**ASP.NET MVC Web API**

ASP.NET Web API is a framework that makes it easy to build HTTP services that reach a broad range of clients, including browsers and mobile devices. ASP.NET Web API is an ideal platform for building RESTful applications on the .NET Framework.

When you're building APIs on the Web, there are several ways you can build APIs on the Web. These include HTTP/RPC, and what this means is using HTTP in Remote Procedure Call to call into things, like Methods, across the Web.

The verbs themselves are included in the APIs, like Get Customers, Insert Invoice, Delete Customer, and that each of these endpoints end up being a separate URI.

1. Create a new project (ASP.NET Web Application .Net Framework)

Choose Empty🡪(check) Web API under “add folders and core references”

1. Add a model class (“Employee.cs” )

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

namespace WebAPIDemo.Models {

public class Employee{

public int ID { get; set; }

public string Name { get; set; }

public DateTime JoiningDate { get; set; }

public int Age { get; set; }

}

}

1. Add controller (Web API Controller – Empty). You will see some default codes:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web.Http;

using WebAPIDemo.Models;

namespace WebAPIDemo.Controllers{

public class EmployeesController : ApiController{

Employee[] employees = new Employee[]{

new Employee { ID = 1, Name = "Mark", JoiningDate =

DateTime.Parse(DateTime.Today.ToString()), Age = 30 },

new Employee { ID = 2, Name = "Allan", JoiningDate =

DateTime.Parse(DateTime.Today.ToString()), Age = 35 },

new Employee { ID = 3, Name = "Johny", JoiningDate =

DateTime.Parse(DateTime.Today.ToString()), Age = 21 }

};

public IEnumerable<Employee> GetAllEmployees(){

return employees;

}

public IHttpActionResult GetEmployee(int id){

var employee = employees.FirstOrDefault((p) => p.ID == id);

if (employee == null){

return NotFound();

}

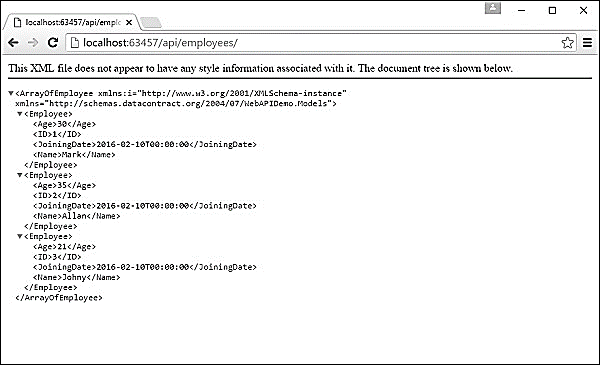
return Ok(employee);

}

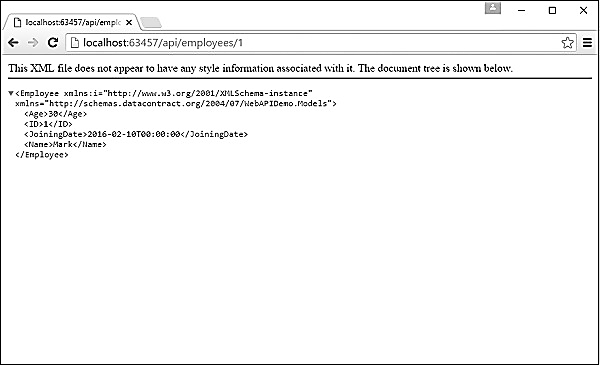
}

}

1. Run this application and specify /api/employees/ at the end of the URL and press ‘Enter’. You will see the following output.



1. Let us specify the following URL http://localhost:63457/api/employees/1 and you will see the following output.



**MVC Web API with CRUD Operations**

1. Create a new project🡪aspnet web application(.net framework)🡪MVC Template (check-MVC & Web API)
2. Create a database, table, fields and add some records. Connect your Database Server to VS Server Explorer.
3. Create a model class

namespace MVCwithWebAPI.Models

{

public class Employee

{

public string Id { get; set; }

public string Name { get; set; }

public string Address { get; set; }

public string Gender { get; set; }

public string Company { get; set; }

public string Designation { get; set; }

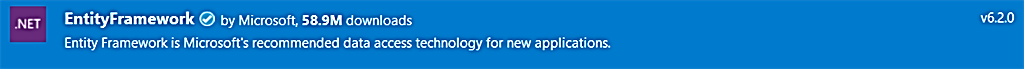
}

}

1. Create a new controller (Web API with read/write actions)
2. We can create a “DbContext” class for database connectivity.

