ASP.NET MVC Solution Architecture – Best Practices

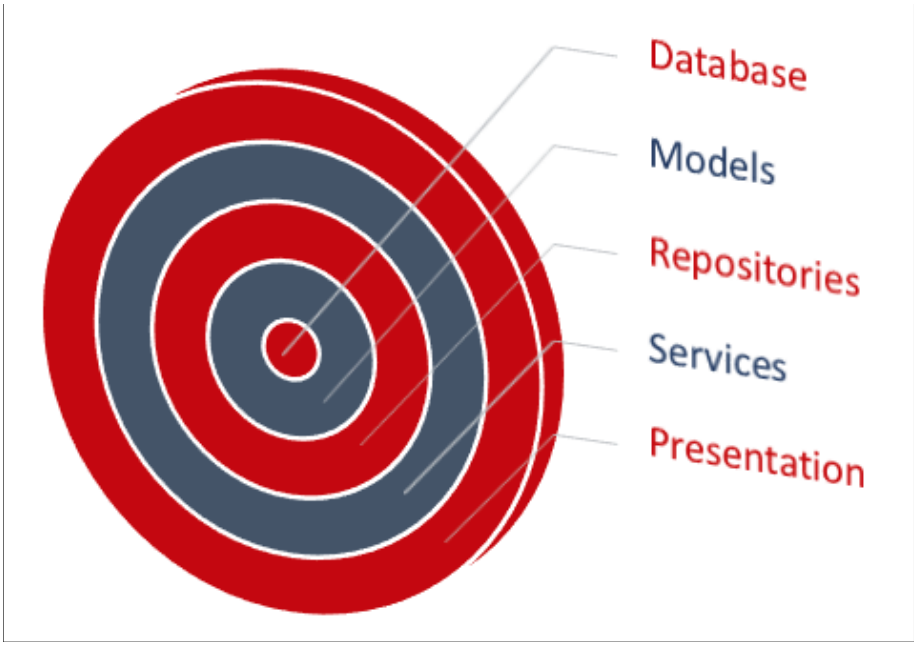
BY CHRISTOS S. *on* FEBRUARY 15, 2015 • (91)

Choosing the right architecture for Web Applications is a must, especially for large scale ones. Using the default Visual Studio ASP.NET MVC Web Application project templates, adding controllers with Scaffolding options, just to bootstrap your application and create pages and data in just a few minutes, sounds awesome for sure, but let's be honest it's not always the right choise. Peeking all the default options, keeping business, data and presentation logic in the same project will impact several factors in your solutions, such as scalability, usability or testability. In this post, will see how to keep things clean, creating a highly loosely coupled ASP.NET MVC Solution, where Data Access, Business and Presentation layers are defined in the right manner. To do this, we 'll make use of several patterns and frameworks, some of those are presented below.

1. Entity Framework Code First development
2. Generic Repository Pattern
3. Dependency Injection using Autofac framework
4. Automapper

The architecture we will try to build in this post is summarized in the following diagram.

Let's start. Assuming we want to built an e-shop Web Application called "Store", create a blank solution with the same name.



Models

Add a class library project to the solution, named *Store.Model*. This library is where we 'll keep all of our domain objects. Entity Framework will count on them in order to build the database but we are not going to configure Code First using DataAnnotations attributes on this project. Instead, we are going to put all the Code First configuration in specific Configuration classes using the Fluent API. Add a folder named *Models* and add the following two simple classes.

Gadget.cs

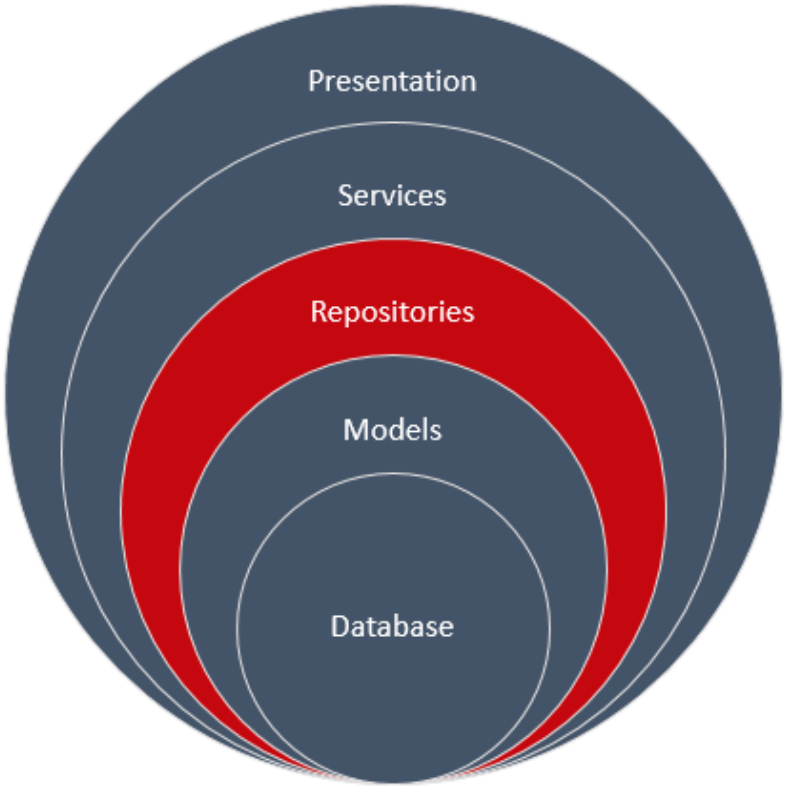
```
1  public class Gadget
2      {
3          public int GadgetID { get; set; }
4          public string Name { get; set; }
5          public string Description { get; set; }
6          public decimal Price { get; set; }
7          public string Image { get; set; }
8
9          public int CategoryID { get; set; }
10         public Category Category { get; set; }
11     }
```

Category.cs

```
1  public class Category
2      {
3          public int CategoryID { get; set; }
4          public string Name { get; set; }
5          public DateTime? DateCreated { get; set; }
6          public DateTime? DateUpdated { get; set; }
7
8          public virtual List<Gadget> Gadgets { get; set; }
9
10         public Category()
11         {
12             DateCreated = DateTime.Now;
13         }
14     }
```

I preffered to keep the namespace *Store.Model* instead of the namespace *Store.Model.Models*

Data Access Layer and Repositories



The purpose of this layer, is the direct access to the database. It's the only layer responsible to communicate with the database. If some other layer wants to access the database, then this will be done through some of the classes (repositories) we will define in this project. This will be strictly the **only** way Add a new class library project named *Store.Data* and make sure you add a reference to the previous created project, *Store.Model*. Install Entity Framework using the Nuget Package Manager. First thing we wanna do, is to define the Entity Type Configurations for our domain objecs. Add a folder named *Configuration* with the following two classes that inherits the EntityTypeConfiguration class.

GadgetConfiguration.cs

```
1  public class GadgetConfiguration: EntityTypeConfiguration<Gadget>
2      {
3          public GadgetConfiguration()
4          {
5              ToTable("Gadgets");
6              Property(g => g.Name).IsRequired().HasMaxLength(50);
7              Property(g => g.Price).IsRequired().HasPrecision(8, 2);
8              Property(g => g.CategoryID).IsRequired();
9          }
10     }
```

CategoryConfiguration.cs

```
1  public class CategoryConfiguration : EntityTypeConfiguration<Category>
2      {
3          public CategoryConfiguration()
4          {
5              ToTable("Categories");
6              Property(c => c.Name).IsRequired().HasMaxLength(50);
7          }
8      }
```

There isn't any difficult configuration to explain, the point is just to understand where to put the

right objects. The next thing we will do is to create the `DbContext` class that will be responsible to access the database. Add the following class under the root of the current project.

