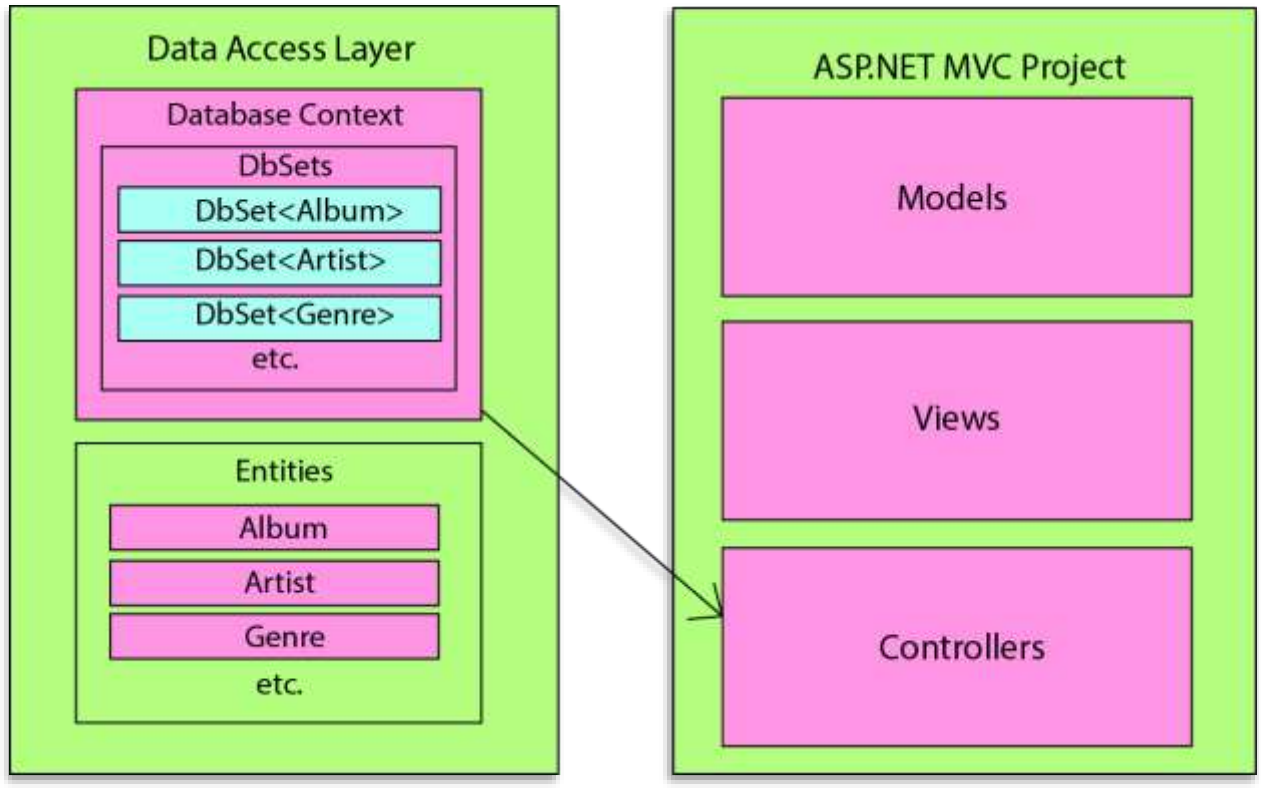
MVC - Repository-And-Unit-Of-Work

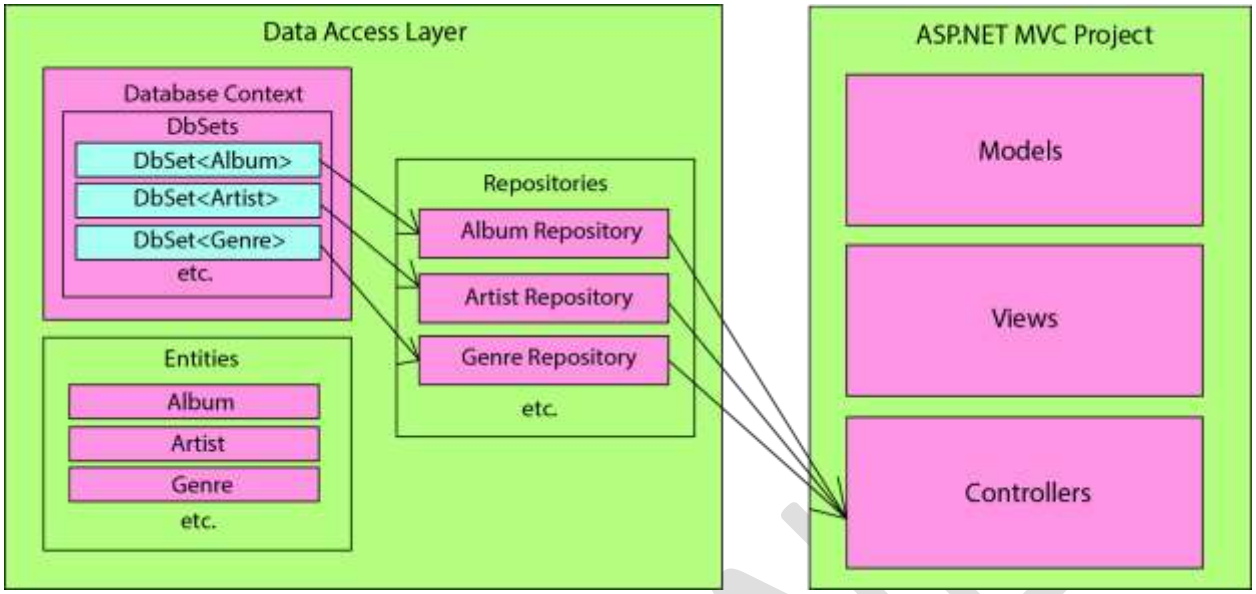
What are the Repository and Unit of Work Design Patterns?

When you set up an ASP.NET MVC project with the Entity framework, you can think of your project as consisting of four major parts: the first three are the obvious Models, Views and Controllers that make up the MVC project and the fourth is the data access layer where you make use of the Entity framework to retrieve data from a data source. The question is, how should we connect our data access layer to the parts of our MVC project? The quickest way is to simply create an instance of your database context in your controller, giving the controller direct access to the database context as illustrated here:



Normal MVC Project

A repository is a class that encapsulates the CRUD operations for an entity. As such, the Repository pattern allows us to write the data access logic for an entity in one place. This is good because if the data access logic needs to change, we don't even need to touch the controllers. This makes your project much more maintainable. What we could do now is give the controllers direct access to the repositories that they need as illustrated below.