The class that derives DbContext is called context class in entity framework. DbContext is an important class in Entity Framework API. It is a bridge between domain or entity classes and the database. DbContext is the primary class that is responsible for interacting with the database.

Before creating our DbContext class, we must install “EntityFramework” NuGet package in our project.



1. We can create “SqlDbContext” class and derives DbContext class insides this class.

**SqlDbContext.cs**

using System.Data.Entity;

namespace MVCwithWebAPI.Models

{

public class SqlDbContext : DbContext

{

public SqlDbContext() : base("name=SqlConn")

{

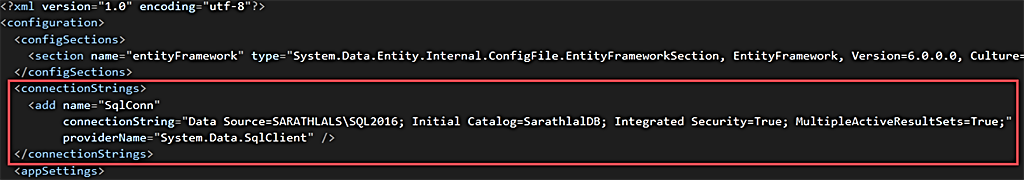
}

public DbSet<Employee> Employees { get; set; }

}

}

Please note, we have used a connection “SqlConn” in above DbContext class. Hence, we can create the connection string in Web.Config file.



1. Create an interface

We are following the repository pattern in this application. We can create a “IEmployeeRepository” interface and define all the functions there.

**IEmployeeRepository.cs**

using System.Collections.Generic;

using System.Threading.Tasks;

namespace MVCwithWebAPI.Models

{

public interface IEmployeeRepository

{

Task Add(Employee employee);

Task Update(Employee employee);

Task Delete(string id);

Task<Employee> GetEmployee(string id);

Task<IEnumerable<Employee>> GetEmployees();

}

}

1. We can implement the exact logic for CRUD actions in “EmployeeRepository” class. We will implement IEmployeeRepository interface in this class.

**EmployeeRepository.cs**

using System;

using System.Collections.Generic;

using System.Data.Entity;

using System.Linq;

using System.Threading.Tasks;

using System.Web;

namespace MVCwithWebAPI.Models

{

public class EmployeeRepository : IEmployeeRepository

{

private readonly SqlDbContext db = new SqlDbContext();

public async Task Add(Employee employee)

{

employee.Id = Guid.NewGuid().ToString();

db.Employees.Add(employee);

try

{

await db.SaveChangesAsync();

}

catch

{

throw;

}

}

public async Task<Employee> GetEmployee(string id)

{

try

{

Employee employee = await db.Employees.FindAsync(id);

if (employee == null)

{

return null;

}

return employee;

}

catch

{

throw;

}

}

public async Task<IEnumerable<Employee>> GetEmployees()

{

try

{

var employees = await db.Employees.ToListAsync();

return employees.AsQueryable();

}

catch

{

throw;

}

}

public async Task Update(Employee employee)

{

try

{

db.Entry(employee).State = EntityState.Modified;

await db.SaveChangesAsync();

}

catch

{

throw;

}

}

public async Task Delete(string id)

{

try

{

Employee employee = await db.Employees.FindAsync(id);

db.Employees.Remove(employee);

await db.SaveChangesAsync();

}

catch

{

throw;

}

}

private bool EmployeeExists(string id)

{

return db.Employees.Count(e => e.Id == id) > 0;

}

}

}

1. Create API controller

**EmployeesApiController.cs**

using MVCwithWebAPI.Models;

using System.Collections.Generic;

using System.Threading.Tasks;

using System.Web.Http;

namespace MVCwithWebAPI.Controllers

{

public class EmployeesApiController : ApiController

{

private readonly IEmployeeRepository \_iEmployeeRepository = new EmployeeRepository();

