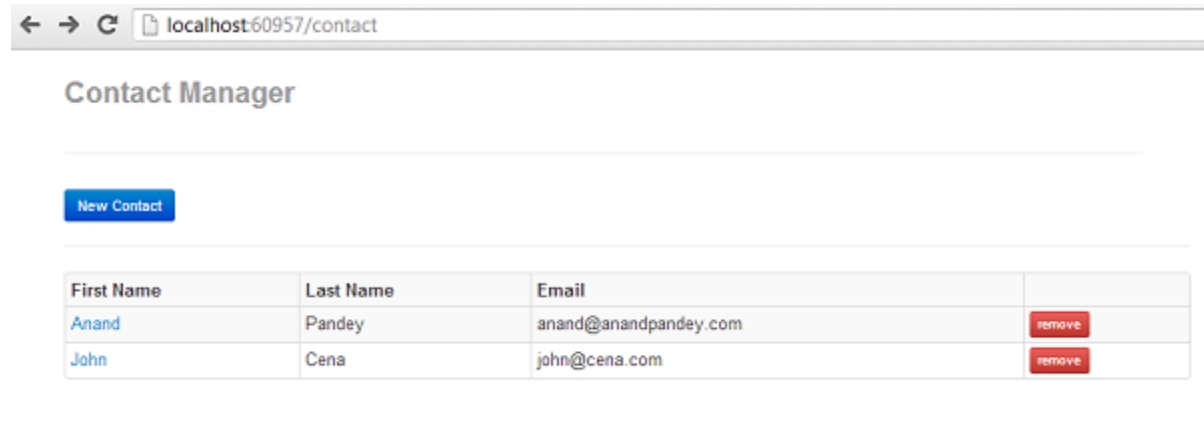
In previous article, we have already implemented this layer using Javascript dummy data. This is independent ASP.NET MVC web application, and contains only User Interface components, like html, .aspx, cshtml, MVC etc. It can also be any windows application. It communicates with some methods from manager layer, then evaluate results and choose whether to show an error or page1 or page2 etc. etc. This layer use javascript to load a model for the presentation, but the data is processed in the server through an ajax request, so the server only manage the business logic and the javascript manages the presentation logic.



To better understand how layers are communicating with each other, let us recall the initial requirement:

Screen 1: Contact List - View all contacts

- 1.1 This screen should display all the contacts available in Database.
- 1.2 User should be able to delete any contact.
- 1.3 User should be able to edit any contact details.
- 1.4 User should be able to create a new contact.



To populate the grid data, on page load, we call GetAllProfiles() method of ContactController. This method returns all Profiles exist in database as an JSON object, then we bind it with JavaScript object self.Profiles. Below is the code to call GetAllProfiles() from contact.js:

```

var ProfilesViewModel = function () {
    var self = this;
    var url = "/contact/GetAllProfiles";

```

```

var refresh = function () {
    $.getJSON(url, {}, function (data) {
        self.Profiles(data);
    });
};

```

On Remove button click we call DeleteProfile () method of ContactController. This method returns removes that profile from database. Below is the code to call DeleteProfile() from contact.js:

```

self.removeProfile = function (profile) {
    if (confirm("Are you sure you want to delete this profile?")) {
        var id = profile.ProfileId;
        waitingDialog({});
        $.ajax({
            type: 'DELETE', url: 'Contact/DeleteProfile/' + id,
            success: function () { self.Profiles.remove(profile); },
            error: function (err) {
                var error = JSON.parse(err.responseText);
                $("<div></div>").html(error.Message).dialog({ modal: true,
                    title: "Error", buttons: { "Ok":
                        function () { $(this).dialog("close"); } } }).show();
            },
            complete: function () { closeWaitingDialog(); }
        });
    }
};

```

For both Create New button and Edit link, we only redirect to CreateEdit page with id as 0 for Create new and for edit the profile id of selected row. Below is the code for createProfile and editProfile from contact.js:

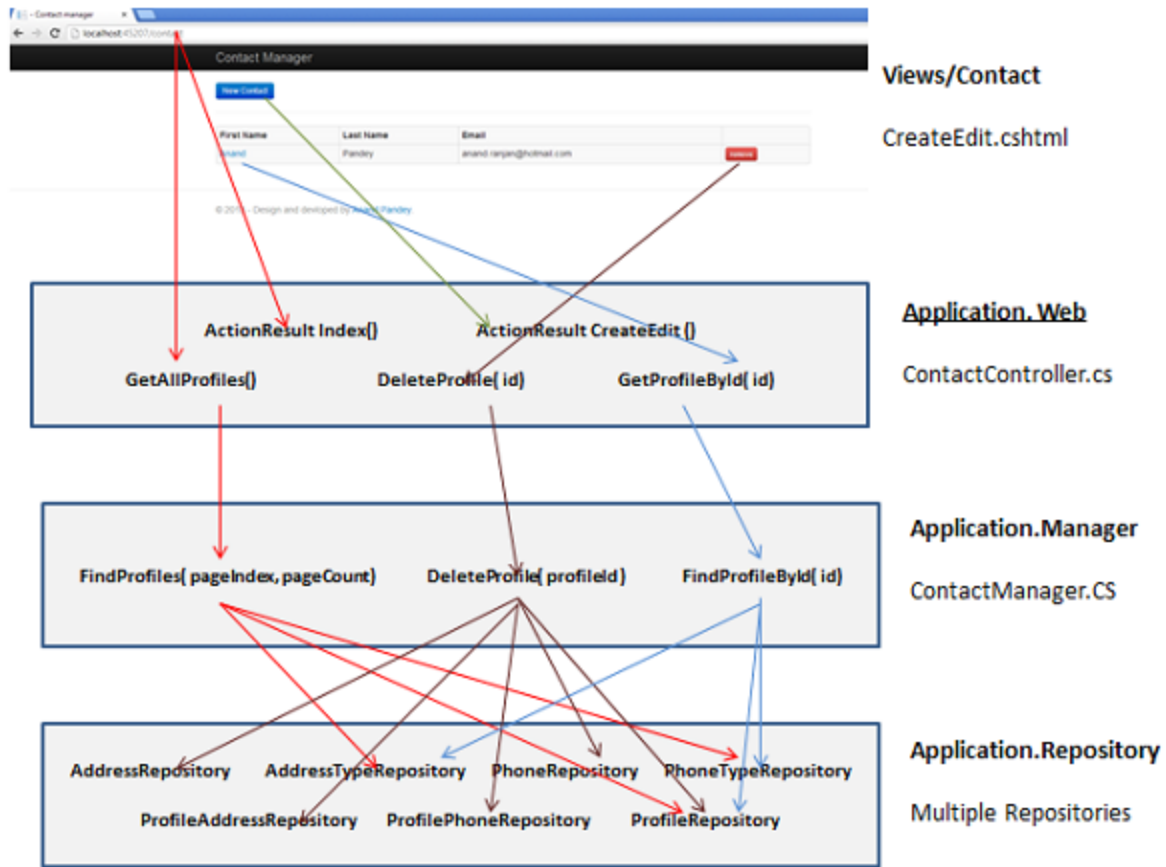
```

self.createProfile = function () {
    window.location.href = '/Contact/CreateEdit/0';
};

self.editProfile = function (profile) {
    window.location.href = '/Contact/CreateEdit/' + profile.ProfileId;
};

```

Below is the diagram representation of communication between three major layers:



Screen 2: Create New Contact

This screen should display one blank screen to provide functionalities as.

- 2.1 User should be able to Enter his/her First name, Last Name and Email Address.
- 2.2 User should be able to add any number of Phone numbers by clicking on Add numbers.
- 2.3 User should be able to remove any phone number.
- 2.4 User should be able to add any number of Addresses by clicking on Add new address.
- 2.5 User should be able to remove any address.
- 2.6 Click on save button should save Contact details in Database and user will return back in Contact List page.

2.7 Click on Back to Profile button should return back the user to Contact List page.

The screenshot shows a web browser window with the address bar displaying 'localhost:60957/Contact/CreateEdit/0'. The page title is 'Contact Manager'. The form is divided into three main sections: Profile Information, Phone Information, and Address Information. The Profile Information section has three input fields: First Name, Last Name, and Email. The Phone Information section has a dropdown for 'Select Phone Type...', a text input for 'Number', and a red 'X' delete button. Below this is a blue 'Add New Phone' button. The Address Information section has a dropdown for 'Select Address Type...', and a grid of input fields for Address Line1, Address Line2, City, State, Country, and Zip Code, with a red 'X' delete button. Below this is a blue 'Add New Address' button. At the bottom of the form are two buttons: a green 'Save Profile' button and a blue 'Back To Profile List' button.

Contact Manager

Profile Information

First Name Last Name Email

Phone Information

Select Phone Type... Number X

Add New Phone

Address Information

Select Address Type...