StoreEntities.cs

```
public class StoreEntities : DbContext
    {
        public StoreEntities() : base("StoreEntities") { }

        public DbSet<Gadget> Gadgets { get; set; }
        public DbSet<Category> Categories { get; set; }

        public virtual void Commit()
        {
            base.SaveChanges();
        }

        protected override void OnModelCreating(DbModelBuilder modelBuil
        {
            modelBuilder.Configurations.Add(new GadgetConfiguration());
            modelBuilder.Configurations.Add(new CategoryConfiguration())
        }
    }
```

We want to seed the database when our application fire up for the first time, so add the following class to the root of the project as well.

StoreSeedData

```
public class StoreSeedData : DropCreateDatabaseIfModelChanges<StoreEntit
    {
        protected override void Seed(StoreEntities context)
        {
            GetCategories().ForEach(c => context.Categories.Add(c));
            GetGadgets().ForEach(g => context.Gadgets.Add(g));

            context.Commit();
        }

        private static List<Category> GetCategories()
        {
            return new List<Category>
            {
                new Category {
                    Name = "Tablets"
                },
                new Category {
                    Name = "Laptops"
                },
                new Category {
                    Name = "Mobiles"
                }
            };
        }

        private static List<Gadget> GetGadgets()
        {
            return new List<Gadget>
            {
                new Gadget {
                    Name = "ProntoTec 7",
                    Description = "Android 4.4 KitKat Tablet PC, Cortex
```

```
                    CategoryID = 1,
                    Price = 46.99m,
                    Image = "prontotec.jpg"
                },
                new Gadget {
                    Name = "Samsung Galaxy",
                    Description = "Android 4.4 Kit Kat OS, 1.2 GHz quad-
                    CategoryID = 1,
                    Price = 120.95m,
                    Image= "samsung-galaxy.jpg"
                },
                new Gadget {
                    Name = "NeuTab® N7 Pro 7",
                    Description = "NeuTab N7 Pro tablet features the ama
                    CategoryID = 1,
                    Price = 59.99m,
                    Image= "neutab.jpg"
                },
                new Gadget {
                    Name = "Dragon Touch® Y88X 7",
                    Description = "Dragon Touch Y88X tablet featuring th
                    CategoryID = 1,
                    Price = 54.99m,
                    Image= "dragon-touch.jpg"
                },
                new Gadget {
                    Name = "Alldaymall A88X 7",
                    Description = "This Alldaymall tablet featuring the
                    CategoryID = 1,
                    Price = 47.99m,
                    Image= "Alldaymall.jpg"
                },
                new Gadget {
                    Name = "ASUS MeMO",
                    Description = "Pad 7 ME170CX-A1-BK 7-Inch 16GB Table
                    CategoryID = 1,
                    Price = 94.99m,
                    Image= "asus-memo.jpg"
                },
                // Code ommitted
            };
        }
    }
```

I have ommitted some of the Gadges objects for brevity but you can always download the solution project at the bottom of this post. Now let's create the *Heart* of this project. Add a folder named **Infrastructure**. In order to use the Repository Pattern in the right manner, we need to define a well designed infrastructure. From the bottom to the top all instances will be available through injected interfaces. And the first instance we will require, guess what.. will be an instance of the `StoreEntities`. So let's create a factory Interface responsible to initialize instances of this class. Add an interface named *IDbFactory* into the Infrastructure folder.

```
public interface IDbFactory : IDisposable
    {
        StoreEntities Init();
    }
```

You can see that this interface inherits the *IDisposable* one, so the Concrete class that will implement the IDbFactory interface, must also implement the IDisposable one. To do this in a

clean way, add a *Disposable* class that will implement the IDisposable interface. Then any class that will implement the IDbFactory interface, will just want to inherit this very class.

Disposable.cs

```
public class Disposable : IDisposable
    {
        private bool isDisposed;

        ~Disposable()
        {
            Dispose(false);
        }

        public void Dispose()
        {
            Dispose(true);
            GC.SuppressFinalize(this);
        }
        private void Dispose(bool disposing)
        {
            if (!isDisposed && disposing)
            {
                DisposeCore();
            }

            isDisposed = true;
        }

        // Ovveride this to dispose custom objects
        protected virtual void DisposeCore()
        {
        }
    }
```

I have highlighted the *DisposeCore* virtual method cause this method will make others classes inherit this one, to dispose their own objects in the way the need to. Now add the implementation class of the IDbFactory interface.

DbFactory.cs

```
public class DbFactory : Disposable, IDbFactory
    {
        StoreEntities dbContext;

        public StoreEntities Init()
        {
            return dbContext ?? (dbContext = new StoreEntities());
        }

        protected override void DisposeCore()
        {
            if (dbContext != null)
                dbContext.Dispose();
        }
    }
```

It's time to create a generic `IRepository` interface where we will declare the default operations that our repositories will support. Here I added some that i thought are the most used ones, but you can extend those operations as you wish.

IRepository.cs

```
public interface IRepository<T> where T : class
    {
        // Marks an entity as new
        void Add(T entity);
        // Marks an entity as modified
        void Update(T entity);
        // Marks an entity to be removed
        void Delete(T entity);
        void Delete(Expression<Func<T, bool>> where);
        // Get an entity by int id
        T GetById(int id);
        // Get an entity using delegate
        T Get(Expression<Func<T, bool>> where);
        // Gets all entities of type T
        IEnumerable<T> GetAll();
        // Gets entities using delegate
        IEnumerable<T> GetMany(Expression<Func<T, bool>> where);
    }
```

Notice that the **CRUD** operations are commented as *Mark to do something...* This means that when a repository implentation adds, updates or removes an entity, does not send the command to the database at that very moment. Instead, the caller (service layer) will be responsible to send a **Commit** command to the database through a IUnitOfWork injected instance. For this to be done will use a pattern called *UnitOfWork*. Add the following two files into the Infrastructure folder.

IUnitOfWork.cs

```
public interface IUnitOfWork
    {
        void Commit();
    }
```

UnitOfWork.cs

```
public class UnitOfWork : IUnitOfWork
    {
        private readonly IDbFactory dbFactory;
        private StoreEntities dbContext;

        public UnitOfWork(IDbFactory dbFactory)
        {
            this.dbFactory = dbFactory;
        }

        public StoreEntities DbContext
        {
            get { return dbContext ?? (dbContext = dbFactory.Init()); }
        }

        public void Commit()
        {
            DbContext.Commit();
        }
    }
```

In the same way we used the *Disposable* class we are going to use an abstract class that has **virtual** implementations of the `IRepository` interface. This **base** class will be inherited from all specific repositories and hence will implement the IRepository interface. Add the following class.