[HttpGet]

[Route("api/Employees/Get")]

public async Task<IEnumerable<Employee>> Get()

{

return await \_iEmployeeRepository.GetEmployees();

}

[HttpPost]

[Route("api/Employees/Create")]

public async Task CreateAsync([FromBody]Employee employee)

{

if (ModelState.IsValid)

{

await \_iEmployeeRepository.Add(employee);

}

}

[HttpGet]

[Route("api/Employees/Details/{id}")]

public async Task<Employee> Details(string id)

{

var result = await \_iEmployeeRepository.GetEmployee(id);

return result;

}

[HttpPut]

[Route("api/Employees/Edit")]

public async Task EditAsync([FromBody]Employee employee)

{

if (ModelState.IsValid)

{

await \_iEmployeeRepository.Update(employee);

}

}

[HttpDelete]

[Route("api/Employees/Delete/{id}")]

public async Task DeleteConfirmedAsync(string id)

{

await \_iEmployeeRepository.Delete(id);

}

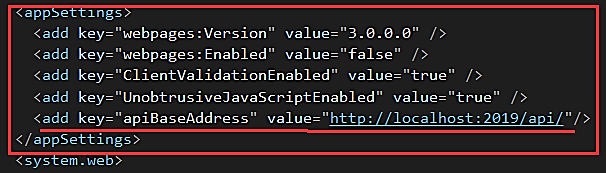
}

}

All the CRUD actions are derived in this API class. We have created an instance for EmployeeRepository class and with the help of this instance, we have accessed all the methods from EmployeeRepository class in our API class.

1. Please note down the URL of the application. We need to add this base URL in our Web.Config file because we will use this base URL in MVC controllers.

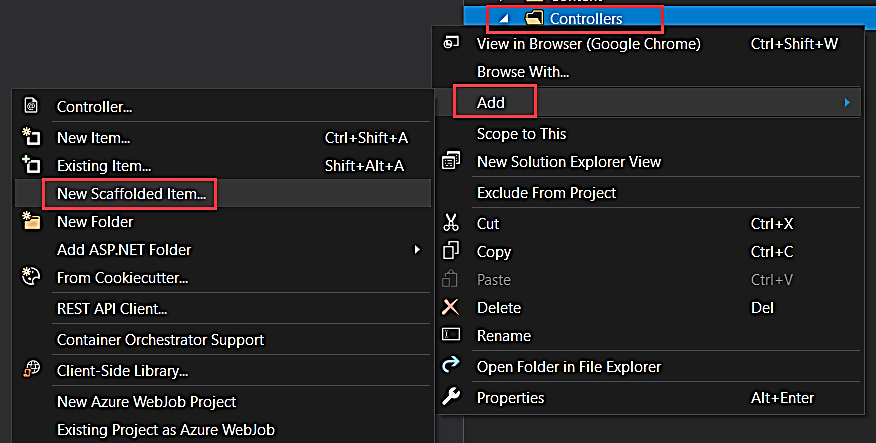
We can create a new key-value pair in Web.Config file under “appSettings” section.



1. Create the MVC Controller that will consume the web api

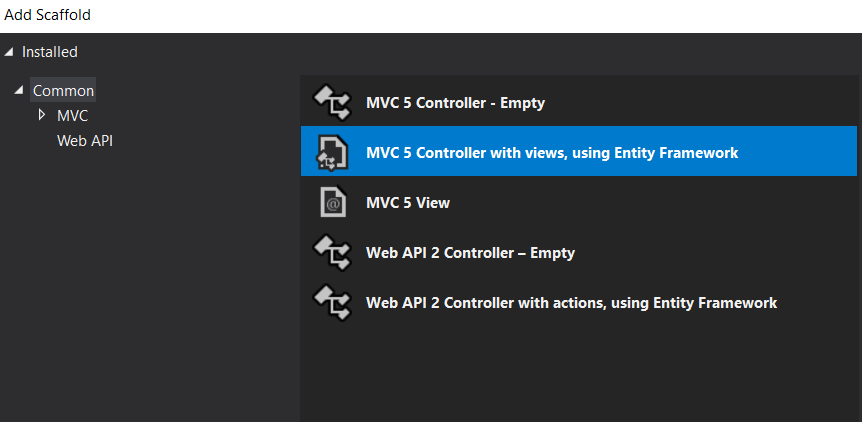
**Create controllers using scaffolding**

We can create the MVC Controller using scaffolding template so that we will get all the views for CRUD actions easily. We will modify these views later.