Address Line1 Address Line2 City
State Country Zip Code X

Add New Address

Save Profile Back To Profile List

Screen 3: Update Existing Contact

This screen should display screen with selected contact information details.

3.1 User should be able to modify his/her First name, Last Name and Email Address.

3.2 User should be able to modify /delete/Add any number of Phone numbers by clicking on Add numbers or delete link.

3.3 User should be able to modify /delete/Add any number of Addresses by clicking on Add new address or delete link.

3.4 Click on save button should update Contact details in Database and user will return back in Contact List page.

3.5 Click on Back to Profile button should return back the user to Contact List page.

← → ↻
localhost:60957/Contact/CreateEdit/1

Contact Manager

Profile Information

<input type="text" value="Anand"/>	<input type="text" value="Pandey"/>	<input type="text" value="anand@anandpandey.com"/>
------------------------------------	-------------------------------------	--

Phone Information

Work Phone	<input type="text" value="111-222-3333"/>	<input type="button" value="X"/>
Personal Phone	<input type="text" value="444-555-6666"/>	<input type="button" value="X"/>

Address Information

Shipping Address

<input type="text" value="10000 Richmond Avenue"/>	<input type="text" value="Apt # 1000"/>	<input type="text" value="Houston"/>
<input type="text" value="Texas"/>	<input type="text" value="USA"/>	<input type="text" value="70000"/> <input type="button" value="X"/>

Billing Address

<input type="text" value="20000 Highway 6"/>	<input type="text" value="Suite # 2000"/>	<input type="text" value="Houston"/>
<input type="text" value="Texas"/>	<input type="text" value="USA"/>	<input type="text" value="80000"/> <input type="button" value="X"/>

As discussed in previous implementation, for both "Create new" and "Edit existing" requirement we are using single page as CreateEdit.cshhtml, by identifying the URL value for profileId i.e. if profileId in URL is 0, then it is request for creating a new profile, and if it is some value, the request is for edit existing profile. Below is the implementation details:

In any case (Create or Edit), on page load we need to initialize data for PhoneType and AddressType. For that we have one method in ContactController as InitializePageData(). Below is the code in CreateEdit.js to initialize both arrays:

```
var AddressTypeData;
```



```

var PhoneTypeData;

$.ajax({
    url: urlContact + '/InitializePageData',
    async: false,
    dataType: 'json',
    success: function (json) {
        AddressTypeData = json.lstAddressTypeDTO;
        PhoneTypeData = json.lstPhoneTypeDTO;
    }
});

```

Next, For Edit profile we need to get the profile data, for that we have GetProfileById() method in our ContactController. We modify our existing CreateEdit.js code as:

```

$.ajax({
    url: urlContact + '/GetProfileById/' + profileId,
    async: false,
    dataType: 'json',
    success: function (json) {
        self.profile = ko.observable(new Profile(json));
        self.phoneNumbers = ko.observableArray(ko.utils.arrayMap(json.PhoneDTO, function
(phone) {
            return phone;
        }));
        self.addresses = ko.observableArray(ko.utils.arrayMap(json.AddressDTO, function
(address) {
            return address;
        }));
    }
});

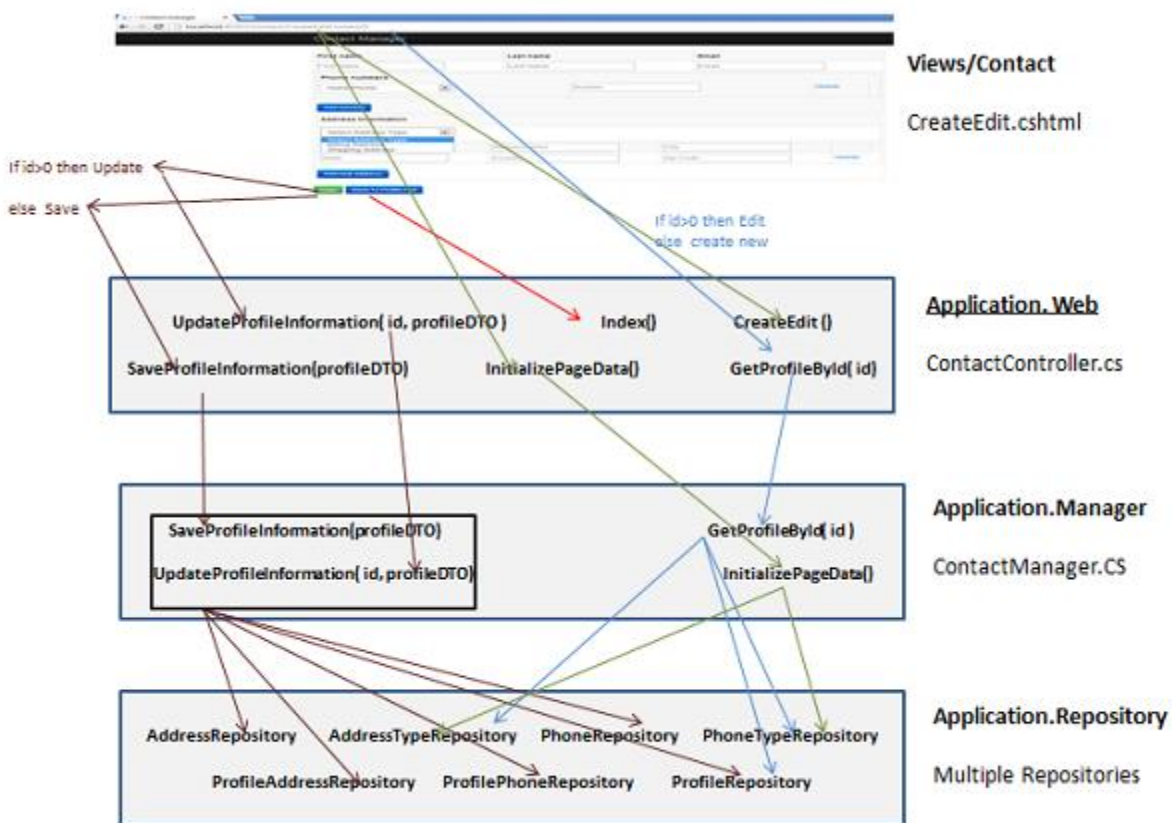
```