MVC With Repository:

- But giving your controllers direct access to repositories like this can still lead to problems. Firstly, where should the database context go? After all, each repository needs access to a database context to perform its data access operations and the database context's SaveChanges() method must be called from somewhere to save any changes to the database. How should we go about doing this?

- One option is that we can create an instance of the database context in the controllers and pass it to the repositories.

- This is where the Unit of Work design pattern comes to the rescue. It functions as a centralized place to house your repositories using a single instance of a database context to access and save data. Giving the controllers access to a UnitOfWork class as a means to retrieve and update data solves many of the above mentioned issues

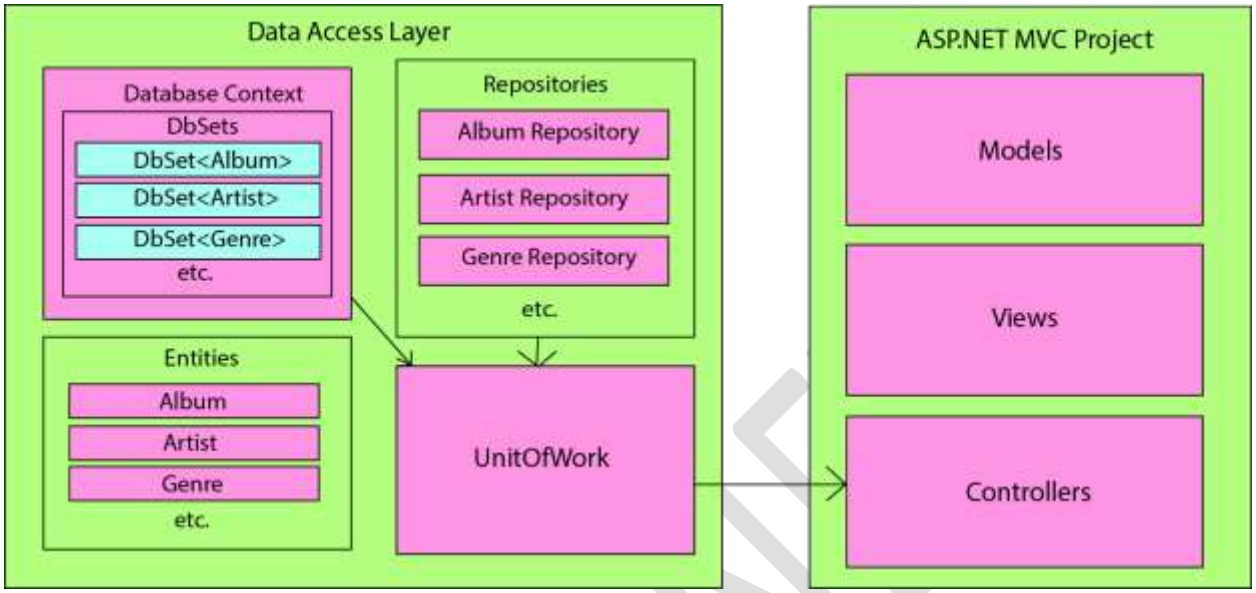
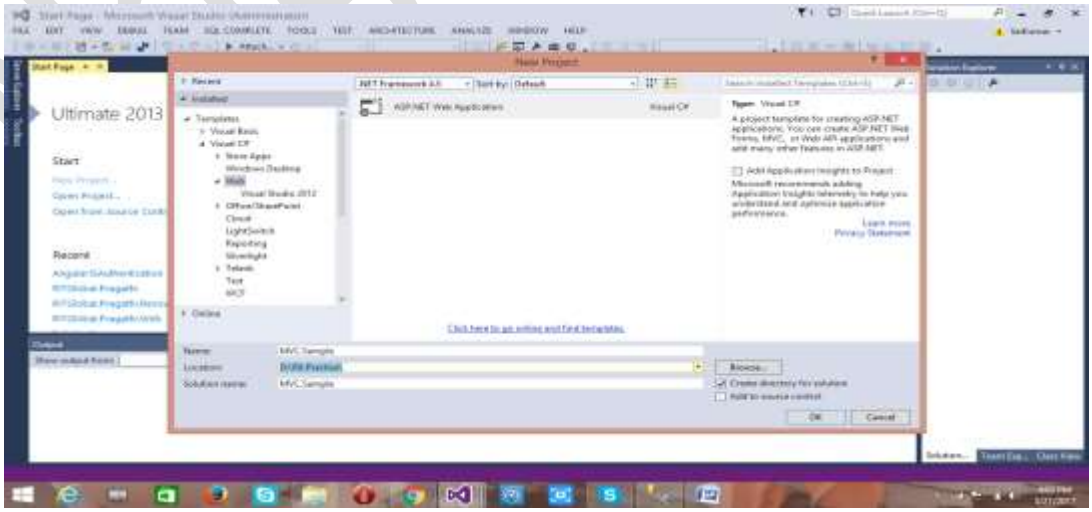