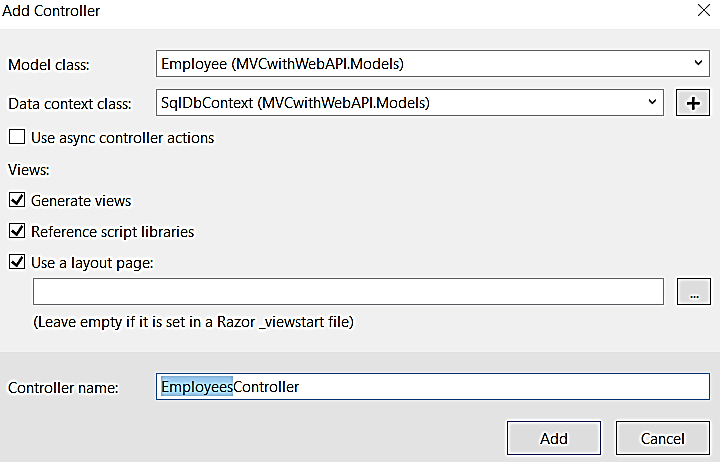


We can right-click the “Controller” folder and click “Add” and choose “New Scaffolded Item” to create a new MVC controller.

You can choose, “MVC 5 Controller with views, using Entity Framework” option. This will create all views for CRUD operations.



We can choose the model class, data context class and give controller name as well. Please note, the system will automatically give a name for a controller. If needed, you can change it.



You can click the “Add” button to proceed further. After a few moments, our Employees controller is created with all views. You can see the views under “Views” -> “Employees” folder.

1. Create MVC controller actions

Let us modify the “EmployeesController” class now. Please note, by default MVC controller does not create actions as async. We have converted all these actions asynchronously. You can copy the below code and paste it inside the controller class.

EmployeesController.cs

using MVCwithWebAPI.Models;

using System;

using System.Collections.Generic;

using System.Configuration;

using System.Linq;

using System.Net;

using System.Net.Http;

using System.Threading.Tasks;

using System.Web.Mvc;

namespace MVCwithWebAPI.Controllers

{

public class EmployeesController : Controller

{

readonly string apiBaseAddress = ConfigurationManager.AppSettings["apiBaseAddress"];

public async Task<ActionResult> Index()

{

IEnumerable<Employee> employees = null;

using (var client = new HttpClient())

{

client.BaseAddress = new Uri(apiBaseAddress);

var result = await client.GetAsync("employees/get");

if (result.IsSuccessStatusCode)

{

employees = await result.Content.ReadAsAsync<IList<Employee>>();

}

else

{

employees = Enumerable.Empty<Employee>();

ModelState.AddModelError(string.Empty, "Server error try after some time.");

}

}

return View(employees);

}

public async Task<ActionResult> Details(string id)

{

if (id == null)

{

return new HttpStatusCodeResult(HttpStatusCode.BadRequest);

}

Employee employee = null;

using (var client = new HttpClient())

{

client.BaseAddress = new Uri(apiBaseAddress);

var result = await client.GetAsync($"employees/details/{id}");

if (result.IsSuccessStatusCode)

{

employee = await result.Content.ReadAsAsync<Employee>();

}

else

{

ModelState.AddModelError(string.Empty, "Server error try after some time.");

}

}

if (employee == null)

{

return HttpNotFound();

}

return View(employee);

}

public ActionResult Create()

{

return View();

}

[HttpPost]

[ValidateAntiForgeryToken]

public async Task<ActionResult> Create([Bind(Include = "Name,Address,Gender,Company,Designation")] Employee employee)