RepositoryBase.cs

```
public abstract class RepositoryBase<T> where T : class
    {
        #region Properties
        private StoreEntities dataContext;
        private readonly IDbSet<T> dbSet;

        protected IDbFactory DbFactory
        {
            get;
            private set;
        }

        protected StoreEntities DbContext
        {
            get { return dataContext ?? (dataContext = DbFactory.Init())
        }
        #endregion

        protected RepositoryBase(IDbFactory dbFactory)
        {
            DbFactory = dbFactory;
            dbSet = DbContext.Set<T>();
        }

        #region Implementation
        public virtual void Add(T entity)
        {
            dbSet.Add(entity);
        }

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

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

        public virtual void Delete(Expression<Func<T, bool>> where)
        {
            IEnumerable<T> objects = dbSet.Where<T>(where).AsEnumerable(
            foreach (T obj in objects)
                dbSet.Remove(obj);
        }

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

        public virtual IEnumerable<T> GetAll()
        {
            return dbSet.ToList();
        }

        public virtual IEnumerable<T> GetMany(Expression<Func<T, bool>>
        {
            return dbSet.Where(where).ToList();
        }

```

```
64          public T Get(Expression<Func<T, bool>> where)
65          {
66              return dbSet.Where(where).FirstOrDefault<T>();
67          }
68
69          #endregion
70
71      }
```

Since the implementations marked as virtual, any repository can ovveride a specific operation as required. And now the Concrete repositories: Add a new folder named `Repositories` and add the following two classes:

GadgetRepository.cs

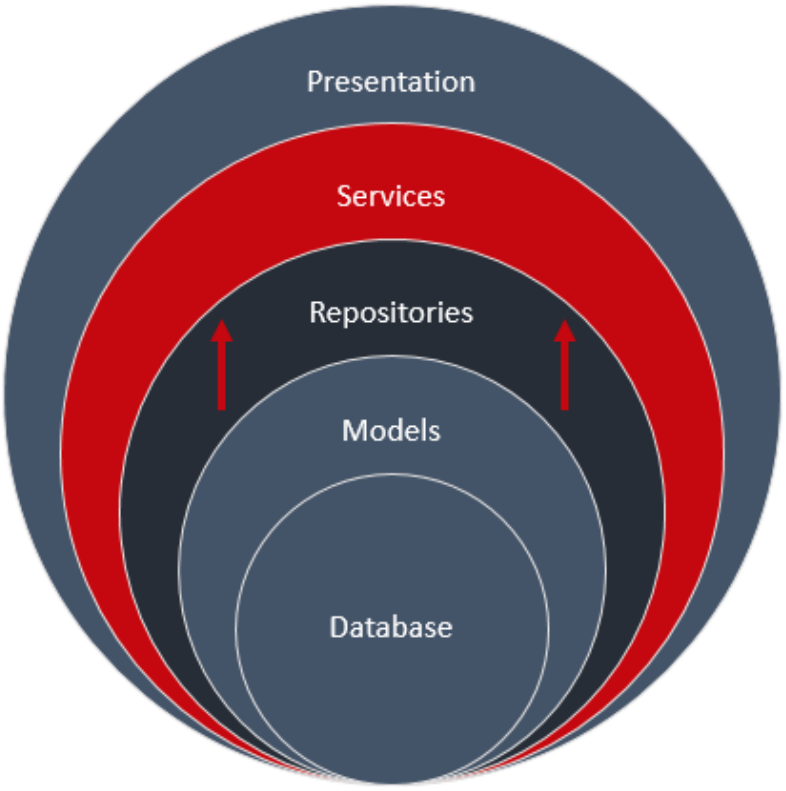
```
1   public class GadgetRepository : RepositoryBase<Gadget>, IGadgetRepositor
2       {
3           public GadgetRepository(IDbFactory dbFactory)
4               : base(dbFactory) { }
5       }
6
7       public interface IGadgetRepository : IRepository<Gadget>
8       {
9
10      }
```

CategoryRepository.cs

```
1   public class CategoryRepository : RepositoryBase<Category>, ICategoryRep
2       {
3           public CategoryRepository(IDbFactory dbFactory)
4               : base(dbFactory) { }
5
6           public Category GetCategoryByName(string categoryName)
7           {
8               var category = this.DbContext.Categories.Where(c => c.Name =
9
10              return category;
11          }
12
13          public override void Update(Category entity)
14          {
15              entity.DateUpdated = DateTime.Now;
16              base.Update(entity);
17          }
18      }
19
20      public interface ICategoryRepository : IRepository<Category>
21      {
22          Category GetCategoryByName(string categoryName);
23      }
```

You can see that the *GadgetRepository* supports the default operations using the default behavior and of course that's OK. On the other hand, you can see an example where a specific repository requires to either extend it's operations *(GetCategoryByName)* or overried the default ones *(Update)*. Usually you add a repository for each of your Model classes, hence each repository of type T, is responsible to manipulate a specific DbSet through the DbContext.Set<T>. We are done implementing the Data Access layer so we can procceed to the next one.

Service Layer



What operations do you want to expose your MVC Controllers? Where is the business logic is going to be implemented? Yeap.. you have guessed right, in this very layer. Add a new class library project named `Store.Service`. You will have to add references to the two previous created projects, *Store.Model* and *Store.Data*. Notice I haven't told you yet to install Entity Framework to this project.. and I am not going to, cause any database operation required will be done through the injected repositories we created before. Add the first Service file to this project.

GadgetService.cs

```
// operations you want to expose
    public interface IGadgetService
    {
        IEnumerable<Gadget> GetGadgets();
        IEnumerable<Gadget> GetCategoryGadgets(string categoryName, stri
        Gadget GetGadget(int id);
        void CreateGadget(Gadget gadget);
        void SaveGadget();
    }

    public class GadgetService : IGadgetService
    {
        private readonly IGadgetRepository gadgetsRepository;
        private readonly ICategoryRepository categoryRepository;
        private readonly IUnitOfWork unitOfWork;

        public GadgetService(IGadgetRepository gadgetsRepository, ICateg
        {
            this.gadgetsRepository = gadgetsRepository;
            this.categoryRepository = categoryRepository;
            this.unitOfWork = unitOfWork;
        }

        #region IGadgetService Members

        public IEnumerable<Gadget> GetGadgets()
        {
            var gadgets = gadgetsRepository.GetAll();
```

```
            return gadgets;
        }

        public IEnumerable<Gadget> GetCategoryGadgets(string categoryNam
        {
            var category = categoryRepository.GetCategoryByName(category
            return category.Gadgets.Where(g => g.Name.ToLower().Contains
        }

        public Gadget GetGadget(int id)
        {
            var gadget = gadgetsRepository.GetById(id);
            return gadget;
        }

        public void CreateGadget(Gadget gadget)
        {
            gadgetsRepository.Add(gadget);
        }

        public void SaveGadget()
        {
            unitOfWork.Commit();
        }

        #endregion

    }
```

The first and the last highlighted code lines reminds you why we created the IUnitOfWork interface. If we wanted to create a gadget object though this service class, we would write something like this..

```
// init a gadget object..
gadgetService.CreateGadget(gadget);
gadgetService.SaveGadget();
```

The other highlighted code lines denotes that any required repository for this service, will be injected through it's constructor. This will be done through a Dependency Container we will setup in the MVC project's start up class, using the *Autofac* framework. In the same way I created the *GadgetService.cs* file.