Fig:MVC Repository & Unit Of Work

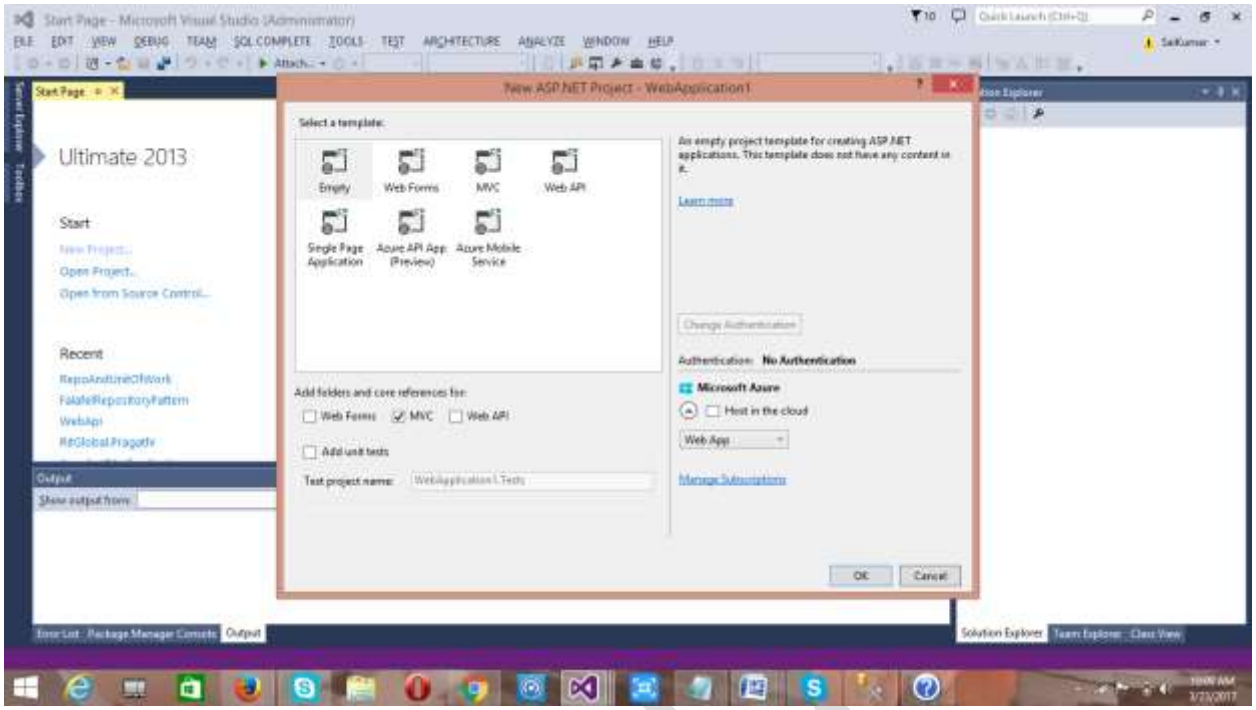
- In this figure, we see the controller only gets direct access to the UnitOfWork class.
- The UnitOfWork class handles the access of and updates to all the repositories using the database context.
- Since the UnitOfWork class encapsulates the repositories and the database context, we have the freedom to easily change the implementation of our data access layer without having to change the controller's logic.

Follow below steps to create MVC Project:

- Open the Visual Studio and click File → New → Project.
- From the left pane, select Templates → Visual C# → Web.(ASP.NET Web Application)