Finally, For Save data we have two methods in database. If it's Create new the we will call SaveProfileInfrmation() method of ContactController else we will call UpdateProfileInformation() method. We modify our existing CreateEdit.js code as:

```

$.ajax({
    type: (self.profile().ProfileId > 0 ? 'PUT' : 'POST'),
    cache: false,
    dataType: 'json',
    url: urlContact + (self.profile().ProfileId > 0 ? '/UpdateProfileInformation?id=' +
        self.profile().ProfileId : '/SaveProfileInformation'),
    data: JSON.stringify(ko.toJS(self.profile())),
    contentType: 'application/json; charset=utf-8',
    async: false,
    success: function (data) {
        window.location.href = '/contact';
    },
    error: function (err) {
        var err = JSON.parse(err.responseText);
        var errors = "";
        for (var key in err) {
            if (err.hasOwnProperty(key)) {
                errors += key.replace("profile.", "") + " : " + err[key];
            }
        }
        $("<div></div>").html(errors).dialog({ modal: true,
            title: JSON.parse(err.responseText).Message, buttons: { "Ok":
                function () { $(this).dialog("close"); } } }).show();
    },
    complete: function () {
    }
});

```



Conclusion

That's it!!! Hope you enjoy this article. I am not an expert, and also I have not followed the entire industry standard at the time of writing this article. And that, in a nutshell, is about all I know to get started designing ASP.NET MVC 4 application. I hope you enjoyed this tutorial and learned something.

All comment/Vote are more than welcome.... , If you have any questions feel free to ask; I will be glad to talk more in the comments. Thanks for your time!

How to use code

Download Database script file here:

[Application_DB.sql](#)

It contains SQL Script to create database and master table entry for table PhoneType and AddressType.

To run this application your Visual Studio setting should be enable for "Allow NuGet to download missing packages during build". Or else refer to below link:

<http://docs.nuget.org/docs/workflows/using-nuget-without-committing-packages>

And finally, modify the connection string under Application.Web project.

References

- <http://knockoutjs.com/>
- <https://github.com/ericmbarnard/Knockout-Validation/wiki/Configuration>
- <http://twitter.github.com/bootstrap/>
- <http://docs.castleproject.org/Windsor.MainPage.ashx>
- <http://microsoftnlayerapp.codeplex.com/>
- <http://msdn.microsoft.com/en-us/library/ff921348.aspx>

Last edited Jan 17, 2013 at 10:41 PM by [anandranjanpandey](#), version 5

	Contact Manager app using
CURRENT	ASP.NET MVC 4
DATE	Thu Jan 17, 2013 at 7:00 AM
STATUS	Stable