{

if (ModelState.IsValid)

{

using (var client = new HttpClient())

{

client.BaseAddress = new Uri(apiBaseAddress);

var response = await client.PostAsJsonAsync("employees/Create", employee);

if (response.IsSuccessStatusCode)

{

return RedirectToAction("Index");

}

else

{

ModelState.AddModelError(string.Empty, "Server error try after some time.");

}

}

}

return View(employee);

}

public async Task<ActionResult> Edit(string id)

{

if (id == null)

{

return new HttpStatusCodeResult(HttpStatusCode.BadRequest);

}

Employee employee = null;

using (var client = new HttpClient())

{

client.BaseAddress = new Uri(apiBaseAddress);

var result = await client.GetAsync($"employees/details/{id}");

if (result.IsSuccessStatusCode)

{

employee = await result.Content.ReadAsAsync<Employee>();

}

else

{

ModelState.AddModelError(string.Empty, "Server error try after some time.");

}

}

if (employee == null)

{

return HttpNotFound();

}

return View(employee);

}

[HttpPost]

[ValidateAntiForgeryToken]

public async Task<ActionResult> Edit([Bind(Include = "Id,Name,Address,Gender,Company,Designation")] Employee employee)

{

if (ModelState.IsValid)

{

using (var client = new HttpClient())

{

client.BaseAddress = new Uri(apiBaseAddress);

var response = await client.PutAsJsonAsync("employees/edit", employee);

if (response.IsSuccessStatusCode)

{

return RedirectToAction("Index");

}

else

{

ModelState.AddModelError(string.Empty, "Server error try after some time.");

}

}

return RedirectToAction("Index");

}

return View(employee);

}

public async Task<ActionResult> Delete(string id)

{

if (id == null)

{

return new HttpStatusCodeResult(HttpStatusCode.BadRequest);

}

Employee employee = null;

using (var client = new HttpClient())

{

client.BaseAddress = new Uri(apiBaseAddress);

var result = await client.GetAsync($"employees/details/{id}");

if (result.IsSuccessStatusCode)

{

employee = await result.Content.ReadAsAsync<Employee>();

}

else

{

ModelState.AddModelError(string.Empty, "Server error try after some time.");

}

}

if (employee == null)

{

return HttpNotFound();

}

return View(employee);

}

[HttpPost, ActionName("Delete")]

[ValidateAntiForgeryToken]

public async Task<ActionResult> DeleteConfirmed(string id)

{

using (var client = new HttpClient())

{

client.BaseAddress = new Uri(apiBaseAddress);

var response = await client.DeleteAsync($"employees/delete/{id}");

if (response.IsSuccessStatusCode)

{

return RedirectToAction("Index");

}

else

ModelState.AddModelError(string.Empty, "Server error try after some time.");

}

return View();

}

}

}

You can see, we have defined an “apiBaseAddress” variable globally and got the value for apiBaseAddress from Web.Config file. We will use this value in all our controller actions.

1. The index action and the index view

Index Action

public async Task<ActionResult> Index()

{

IEnumerable<Employee> employees = null;

using (var client = new HttpClient())

{

client.BaseAddress = new Uri(apiBaseAddress);

var result = await client.GetAsync("employees/get");

if (result.IsSuccessStatusCode)

{

employees = await result.Content.ReadAsAsync<IList<Employee>>();

}

else

{

employees = Enumerable.Empty<Employee>();

ModelState.AddModelError(string.Empty, "Server error try after some time.");

}

}

return View(employees);

}

If you look at the index action, you can see, we have declared a “HttpClient” variable and using client.GetAsync method, we have got the employee data result from Web API and store in a “result” variable. We have again read the employee data from this result variable using “ReadAsync” method.

We can now modify the Index view.

