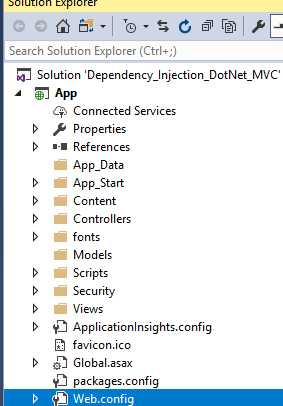
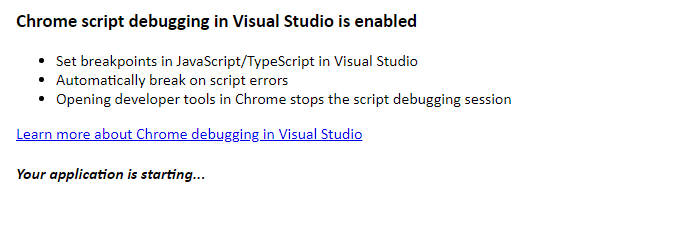
Steps:

1. Open visual studio and create web application selecting MVC.

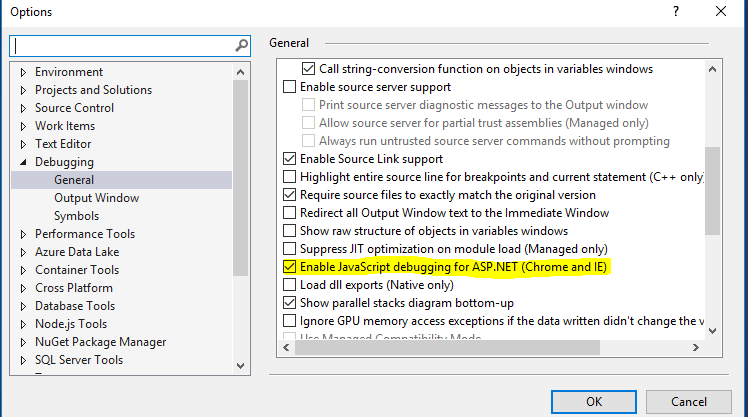


1. Run the application.

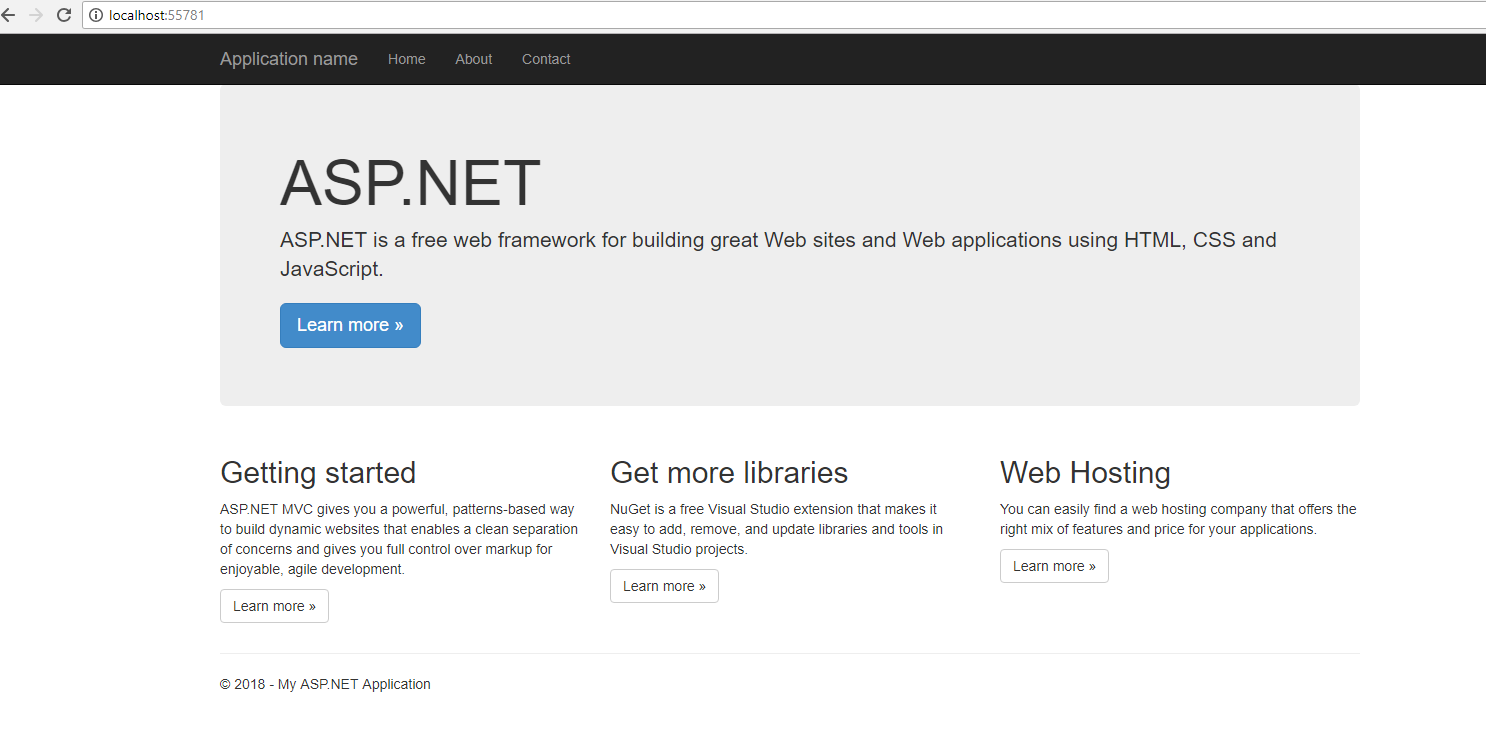
Problem occurred: while running application in visual studio 2017 community, you can see below screen



So wait until the application get loads, it was taking too much time so you can disable it from tools->options->debugging and uncheck the item



1. Once application loaded you can see



1. Install “Ninject.MVC5” and “WebActivatorEx” package from nuget. Packages.config file will have now entries

<package id="Ninject" version="3.3.4" targetFramework="net461" />

<package id="Ninject.MVC5" version="3.3.0" targetFramework="net461" />

<package id="Ninject.Web.Common" version="3.3.0" targetFramework="net461" />

<package id="Ninject.Web.Common.WebHost" version="3.3.0" targetFramework="net461" />

<package id="WebActivatorEx" version="2.2.0" targetFramework="net461" />

1. After installing all packages Ninject will create “NinjectWebCommon.cs” file under App\_start folder, if not then create one file with same name.
2. Adding all necessary methods in this file.

[assembly: WebActivatorEx.PreApplicationStartMethod(typeof(Dependency\_Injection\_DotNet\_MVC.App\_Start.NinjectWebCommon), "Start")]

[assembly: WebActivatorEx.ApplicationShutdownMethodAttribute(typeof(Dependency\_Injection\_DotNet\_MVC.App\_Start.NinjectWebCommon), "Stop")]

namespace Dependency\_Injection\_DotNet\_MVC.App\_Start

{

using System;

using System.Web;

using Microsoft.Web.Infrastructure.DynamicModuleHelper;

using Ninject;

using Ninject.Web.Common;

using Ninject.Web.Common.WebHost;

public class NinjectWebCommon

{

private static readonly Bootstrapper bootstrapper = new Bootstrapper();

/// <summary>

/// Starts the application

/// </summary>

public static void Start()

{

DynamicModuleUtility.RegisterModule(typeof(OnePerRequestHttpModule));

DynamicModuleUtility.RegisterModule(typeof(NinjectHttpModule));

bootstrapper.Initialize(CreateKernel);

}

/// <summary>

/// Stops the application.