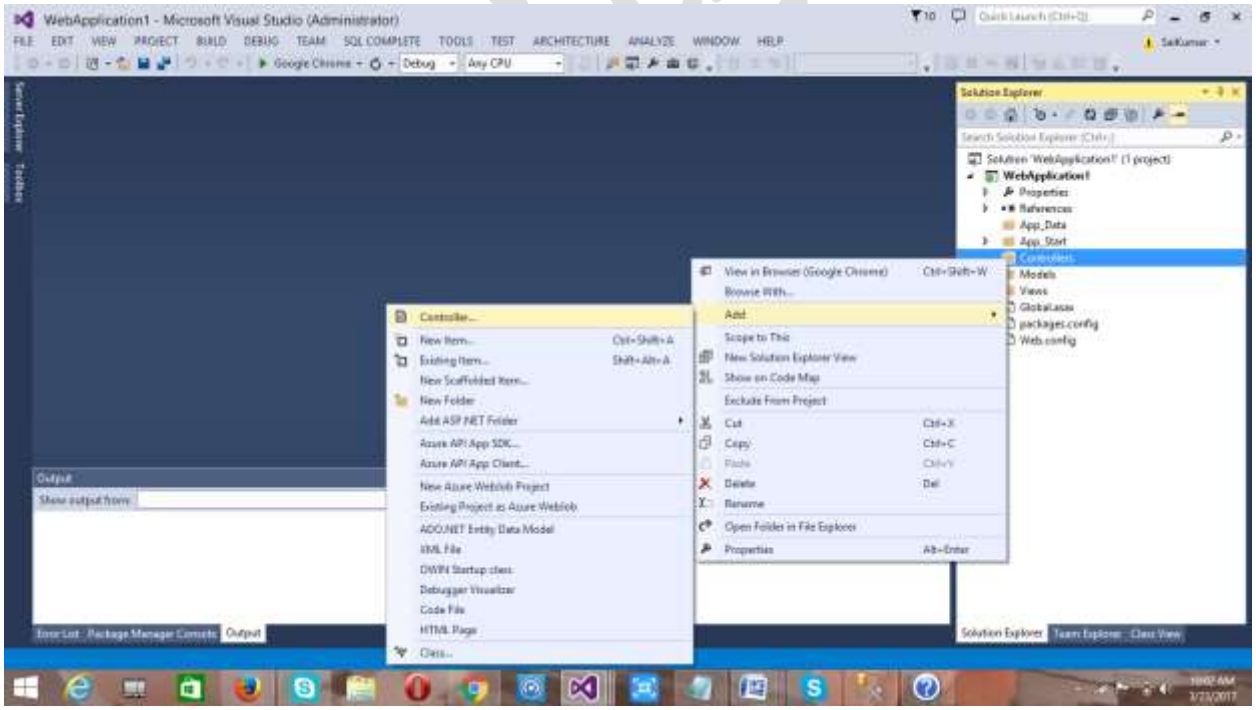


By Clicking Ok it prompts window

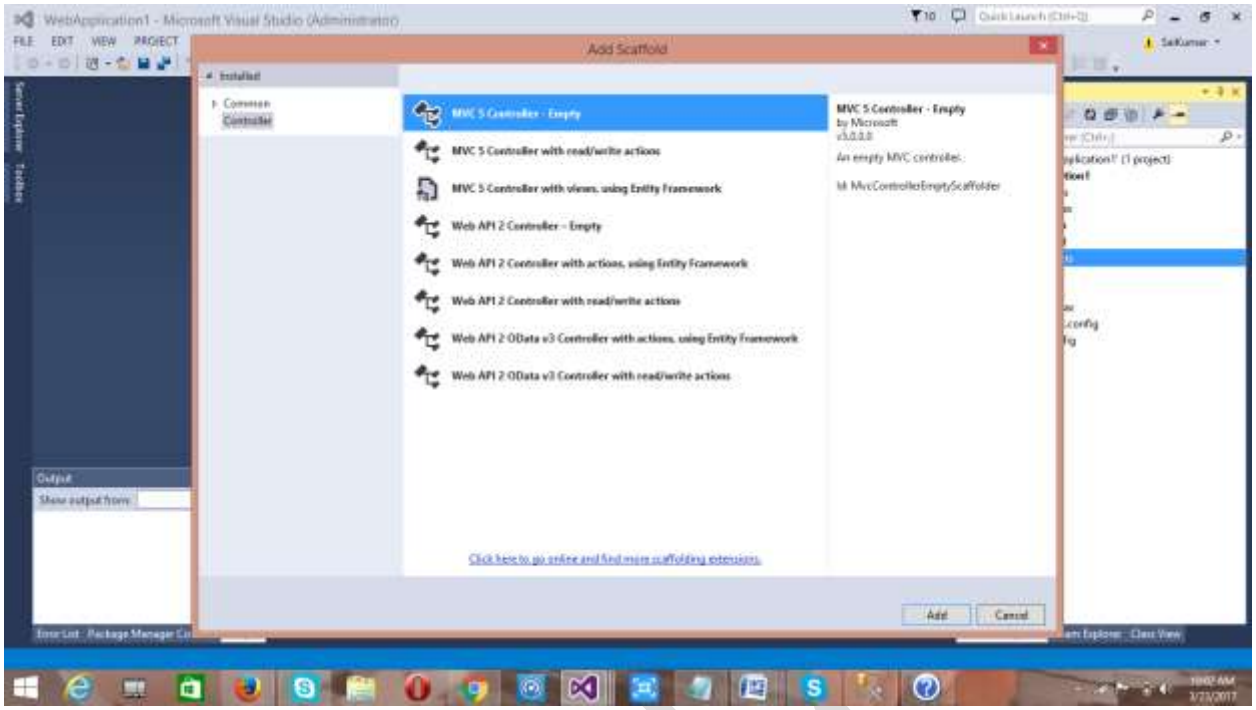


Now Add Controller:

Controllers→Add→Controller

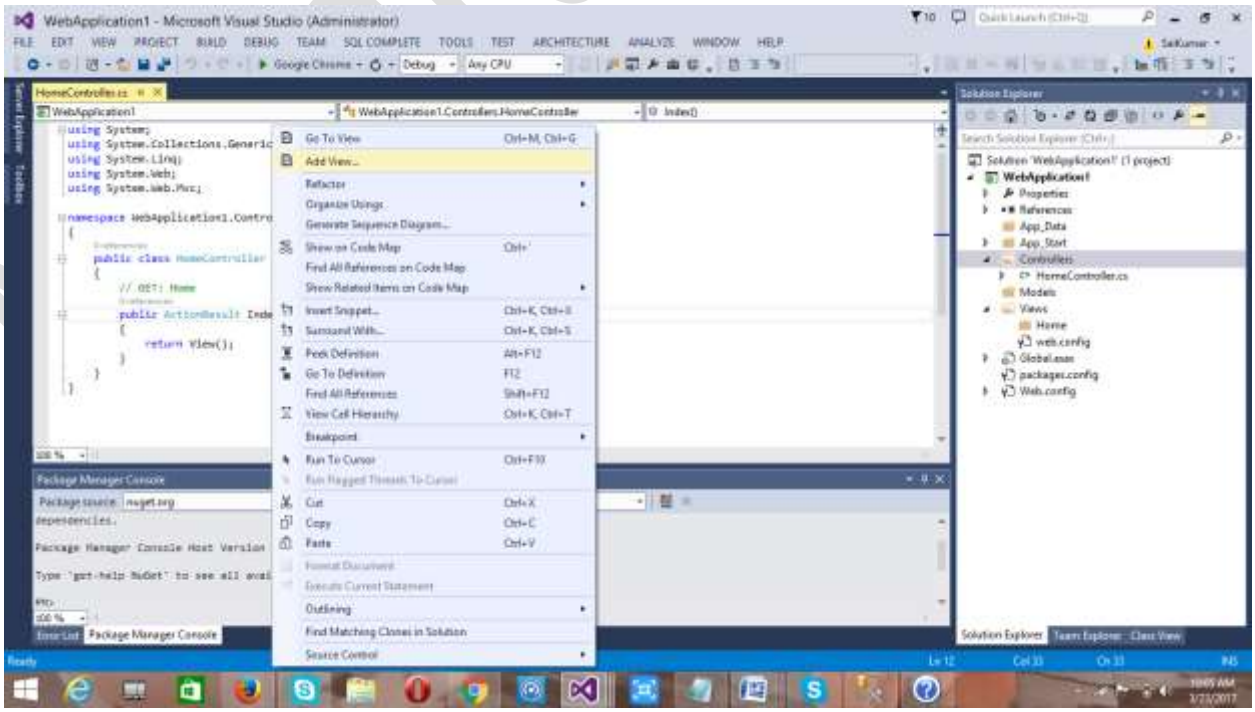


By Clicking opens window:

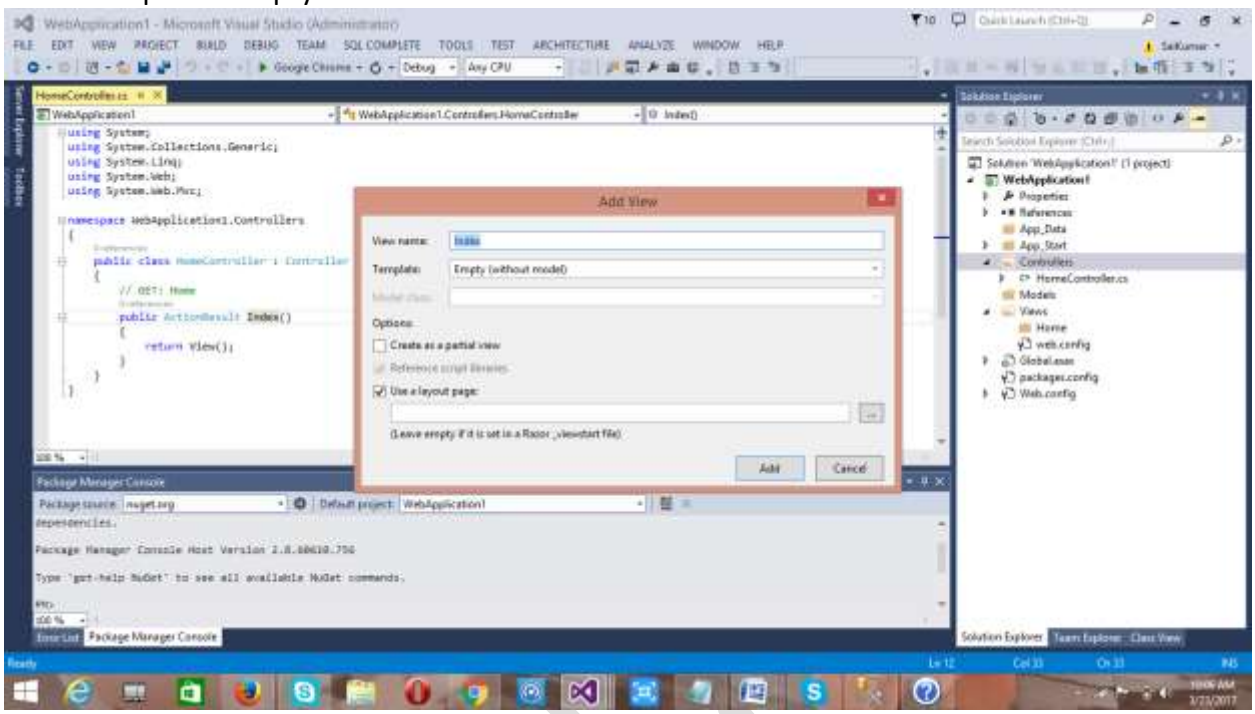


Now Create View for Controller:

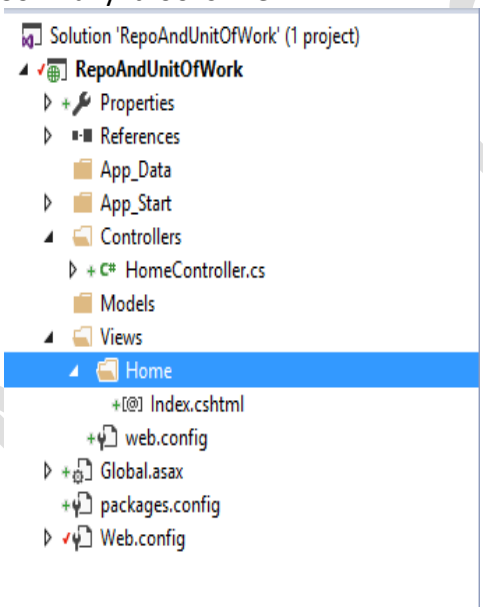
- Right Click on Index→Add View



Opens window:
Select Template : Empty



So finally it looks like:



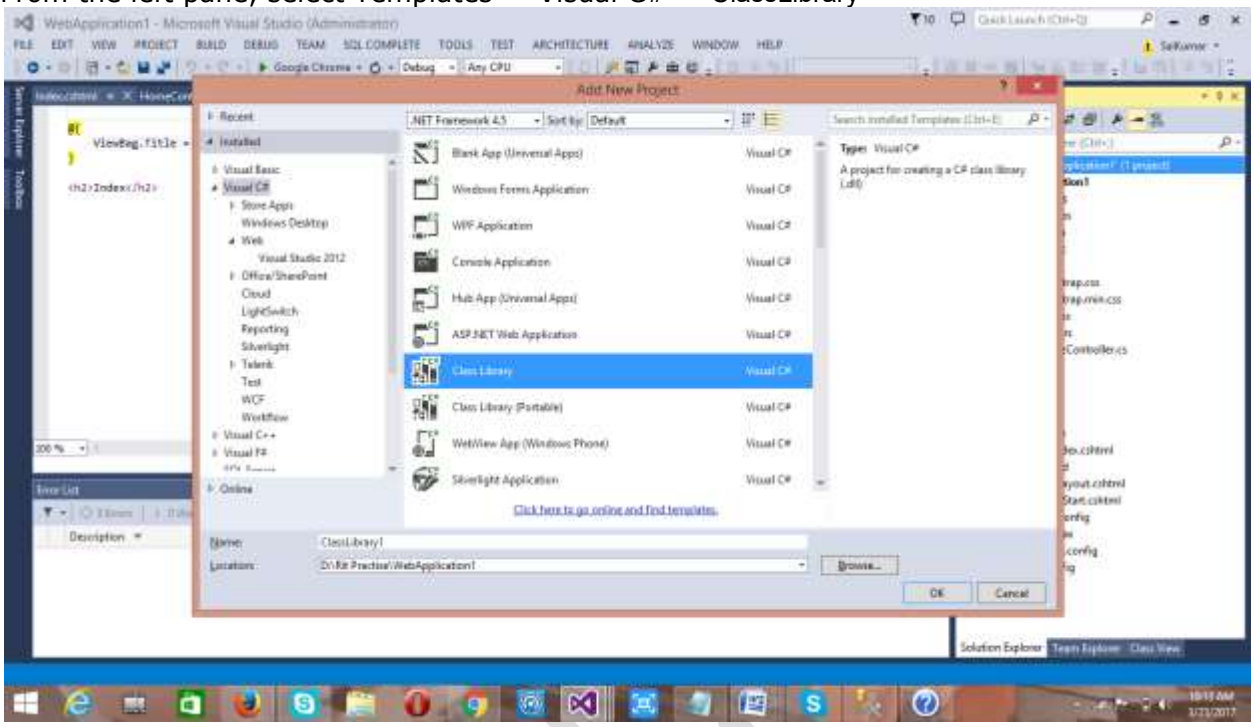
- We are finished with the MVC portion of the project so we will move on to the entities, that is, the classes containing the data you want to display and/or modify on your web page

- Create a new Class Library project in the solution. These will be classes that model domain objects as well as the classes that interact with the data source.

- Create the Class Library in your solution:

Solution→Add→NewProject

From the left pane, select Templates → Visual C# → ClassLibrary

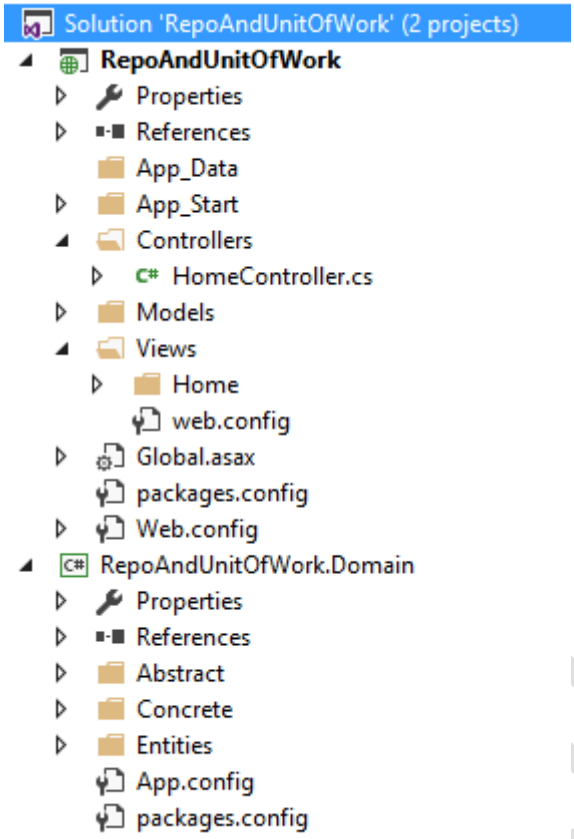


Add 3 Folders in ClassLibrary Project name as Abstract ,Concrete and Entities

- Abstract folder is for interfaces and/or abstract classes.
- The Concrete folder will hold implementations of these classes as well as other data access classes.
- The Entities folder will hold POCO domain object classes.