CategoryService.cs

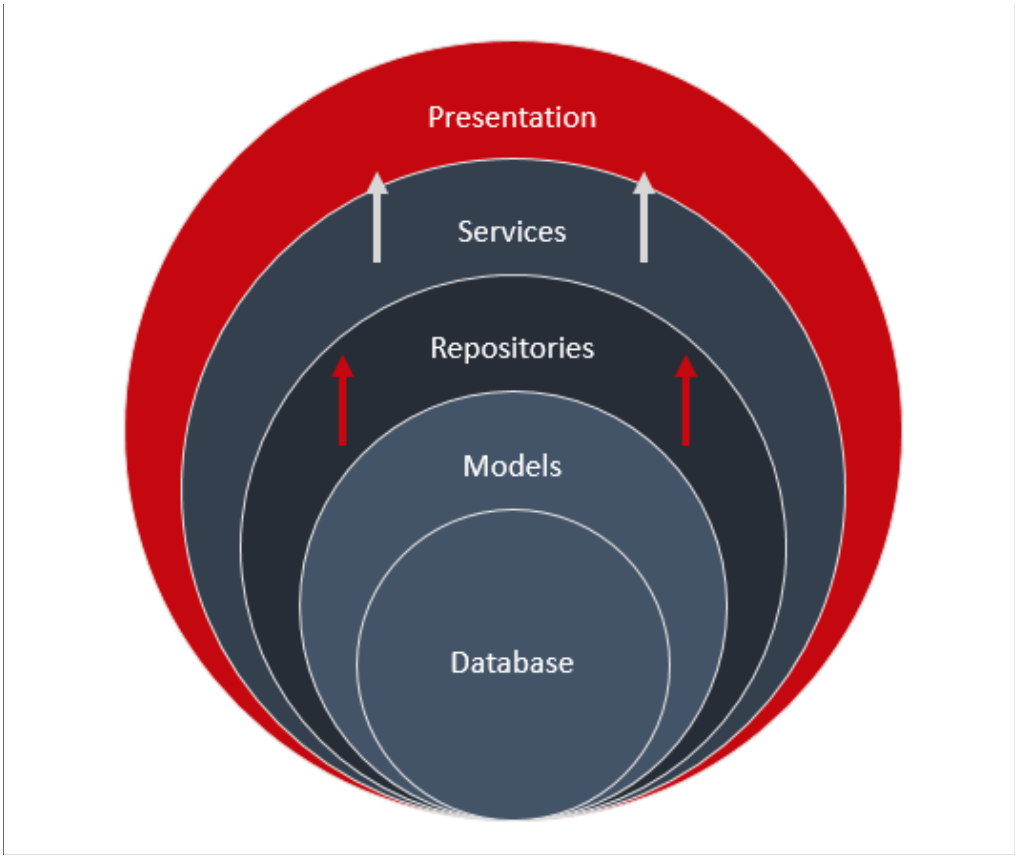
```
// operations you want to expose
    public interface ICategoryService
    {
        IEnumerable<Category> GetCategories(string name = null);
        Category GetCategory(int id);
        Category GetCategory(string name);
        void CreateCategory(Category category);
        void SaveCategory();
    }

    public class CategoryService : ICategoryService
    {
        private readonly ICategoryRepository categorysRepository;
        private readonly IUnitOfWork unitOfWork;

        public CategoryService(ICategoryRepository categorysRepository, I
        {
```

```
18              this.categorysRepository = categorysRepository;
19              this.unitOfWork = unitOfWork;
20          }
21
22          #region ICategoryService Members
23
24          public IEnumerable<Category> GetCategories(string name = null)
25          {
26              if (string.IsNullOrEmpty(name))
27                  return categorysRepository.GetAll();
28              else
29                  return categorysRepository.GetAll().Where(c => c.Name ==
30          }
31
32          public Category GetCategory(int id)
33          {
34              var category = categorysRepository.GetById(id);
35              return category;
36          }
37
38          public Category GetCategory(string name)
39          {
40              var category = categorysRepository.GetCategoryByName(name);
41              return category;
42          }
43
44          public void CreateCategory(Category category)
45          {
46              categorysRepository.Add(category);
47          }
48
49          public void SaveCategory()
50          {
51              unitOfWork.Commit();
52          }
53
54          #endregion
55      }
```

And we are done with the service layer as well. Let's procceed with the last one, the ASP.NET MVC Web Application.

Presentation Layer

Add a new ASP.NET Web Application named `Store.Web` choosing the empty template with the MVC option checked. We need to add references to all of the previous class library projects and an Entity Framework installation via Nuget Packages as well. You may be wondering, are we going to write any Entity Framework related queries in this project? Not at all, we need though some of it's namespaces so we can setup the database configurations for our application, such as the `database initializer`. And since we started with this, open *Global.asax.cs* file and add the following line of code to setup the seed initializer we created in the *Store.Data* project.



Glbal.asax.cs

```
protected void Application_Start()
        {
            // Init database
            System.Data.Entity.Database.SetInitializer(new StoreSeedData(

            AreaRegistration.RegisterAllAreas();
            RouteConfig.RegisterRoutes(RouteTable.Routes);
        }
```

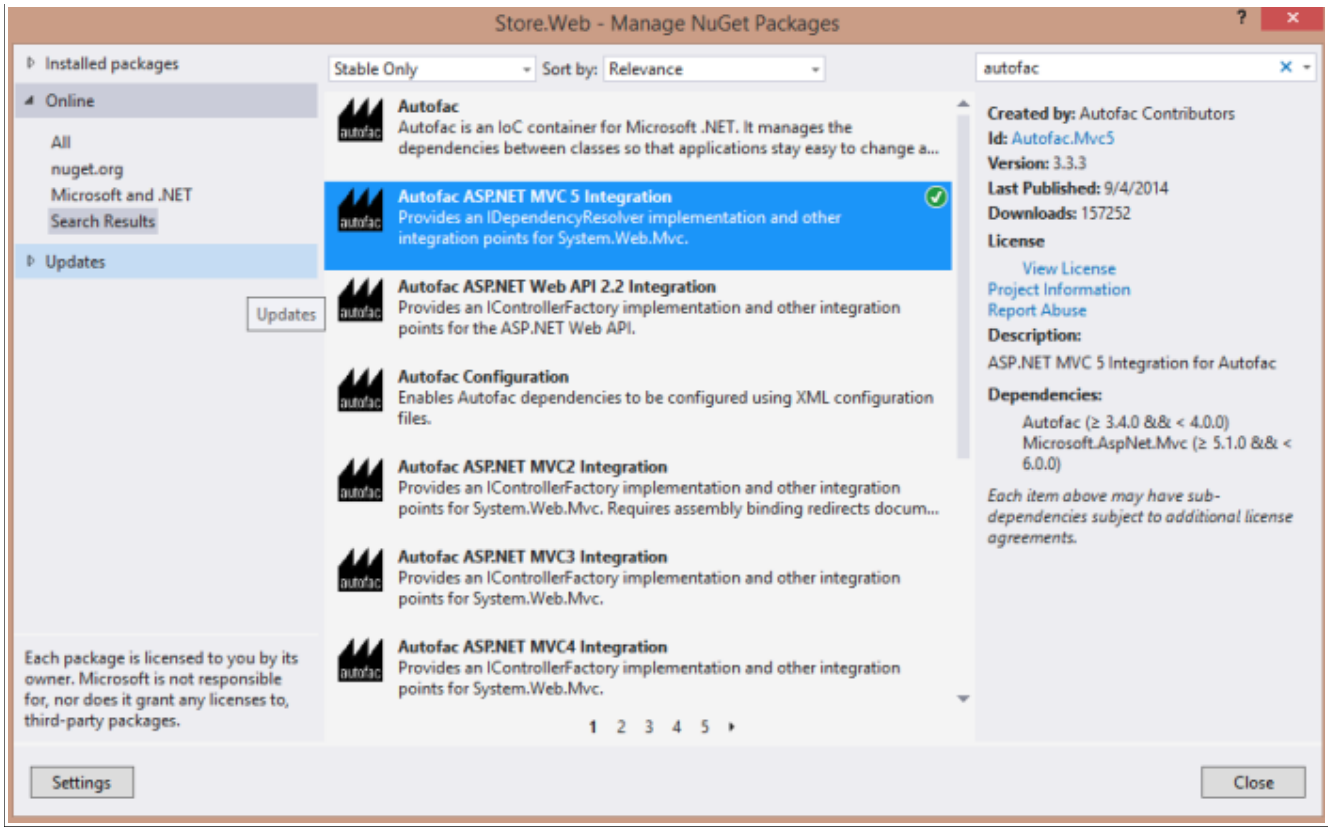
You will also need to create a connection string element to define where you want the database to be created. Add the following element in the Web.config file and changed it accoarding to your development enviroment requirements.