Index.cshtml

@model IEnumerable<MVCwithWebAPI.Models.Employee>

@{

ViewBag.Title = "Employee List";

}

<h3>Employee List</h3>

<p>

@Html.ActionLink("New Employee", "Create")

</p>

<table class="table">

<tr>

<th>

@Html.DisplayNameFor(model => model.Name)

</th>

<th>

@Html.DisplayNameFor(model => model.Address)

</th>

<th>

@Html.DisplayNameFor(model => model.Gender)

</th>

<th>

@Html.DisplayNameFor(model => model.Company)

</th>

<th>

@Html.DisplayNameFor(model => model.Designation)

</th>

<th></th>

</tr>

@foreach (var item in Model)

{

<tr>

<td>

@Html.ActionLink(item.Name, "Details", new { id = item.Id })

</td>

<td>

@Html.DisplayFor(modelItem => item.Address)

</td>

<td>

@Html.DisplayFor(modelItem => item.Gender)

</td>

<td>

@Html.DisplayFor(modelItem => item.Company)

</td>

<td>

@Html.DisplayFor(modelItem => item.Designation)

</td>

<td>

@Html.ActionLink("Edit", "Edit", new { id = item.Id }) |

@Html.ActionLink("Delete", "Delete", new { id = item.Id })

</td>

</tr>

}

</table>

We have modified the existing “Index” view. We have removed the “Details” link from this view and instead, we have given a hyperlink in the employee name itself for details.

1. The create view

We can modify the “Create” view by removing the Id field. Because for us, employee Id will be created automatically while inserting new data. We have used system GUID for this.

Create.cshtml

@model MVCwithWebAPI.Models.Employee

@{

ViewBag.Title = "Create Employee";

}

<h3>Create Employee</h3>

@using (Html.BeginForm())

{

@Html.AntiForgeryToken()

<div class="form-horizontal">

<hr />

@Html.ValidationSummary(true, "", new { @class = "text-danger" })

<div class="form-group">

@Html.LabelFor(model => model.Name, htmlAttributes: new { @class = "control-label col-md-2" })

<div class="col-md-10">

@Html.EditorFor(model => model.Name, new { htmlAttributes = new { @class = "form-control" } })

@Html.ValidationMessageFor(model => model.Name, "", new { @class = "text-danger" })

</div>

</div>

<div class="form-group">

@Html.LabelFor(model => model.Address, htmlAttributes: new { @class = "control-label col-md-2" })

<div class="col-md-10">

@Html.EditorFor(model => model.Address, new { htmlAttributes = new { @class = "form-control" } })

@Html.ValidationMessageFor(model => model.Address, "", new { @class = "text-danger" })

</div>

</div>

<div class="form-group">

@Html.LabelFor(model => model.Gender, htmlAttributes: new { @class = "control-label col-md-2" })

<div class="col-md-10">

@Html.EditorFor(model => model.Gender, new { htmlAttributes = new { @class = "form-control" } })

@Html.ValidationMessageFor(model => model.Gender, "", new { @class = "text-danger" })

</div>

</div>

<div class="form-group">

@Html.LabelFor(model => model.Company, htmlAttributes: new { @class = "control-label col-md-2" })

<div class="col-md-10">

@Html.EditorFor(model => model.Company, new { htmlAttributes = new { @class = "form-control" } })

@Html.ValidationMessageFor(model => model.Company, "", new { @class = "text-danger" })

</div>

</div>

<div class="form-group">

@Html.LabelFor(model => model.Designation, htmlAttributes: new { @class = "control-label col-md-2" })

<div class="col-md-10">

@Html.EditorFor(model => model.Designation, new { htmlAttributes = new { @class = "form-control" } })

@Html.ValidationMessageFor(model => model.Designation, "", new { @class = "text-danger" })

</div>

</div>

<div class="form-group">

<div class="col-md-offset-2 col-md-10">

<input type="submit" value="Create" class="btn btn-default" />

</div>

</div>

</div>

}

<div>

@Html.ActionLink("Back to List", "Index")