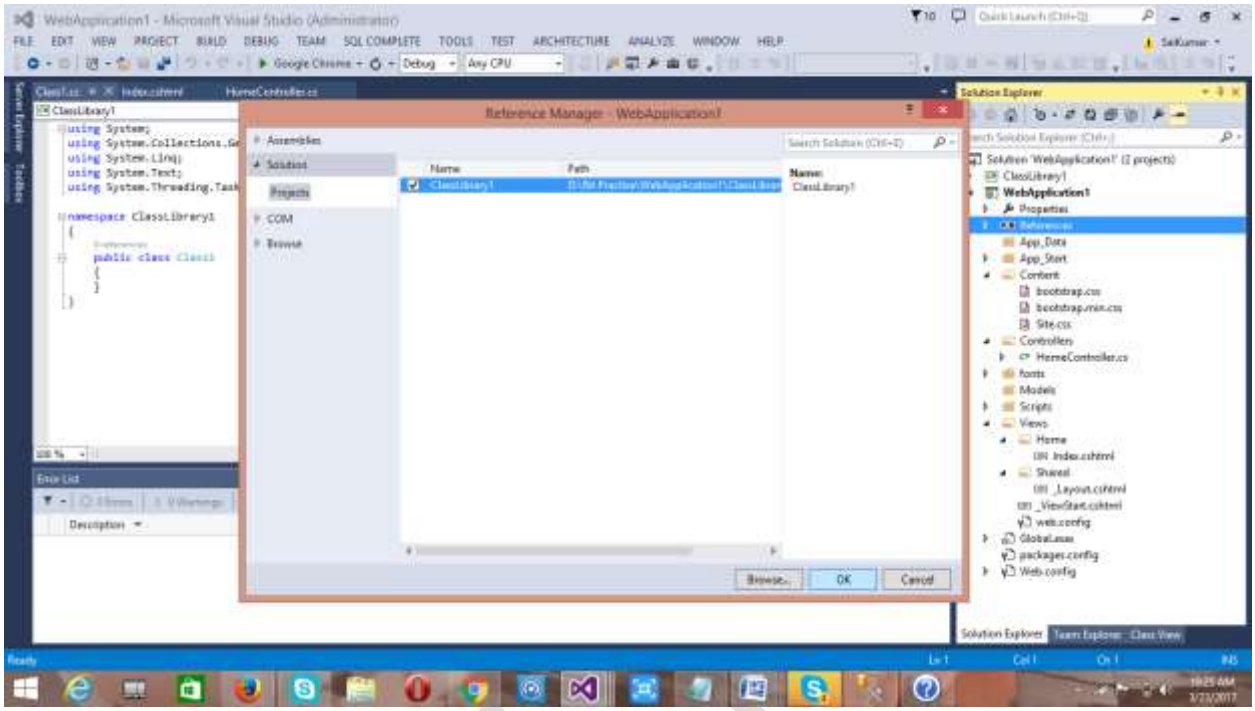
- **POCO.** A simple object without complicated logic which doesn't require to be identifiable, usually it has just a few properties and is used with ORM or as a Data Transfer Object

Finally Stucture looks like :

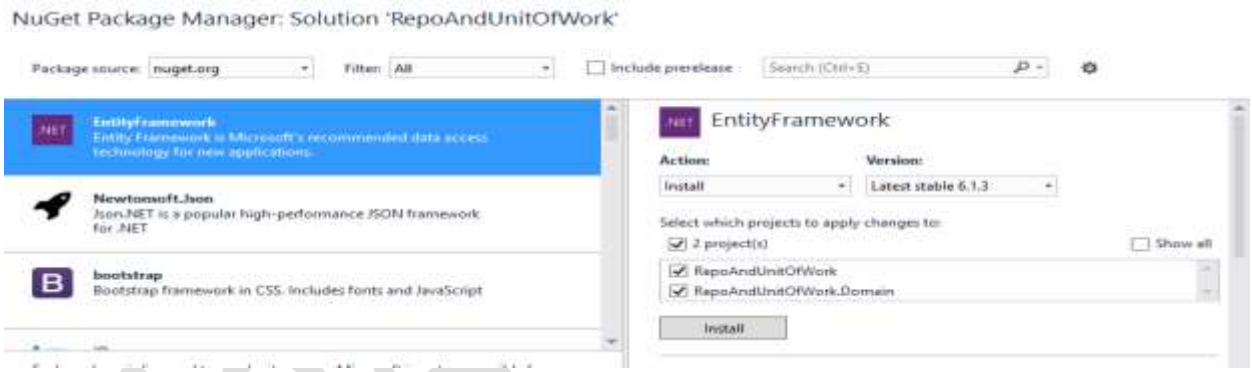


- Since we want to be accessing classes from the class library you will need to add a reference to it in your MVC project. To do so,

MVC project → References → Right Click →Add Reference and add a reference to Class Library Project.



Now Open the Nuget Package Manager and install the **Entity Framework**.