Web.config

```
<connectionStrings>
  <add name="StoreEntities" connectionString="Data Source=(localdb)\v11.0
</connectionStrings>
```

We have made such an effort in the previous steps to create repositories and services but now it's the time to make them work all together. If you recall, all services contructors have repositories interfaces that must be injected to. The services themeselfs will later be injected in to the controllers conustructors and this is how our application will work. To achieve this, we need to setup Dependancy Injection and for thish reason I decided to use **Autofac**. Make sure you install Autofac ASP.NET MVC 5 Integration through Nuget Packages.

Create a *Bootstrapper.cs* file under the Start_App folder and paste the following code.



Bootstrapper.cs

```
1   public static void Run()
2           {
3               SetAutofacContainer();
4           }
5
6           private static void SetAutofacContainer()
7           {
8               var builder = new ContainerBuilder();
9               builder.RegisterControllers(Assembly.GetExecutingAssembly())
10              builder.RegisterType<UnitOfWork>().As<IUnitOfWork>().Instanc
11              builder.RegisterType<DbFactory>().As<IDbFactory>().InstanceP
12
13              // Repositories
14              builder.RegisterAssemblyTypes(typeof(GadgetRepository).Assem
15                  .Where(t => t.Name.EndsWith("Repository"))
16                  .AsImplementedInterfaces().InstancePerRequest();
17              // Services
18              builder.RegisterAssemblyTypes(typeof(GadgetService).Assembly
19                 .Where(t => t.Name.EndsWith("Service"))
20                 .AsImplementedInterfaces().InstancePerRequest();
21
22              IContainer container = builder.Build();
23              DependencyResolver.SetResolver(new AutofacDependencyResolver
24          }
25      }
```

The code itself is self explanatory. I hope you have followed along with me and you have named your repository and service classes as I did, cause otherwise, this is not gonna work. There are two importants things left to complete the tutotiral. The first one to define ViewModel classes and set Automapper to map domain entities to viewmodels and backwards. The second one is to see how to setup CSS Bootstrap in our web application. I suppose most of you, install bootstrap from Nuget Packages and start adding css and script references to your project. Here though we will

follow a different approach.

CSS Bootstrap

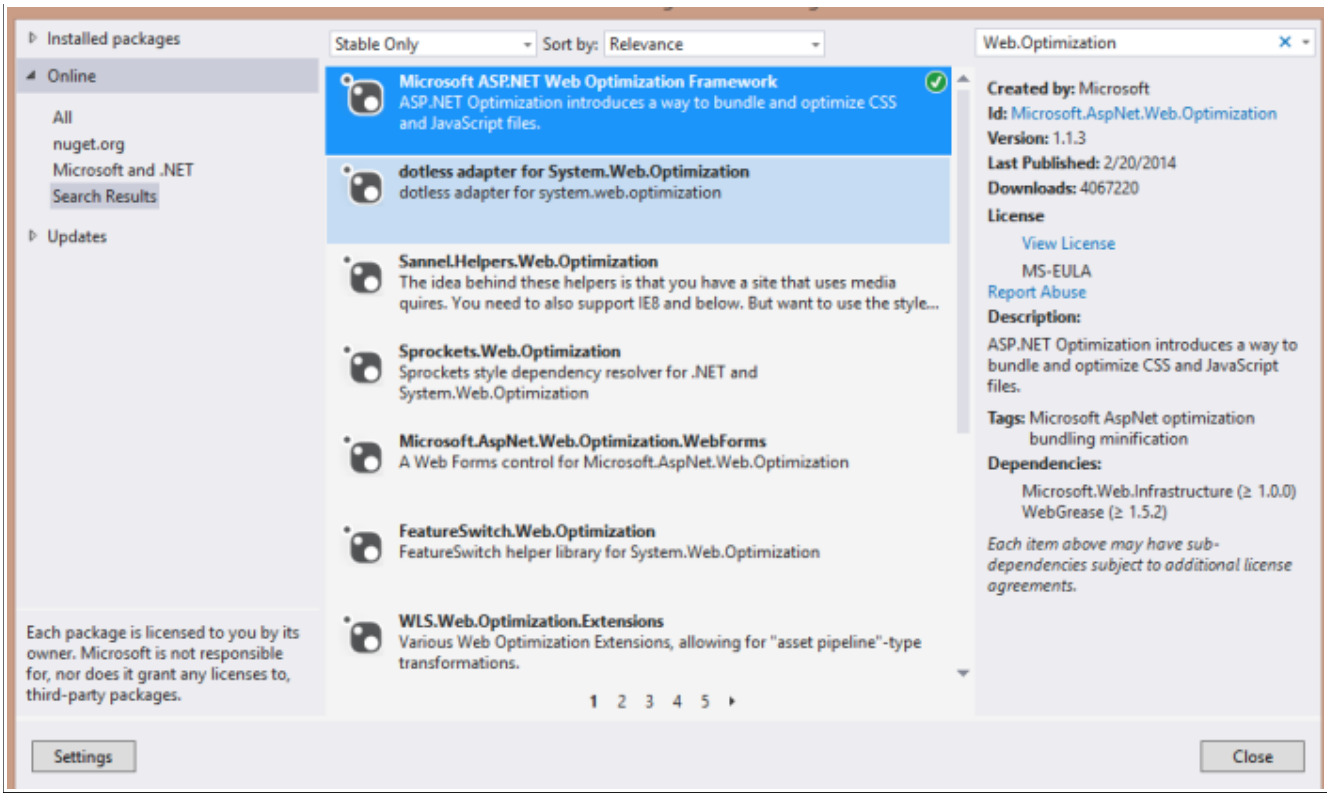
First of all download Boostrap distribution from the official site. Add three folders to your application named css, fonts and js respectively. In the *css* folder paste the bootstrap.css file from what you have downloaded, in the fonts folder paste everything is inside the respective fonts folder and in the js folder, just paste the bootstrap.js file. We are going to use **Bundling and Minification** for bootstrap and to achieve that you need to install Microsoft ASP.NET Web Optimazation Framework through Nuget Packages.

When you finish installing this, add a new class named *BundleConfig* into the App_Start folder as follow:

BundleConfig.cs

```
1  public class BundleConfig
2      {
3          public static void RegisterBundles(BundleCollection bundles)
4          {
5              bundles.Add(new ScriptBundle("~/bootstrap/js").Include("~/js
6              bundles.Add(new StyleBundle("~/bootstrap/css").Include("~/cs
7
8              BundleTable.EnableOptimizations = true;
9          }
10     }
```

As you can see I have also referenced *site.js* and *site.css* javascript and css files. Those files can host any bootstrap css customazations you wanna do or any javascript related code. Feel free to add the respective files and leave them empty. Now we need to declare that we want MVC to use



bundling and minication, so add the following line into the Global.asax.cs file.