/// </summary>

public static void Stop()

{

bootstrapper.ShutDown();

}

/// <summary>

/// Creates the kernel that will manage your application.

/// </summary>

/// <returns>The created kernel.</returns>

private static IKernel CreateKernel()

{

var kernel = new StandardKernel();

try

{

kernel.Bind<Func<IKernel>>().ToMethod(ctx => () => new Bootstrapper().Kernel);

kernel.Bind<IHttpModule>().To<HttpApplicationInitializationHttpModule>();

RegisterServices(kernel);

return kernel;

}

catch

{

kernel.Dispose();

throw;

}

}

/// <summary>

/// Load your modules or register your services here!

/// </summary>

/// <param name="kernel">The kernel.</param>

private static void RegisterServices(IKernel kernel)

{

}

}

}

RegisterServices method is still empty. Application dependencies will get added here.

1. Create one new folder “Security” under project and add new class with name “ApplicationDependencyModule.cs”. Content of file

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using Ninject.Modules;

namespace Dependency\_Injection\_DotNet\_MVC.Security

{

public class ApplicationDependencyModule : NinjectModule

{

public override void Load()

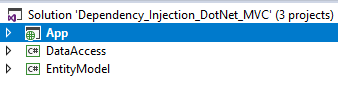
{

}

}

}

1. Add two projects “DataAccess” and “EntityModel”. Dataaccess is for interacting with database and entity model will contains entities. Add references of these projects in “App” project



1. In DataAccess layer, create one folder “Interface” and interface file “ICompanyRepository.cs”

File contents:

namespace DataAccess.Interface

{

public interface ICompanyRepository

{

List<Company> GetCompanies();

}

}

Now, create “Repository” folder and add class file “CompanySqlRepository.cs”. Implement interface and created the constructor to inject IDbConnection which will return sqlconnection. Here we also added connectionstring parameter, which we will send when adding DI in NInject.

File contents:

using System.Collections.Generic;

using System.Data;

using DataAccess.Interface;

using EntityModel.DomainModel;

namespace DataAccess.Repository.Dapper.MSSQL

{

public class CompanySqlRepository : ICompanyRepository

{

private IDbConnection \_connection;

public CompanySqlRepository(IDbConnection conn, string conString)

{

\_connection = conn;

\_connection.ConnectionString = conString;

}

public List<Company> GetCompanies()

{

//return Query<Company>("usp\_GetAllCompanies").ToList();

return null;

}

}

}

1. Now we will be adding DI for SqlConnection and also for CompanySqlRepository classes. Open ApplicationDependencyModule.cs

Add below lines in load function

public override void Load()

{

var conString = ConfigurationManager.ConnectionStrings["mssql"].ConnectionString;

//Whereever IDbConnection is injected, it will return SqlConnection class

Bind<IDbConnection>().To<SqlConnection>().InRequestScope();

//Whereever ICompanyRepository is injected, it will return CompanySqlRepository class

Bind<ICompanyRepository>().To<DataAccess.Repository.Dapper.MSSQL.CompanySqlRepository>().InRequestScope().WithConstructorArgument("conString", conString);

}

Add your connection string to web.config file. In case if you don’t want to inject IDbConnection then remove constructor from CompanySqlRepository class and above Bind declaration in ApplicationDependencyModule class.

1. Open HomeController.cs page and add constructor.

private ICompanyRepository objCompanyRepository;

public HomeController(ICompanyRepository iCompanyRepository)

{

objCompanyRepository = iCompanyRepository;

}

in Index() method call Repository function using Interface

public ActionResult Index()

{

ViewBag.Companies = objCompanyRepository.GetCompanies();

return View();

}

1. Open Index.cshtml in views folder and add following code to list companies name

<div>

@foreach (var item in ViewBag.Companies)

{

<li>@item.Id: @item.Name</li>

}

</div>