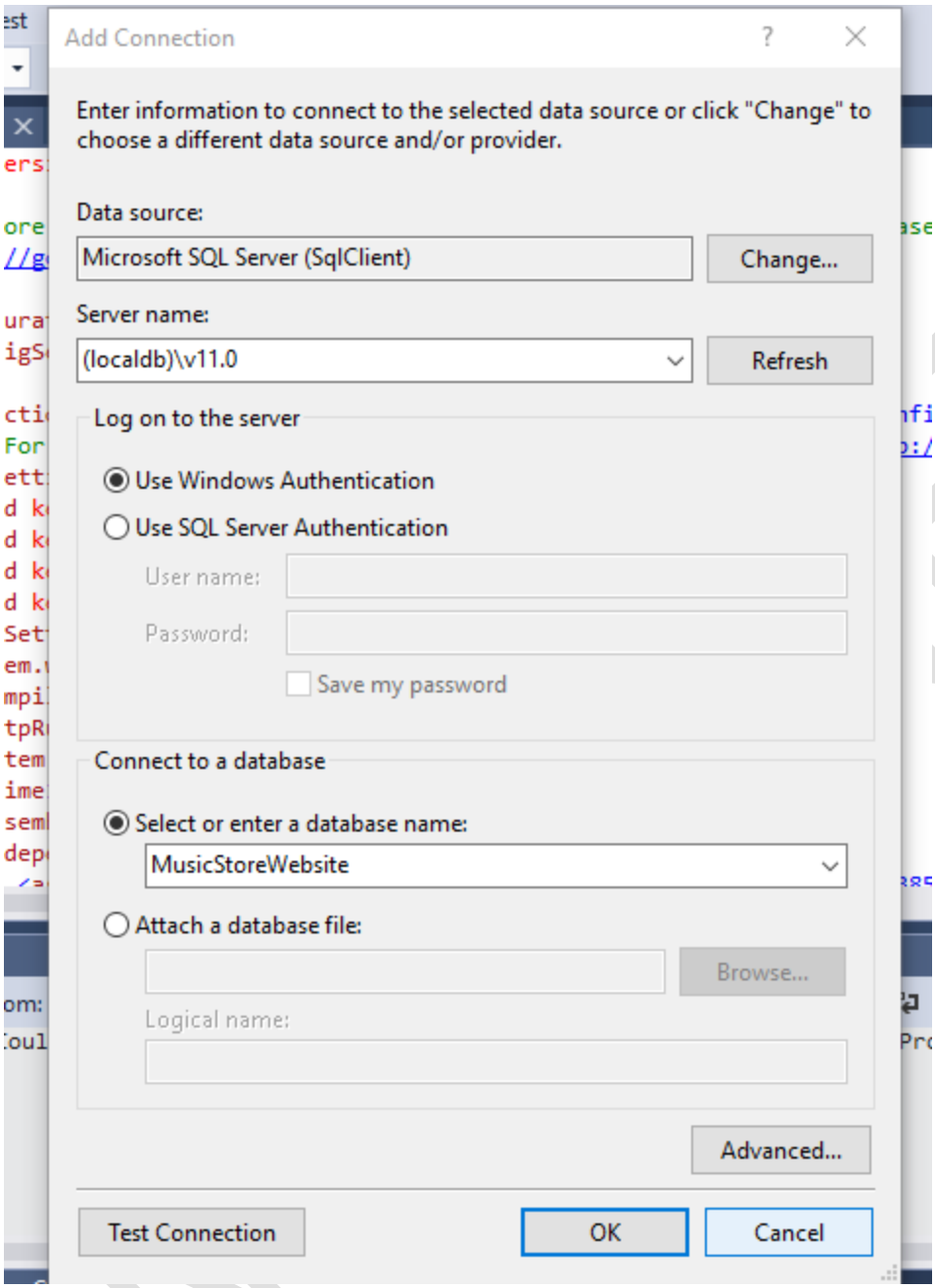
Create an **Album** and **Account** class in the Entities folder like this:

```
namespace RepoAndUnitOfWork.Domain.Entities
{
    public class Album
    {
        public int AlbumId { get; set; }
        public string Title { get; set; }
        public string Artist { get; set; }
    }
```

```
}
```

```
namespace RepoAndUnitOfWork.Domain.Entities
{
    public class Account
    {
        public int AccountId { get; set; }
        public string UserName { get; set; }
        public string Password { get; set; }
    }
}
```

Now we can establish our database connection. I will just use the SQL Server LocalDB. Firstly, add the database connection by going to Tools > 'Connect to Database...' and filling out the form like this:



Next, in the RepoAndUnitOfWork MVC project add the following connection string to the web.config file :

```
<connectionStrings>
    <add name="MusicStoreDbContext" connectionString="Data
Source=(localdb)\v11.0;Initial Catalog=MusicStoreWebsite;Integrated
Security=True" providerName="System.Data.SqlClient" />
  </connectionStrings>
```

Now, in the **Concrete** folder, add the **MusicStoreDbContext** class:

MusicStoreDbContext.cs

```
using System.Data.Entity;
using RepoAndUnitOfWork.Domain.Entities;

namespace RepoAndUnitOfWork.Domain.Concrete
{
    public class MusicStoreDbContext : DbContext
    {
        public DbSet<Album> Albums { get; set; }
        public DbSet<Account> Accounts { get; set; }
    }
}
```

Add some sample data to the database. To do so, we will just use our HomeController in the Controllers folder of the MVC project.

HomeController:

```
using System.Web.Mvc;
using RepoAndUnitOfWork.Domain.Concrete;
using RepoAndUnitOfWork.Domain.Entities;
namespace RepoAndUnitOfWork.Controllers
{
    public class HomeController : Controller
    {
        MusicStoreDbContext dbContext = new MusicStoreDbContext();
        public ActionResult Index()
        {
            dbContext.Albums.Add(new Album{AlbumId = 1, Title =
"Odelay", Artist = "Beck"});
            dbContext.Accounts.Add(new Account { AccountId = 1,
UserName = "JaneDoe", Password = "123456"});
            dbContext.Accounts.Add(new Account { AccountId = 2,
UserName = "user18081971", Password = "QKThr" });
```

```
            dbContext.SaveChanges();
            return View();
        }
    }
}
```

Now, build and run the solution

This will call the HomeController's Index() method and create the tables in your database and populate them with data. To verify the date is added, you can check out your database in the server explorer which should now have the tables with the added data.:

Now Set Up is ready , so we will implement Repository and Unit of Work design patterns.

Creating the Repositories

To begin, create the following repository interface in the Abstract folder of your ClassLibray project.

IRepository.cs

```
using System;
using System.Collections.Generic;

namespace RepoAndUnitOfWork.Domain.Abstract
{
    public interface IRepository<T> : IDisposable where T : class
    {
        //Method to get all rows in a table
        IEnumerable<T> DataSet { get;}

        //Method to add row to the table
        void Create(T entity);

        //Method to fetch row from the table
        T GetById(int? id);
```

```
        //Method to update a row in the table
        void Update(T entity);

        //Method to delete a row from the table
        void Delete(T entity);
    }
}
```

This generic interface merely serves as a contract for all repositories
They all must have implementations of these methods

Next, in the Concrete folder, create the following generic class which implements IRepository