Global.asax.cs

```
protected void Application_Start()
        {
            // Init database
            System.Data.Entity.Database.SetInitializer(new StoreSeedData

            AreaRegistration.RegisterAllAreas();
            RouteConfig.RegisterRoutes(RouteTable.Routes);
            BundleConfig.RegisterBundles(BundleTable.Bundles);

            // Autofac and Automapper configurations
            Bootstrapper.Run();
        }
```

Notice that I have also called the **Bootstrapper.Run()** function that will setup the Autofac's configuration we made before. This function will also configure Automapper, something we gonna see in a bit. Let's finish with Bootrap for now. We will need a Layout to use for our application, so go and create a *Shared* folder under the Views folder and add a new item of type MVC 5 Layout Page (Razor) named *_Layout.cshtml*.

_Layout.cshtml

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>@ViewBag.Title</title>
    <!-- Bootstrap -->
    @Styles.Render("~/bootstrap/css")
    <!--[if lt IE 9]>
    <script src="https://oss.maxcdn.com/libs/html5shiv/3.7.0/
    html5shiv.js"></script>
    <script src="https://oss.maxcdn.com/libs/respond.js/1.4.2/
    respond.min.js"></script>
    <![endif]-->
</head>
<body>
    <nav id="myNavbar" class="navbar navbar-default navbar-inverse navba
        <!-- Brand and toggle get grouped for better mobile display -->
            <div class="navbar-header">
                <button type="button" class="navbar-toggle" data-toggle=
                    <span class="sr-only">Toggle navigation</span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                @Html.ActionLink("Store", "Index", "Home", new { }, new
            </div>
            <!-- Collect the nav links, forms, and other content for tog
            <div class="collapse navbar-collapse" id="navbarCollapse">
                <ul class="nav navbar-nav">
                    <li class="active">
                        @Html.ActionLink("Tablets", "Index", "Home", new
                    </li>
                    <li class="active">
                        @Html.ActionLink("Laptops", "Index", "Home", new
```

```
                    </li>
                    <li class="active">
                        @Html.ActionLink("Mobiles", "Index", "Home", new
                    </li>
                </ul>
            </div>
        </nav>
        @RenderBody()
        <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.0/jquery.
        @Scripts.Render("~/bootstrap/js")
    </body>
    </html>
```

The page will probably complain that cannot resolve Razor syntax so you have to add the following using statement in the **web.config** file which is under the Views folder (not application's web.config). Following is part of that file..

web.config

```
<namespaces>
        <add namespace="System.Web.Mvc" />
        <add namespace="System.Web.Mvc.Ajax" />
        <add namespace="System.Web.Mvc.Html" />
        <add namespace="System.Web.Routing" />
        <add namespace="Store.Web" />
        <add namespace="System.Web.Optimization" />
      </namespaces>
```

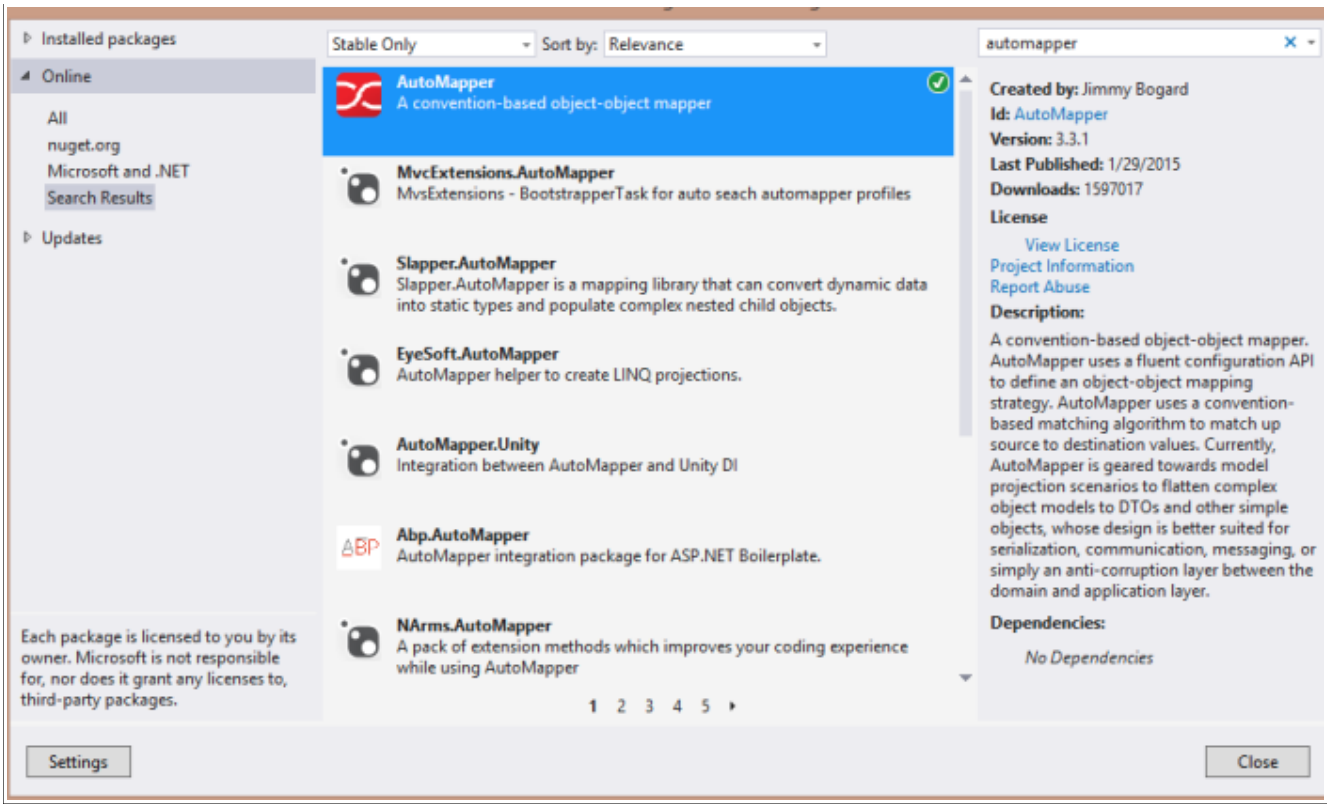
Automapper

In a real application your domain objects will probably have a lot of properties but you only want to display some of them in the browser. More over when posting back to server, for example when creating objects though a *form* element, you also want to post only a few of the domain object's properties. For this reason you define **ViewModel** objects and use them instead of the real domain ones. Make sure you install Automapper from Nuget Packages.

Add a new folder named *ViewModels* with the following classes.

GadgetViewModel.cs

```
public class GadgetViewModel
    {
        public int GadgetID { get; set; }
        public string Name { get; set; }
        public string Description { get; set; }
        public decimal Price { get; set; }
        public string Image { get; set; }

        public int CategoryID { get; set; }
    }
```



CategoryViewModel.cs

```
public class CategoryViewModel
    {
        public int CategoryID { get; set; }
        public string Name { get; set; }

        public List<GadgetViewModel> Gadgets { get; set; }
    }
```

GadgetFormViewModel.cs