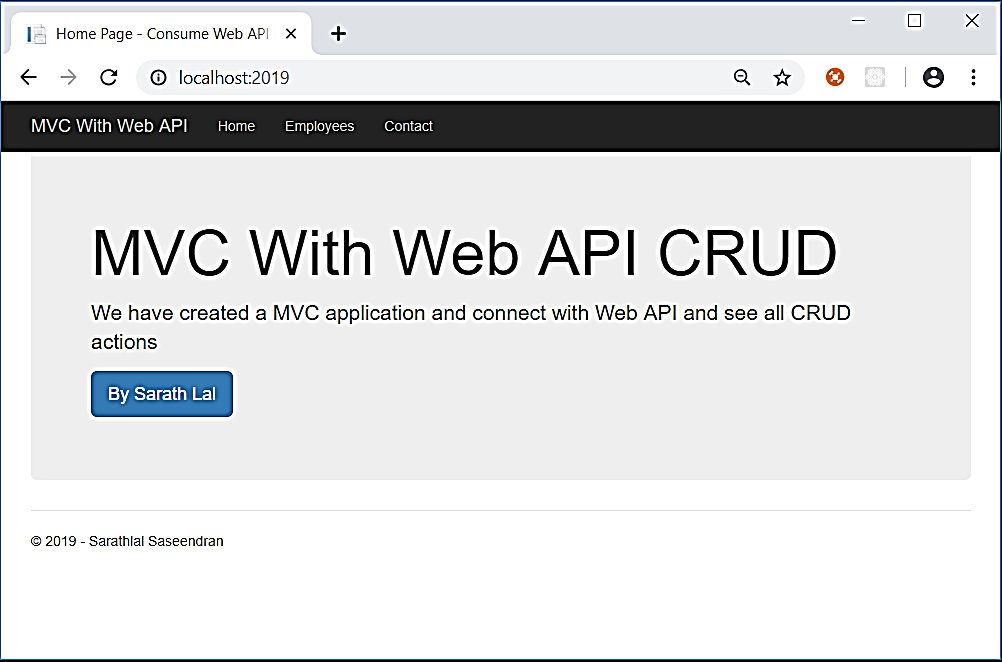
</div>

@section Scripts {

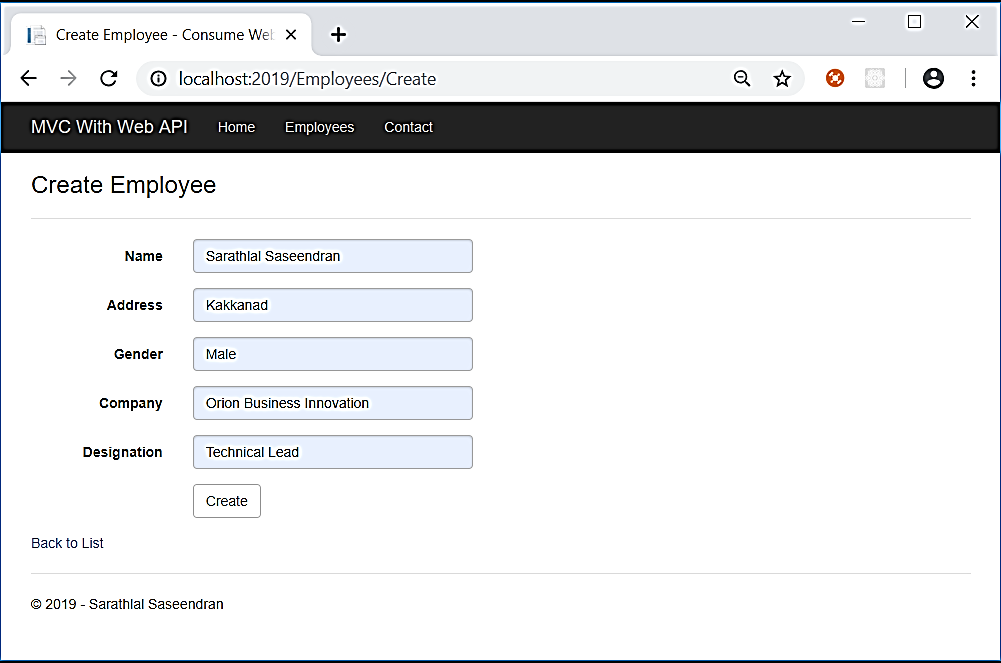
@Scripts.Render("~/bundles/jqueryval")

}

We can run the application now. The landing page looks like the below screenshot.



We can click the “Employees” link and click the “New Employee” link to create a new employee.



1. Test CRUD Actions