Repository.cs

```
using System;
using System.Collections.Generic;
using System.Data.Entity;
using RepoAndUnitOfWork.Domain.Abstract;
namespace RepoAndUnitOfWork.Domain.Concrete
{
    public class Repository<T> : IRepository<T>, IDisposable where T :
class
    {
        protected MusicStoreDbContext dbContext;

        //DbSet usable with any of our entity types
        protected DbSet<T> dbSet;

        //constructor taking the database context and getting the
appropriately typed data set from it
        public Repository(MusicStoreDbContext context)
        {
            dbContext = context;
            dbSet = context.Set<T>();
```

```
        }

        //Implementation of IRepository methods
        public virtual IEnumerable<T> DataSet { get { return dbSet; } }
}

        public virtual void Create(T entity)
        {
            dbSet.Add(entity);
        }

        public virtual T GetById(int? id)
        {
            return dbSet.Find(id);
        }

        public virtual void Update(T entity)
        {
            dbContext.Entry(entity).State = EntityState.Modified;
        }

        public virtual void Delete(T entity)
        {
            dbSet.Remove(entity);
        }

        //IDisposable implementation
        private bool disposed = false;

        protected virtual void Dispose(bool disposing)
        {
```

```
            if (!this.disposed)
            {
                if (disposing)
                {
                    dbContext.Dispose();
                }
            }
        }

        public void Dispose()
        {
            Dispose(true);
            GC.SuppressFinalize(this);
        }
    }
}
```

As you can see, this class includes implementations of IRepository's create, read, update and delete operations. It is a generic class because we can reuse it to deal with different types of entities. Its usage will become clearer when we create the UnitOfWork class.

At this stage, the repository portion of the Repository and Unit of Work Patterns is complete.

Creating the Unit of Work Class

Now that we have our repository class set up, we can now create our unit of work class. This class will be used to provide access to and save changes made to our repositories, all using a single database context.

UnifOfWork.cs

```
using System;
using RepoAndUnitOfWork.Domain.Entities;
using RepoAndUnitOfWork.Domain.Abstract;

namespace RepoAndUnitOfWork.Domain.Concrete
{
    public class UnitOfWork : IDisposable
```

```
    {
        //Our database context
        private MusicStoreDbContext dbContext = new
MusicStoreDbContext();

        //Private members corresponding to each concrete repository
        private Repository<Account> accountRepository;
        private Repository<Album> albumRepository;

        //Accessors for each private repository, creates repository if
null
        public IRepository<Account> AccountRepository{
            get
            {
                if (accountRepository == null)
                {
                    accountRepository = new
Repository<Account>(dbContext);
                }
                return accountRepository;
            }

        }

        public IRepository<Album> AlbumRepository
        {
            get
            {
                if (albumRepository == null)
                {
                    albumRepository = new
Repository<Album>(dbContext);
```

```
            }
            return albumRepository;
        }

    }

    //Method to save all changes to repositories
    public void Commit()
    {
        dbContext.SaveChanges();
    }

    //IDisposible implementation
    private bool disposed = false;

    protected virtual void Dispose(bool disposing)
    {
        if (!this.disposed)
        {
            if (disposing)
            {
                dbContext.Dispose();
            }
        }
    }

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }
```

```
    }
}
```

How to Use the Unit of Work Class

With our UnitOfWork class complete we can now use it to access and modify data in our MVC project. To do so, simply create an instance of UnitOfWork in your controller like so.

HomeController.cs

```
using System.Web.Mvc;
using RepoAndUnitOfWork.Domain.Concrete;
namespace RepoAndUnitOfWork.Controllers
{
    public class HomeController : Controller
    {
        UnitOfWork unitOfWork = new UnitOfWork();
        public ActionResult Index()
        {
            return View();
        }
    }
}
```

Now you can access and modify data within your controller
In the Models folder of the MVC project, create the following view model called HomeModel

HomeModel.cs

```
using RepoAndUnitOfWork.Domain.Concrete;
using RepoAndUnitOfWork.Domain.Entities;

namespace RepoAndUnitOfWork.Models
{
    public class HomeModel
    {
        public Album Album {get; set;}
```

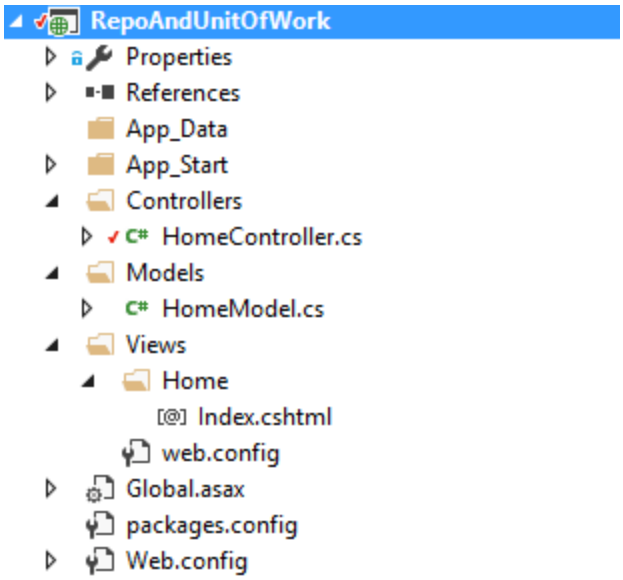
```
        public Account Account {get; set;}


        public HomeModel(UnitOfWork unitOfWork, int? albumId, int?
accountId)
        {
            Album = unitOfWork.AlbumRepository.GetById(albumId);
            Account = unitOfWork.AccountRepository.GetById(accountId);
        }
    }
}
```

Now, in your controller, all you need to do is create an instance of HomeModel and pass it to the view. Then, all data within the view model (all albums and accounts) will be accessible to your view

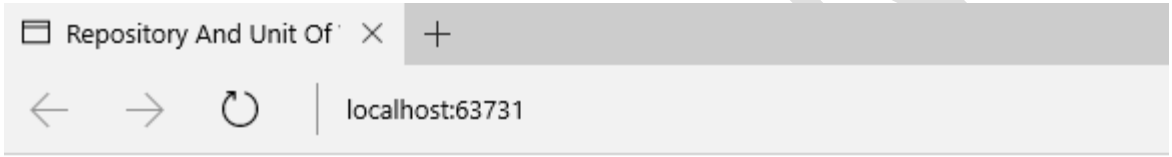
HomeController.cs

```
using System.Web.Mvc;
using RepoAndUnitOfWork.Domain.Concrete;
using RepoAndUnitOfWork.Models;
namespace RepoAndUnitOfWork.Controllers
{
    public class HomeController : Controller
    {
        UnitOfWork unitOfWork = new UnitOfWork();
        public ActionResult Index()
        {
            HomeModel model = new HomeModel(unitOfWork, 1, 2);
            return View(model);
        }
    }
}
```

So finally Our Stucture looks like :



Now Run the Project Your o/p:



Account: user18081971

Album: Odelay

Artist: Beck