```
public class GadgetFormViewModel
    {
        public HttpPostedFileBase File { get; set; }
        public string GadgetTitle { get; set; }
        public string GadgetDescription { get; set; }
        public decimal GadgetPrice { get; set; }
        public int GadgetCategory { get; set; }
    }
```

When your ViewModel classes have properties named as the respective domain objects, Automapper is smart enough to make the mapping through default conventions. Otherwise you have to set the mapping manualy by yourself. Notice the last class I have added, the *GadgetFormViewModel*. We can make a convetion to add a “Form” word before “ViewModel” so that we know that this type of view model, is posted back to server through a form element. Let’s now configure the mappings. Add a new folder Mappings and add the following class file.

AutoMapperConfiguration.cs

```
public class AutoMapperConfiguration
    {
        public static void Configure()
        {
            Mapper.Initialize(x =>
            {
```

```
                x.AddProfile<DomainToViewModelMappingProfile>();
                x.AddProfile<ViewModelToDomainMappingProfile>();
            });
        }
    }
```

We haven't created the required profiles yet but we will in a bit. What I wanted to show you is that you can create as many Automapper profiles you want and then add them into **Mapper.Initialize** function. Here we will define two profiles, one to map domain models to ViewModels and another one for backwards. Add the following classes in the same folder as the previous.

DomainToViewModelMappingProfile.cs

```
public class DomainToViewModelMappingProfile : Profile
    {
        public override string ProfileName
        {
            get { return "DomainToViewModelMappings"; }
        }

        protected override void Configure()
        {
            Mapper.CreateMap<Category,CategoryViewModel>();
            Mapper.CreateMap<Gadget, GadgetViewModel>();
        }
    }
```

ViewModelToDomainMappingProfile.cs

```
public class ViewModelToDomainMappingProfile : Profile
    {
        public override string ProfileName
        {
            get { return "ViewModelToDomainMappings"; }
        }

        protected override void Configure()
        {
            Mapper.CreateMap<GadgetFormViewModel, Gadget>()
                .ForMember(g => g.Name, map => map.MapFrom(vm => vm.Gadg
                .ForMember(g => g.Description, map => map.MapFrom(vm =>
                .ForMember(g => g.Price, map => map.MapFrom(vm => vm.Gad
                .ForMember(g => g.Image, map => map.MapFrom(vm => vm.Fil
                .ForMember(g => g.CategoryID, map => map.MapFrom(vm => v
        }
    }
```

For the Domain -> ViewModels mapping we didn't need to setup anything. Automapper will use the default conventions and that's fine. For our GadgetFormViewModel -> Gadget mapping though, we set manually the configuration as shown above. The last thing remained to finish with Automapper is to add the following line in the Bootstrapper class.

Bootsrapper.cs

```
public static class Bootstrapper
    {
        public static void Run()
        {
            SetAutofacContainer();
            //Configure AutoMapper
```

```
            AutoMapperConfiguration.Configure();
        }
// Code ommitted
```

Controllers and Views

We are almost finished. Add a new MVC Controller named *HomeController* and paste the following code.

HomeController.cs

```
public class HomeController : Controller
    {
        private readonly ICategoryService categoryService;
        private readonly IGadgetService gadgetService;

        public HomeController(ICategoryService categoryService, IGadgetS
        {
            this.categoryService = categoryService;
            this.gadgetService = gadgetService;
        }

        // GET: Home
        public ActionResult Index(string category = null)
        {
            IEnumerable<CategoryViewModel> viewModelGadgets;
            IEnumerable<Category> categories;

            categories = categoryService.GetCategories(category).ToList(

            viewModelGadgets = Mapper.Map<IEnumerable<Category>, IEnumer
            return View(viewModelGadgets);
        }
    }
```

Now you can see in action why we have made such an effort to setup Repositories, Services, Autofac and Automapper. Services will be injected in the controller for each request and their data will be mapped to ViewModels before send to the Client. Right click in the Index action and add a View named Index with the following code. I must mention here, that the gadgets objects we use, have image references to a folder named *images* in the Web Application project. You can use your images or just download this project at the end of this post.

Views/Home/Index.cshtml

```
@model IEnumerable<Store.Web.ViewModels.CategoryViewModel>

@{
    ViewBag.Title = "Store";
    Layout = "~/Views/Shared/_Layout.cshtml";
}

<p>

</p>
<div class="container">
    <div class="jumbotron">

        @foreach (var item in Model)
        {
            <div class="panel panel-default">
```

```
                <div class="panel-heading">
                    @*@Html.DisplayFor(modelItem => item.Name)*@
                    @Html.ActionLink("View all " + item.Name, "Index", n
                    @using (Html.BeginForm("Filter", "Home", new { categ
                    {
                        @Html.TextBox("gadgetName", null, new { @class =
                    }


                </div>
                @foreach (var gadget in item.Gadgets)
                {
                    @Html.Partial("Gadget", gadget)
                }
                <div class="panel-footer">
                    @using (Html.BeginForm("Create", "Home", FormMethod.
                            new { enctype = "multipart/form-data", @clas
                    {
                        @Html.Hidden("GadgetCategory", item.CategoryID)
                        <div class="form-group">
                            <label class="sr-only" for="file">File</labe
                            <input type="file" class="form-control" name
                        </div>
                        <div class="form-group">
                            <label class="sr-only" for="gadgetTitle">Tit
                            <input type="text" class="form-control" name
                        </div>
                        <div class="form-group">
                            <label class="sr-only" for="gadgetName">Pric
                            <input type="number" class="form-control" na
                        </div>
                        <div class="form-group">
                            <label class="sr-only" for="gadgetName">Desc
                            <input type="text" class="form-control" name
                        </div>
                        <button type="submit" class="btn btn-primary">Up
                    }
                </div>
            </div>
        }

    </div>

</div>
```

Two things to notice here. The first one is that we need to create a Partial view to diplay a `GadgetViewModel` object and the second one is the Form's control element's names. You can see that they much our *GadgetFormViewModel* properties. Under the Shared folder create the following Partial view for displaying a GadgetViewModel object.

Views/Shared/Gadget.cshtml

```
@model Store.Web.ViewModels.GadgetViewModel

<div class="panel-body">
    <div class="media">
        <a class="pull-left" href="#">
            <img class="media-object" src="../../images/@Model.Image" />
        </a>
        <div class="media-body">
            <h3 class="media-heading">
                @Model.Name
```

```
            </h3>
            <p>@Model.Description</p>
        </div>
    </div>
</div>
```

In the *Index.cshtml* page I have added search and filter functionality and Create gadget as well. To achieve that you need to add the following Action methods to the HomeController.

HomeController.cs

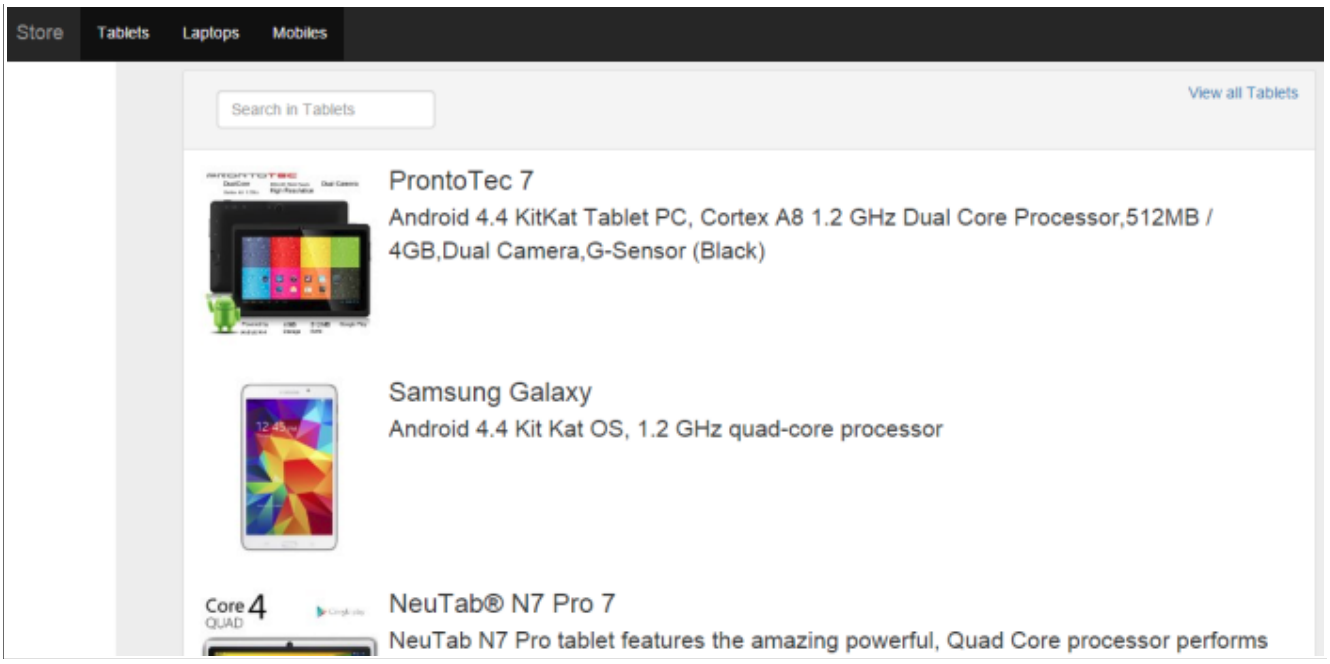
```
public ActionResult Filter(string category, string gadgetName)
        {
            IEnumerable<GadgetViewModel> viewModelGadgets;
            IEnumerable<Gadget> gadgets;

            gadgets = gadgetService.GetCategoryGadgets(category, gadgetN

            viewModelGadgets = Mapper.Map<IEnumerable<Gadget>, IEnumerab

            return View(viewModelGadgets);
        }

        [HttpPost]
        public ActionResult Create(GadgetFormViewModel newGadget)
        {
            if (newGadget != null && newGadget.File != null)
            {
                var gadget = Mapper.Map<GadgetFormViewModel, Gadget>(new
                gadgetService.CreateGadget(gadget);

                string gadgetPicture = System.IO.Path.GetFileName(newGad
                string path = System.IO.Path.Combine(Server.MapPath("~/i
                newGadget.File.SaveAs(path);

                gadgetService.SaveGadget();
            }

            var category = categoryService.GetCategory(newGadget.GadgetC
            return RedirectToAction("Index", new { category = category.N
        }
```

I am sure that at this point you understand the purpose of all the above code so I won't explain anything. You need to add a *Filter* page so right click in the Filter action and create the following View.

Home/Views/Filter.cshtml

```
@model IEnumerable<Store.Web.ViewModels.Gadget

@{
    ViewBag.Title = "Filter";
    Layout = "~/Views/Shared/_Layout.cshtml";
}

<div class="container">
    <div class="jumbotron">

        @foreach (var item in Model)
        {
```

```
13                 <div class="panel panel-default">
14                     <div class="panel-heading">
15                         @Html.Label(item.Name)
16                     </div>
17                     @Html.Partial("Gadget", item)
18                 </div>
19             }
20
21         </div>
22     </div>
```

You can filter gadgets by category or search gadgets in a specific category. That's it, we have finished creating a highly decoupled architecture that could support large scale MVC applications.

Github

I have decided to move all of my code to my Github account so you can download most of the projects that we have seen on this blog from there. You can download this project from here. I hope you enjoyed this post as much as I did.

In case you find my blog's content interesting, register your email to receive notifications of new posts and follow *chsakell's Blog* on its Facebook or Twitter accounts.

91 replies

Lang

April 18, 2016 • 5:07 pm

Thank you for this article.
I have a question. In service layer, Repository and UnitOfWork have their own dbContext, so how can the UnitOfWork commit the session that was handled by the Repository?

Christos S.

April 18, 2016 • 5:34 pm

Good question. It's actually the same *dbContext* instance that comes from the unique injected DbFactory instance per request. Chears to Autofac!

part of Bootstrapper.cs

```
1  var builder = new ContainerBuilder();
2  builder.RegisterControllers(Assembly.GetExecutingAssembly());
3  builder.RegisterType<UnitOfWork>().As<IUnitOfWork>().InstancePerReques
4  builder.RegisterType<DbFactory>().As<IDbFactory>().InstancePerRequest(
```

Lang

April 19, 2016 • 2:12 am

Thanks a lot, now I understand. Have a nice day sir.

subramanihm

May 22, 2016 • 6:54 pm

> Very nice article

zicki

April 20, 2016 • 4:17 am

Thanks for your awesome article!

I'm currently setting up my third MVC Project using your tutorial 🙂

it was really easy to follow your guide and to adapt it for my requirements!

David Cherian

May 4, 2016 • 6:56 pm

Thanks a lot for this excellent article.
I'll be using this architecture style in my next MVC project – really committed to you for sharing this sample.

Tom Frajer

May 5, 2016 • 8:58 am

Can I use Dapper instead of Entity Framework in this architecture?

> 
>
> **Christos S.**
>
> May 5, 2016 • 9:08 am
>
> Of course.

Viet hoang

May 6, 2016 • 4:40 am

Hi,
I have a question about DI.
How to know which classes will be injected the objects? We need to config this or all of classes that has constructor with registered object parameter will be injected?

Arash Motamedi

May 7, 2016 • 7:46 pm

Great post! Thank you Christos. One suggestion: took me a little while to figure out why my Web Api controllers were throwing a “default constructor not found” exception and realize that a separate Autofac dependency and a couple extra lines of configuration were needed. Maybe mention somewhere that if the dev likes to include Web Api controllers in their project, they need to reference Autofac.WebApi2 and then in the Bootstrapper, RegisterApiControllers() and also set the GlobalConfiguration.Configuration.DependencyResolver to Autofac Web Api resolver.

Amit Jain

May 11, 2016 • 10:06 am

Really a great article!!

Amit Jain

May 11, 2016 • 10:16 am

For the icing on the cake, we can do small addition to this solution i.e. Data hiding Approach. We can hide the reference of Models in the Web Project

---

Mathias

May 23, 2016 • 5:49 pm

Simply brilliant. A good way for newcommers to asp.net to dive into architectures! Thank you!

hove1543

May 23, 2016 • 5:52 pm

Brilliant! Thank you very much for this guide! Great for newcomers (as myself) to learn more about architecting an asp.net mvc app.

bhavesh

May 25, 2016 • 2:37 pm

Nice Article easy to understand

ctysingh

May 30, 2016 • 9:05 pm

Reblogged this on **chaitanyasingh**.

Manoj

June 4, 2016 • 12:45 pm

Thank you sir for this article. Really a great article. It is easy to understand

---

Mohammed Nazim Feroz

June 6, 2016 • 3:17 pm

Hi Christos,
Thanks for this amazing post! Can you please let me know if you have an example of using FluentValidation on the 'FormViewModel' classes? Thanks.

‹ Older Comments

Trackbacks

1. Asp Net Mvc Architecture Best Practices | New Architecture Fan