Adrian Ilewicz

Komsky Inc is a fictitious technology company based in London, UK. We specialize in producing IoT devices for board market

Komsky Ltd

ASP.NET MVC 5 with Development Patterns Training

LAB OVERVIEW

# Introduction

KOMSKY Ltd is a technology company producing IoT devices for the market. As a Web Developer, you will create an internal ticketing tool to solve cases.

# Use Cases

We have customers and they need a web interface where they submit their cases to Komsky to be resolved. Each case should have Case title, state, description, assigned agent, priority.

Below are the use cases that this application should cater for. However, you can make assumptions which are in line with the requirements.

* Each user belongs to Company
* Customer users shall be able to create cases
* Customer users shall be able to view their own company's cases
* Agents shall be able reply to customer cases
* Agents shall be able to pick cases from case pool and assign them to themselves.
* Agents can reply to any case but each case should only have one owner
* Admins shall be able to create and edit new Customers
* Admins shall be able to create users for Customers
* Admins shall be able to create agents
* Cases shall have discussion board where Customer users and agents discuss the cases
* Only authorized users shall be able to login to portal

# DEVELOPMENT METHODOLOGY

As with SCRUM and AGILE methodology, we want to deliver usable software, even if the only thing it will do is display a welcome page, create a ticket/case or allow user to log in. Within this iterative process we don’t always follow the shortest path to ultimate success, but rather a path that leads to local success, that is working application. Within, the path itself, utilizes workflow that emphasizes and explains design decisions in practise.

# Github REPOSIOTORY

I use GitHub to store materials for this training (Projects, Lab documents and PPTX). It’s publicly available here

<https://github.com/komsky/Reporting.git>

# CONVENTIONS

**Bold** – project names (sometimes with paths) Consolas – Classes, Properties, Fields and other pieces of code

Revision: A

January 2016

All rights reseved

Lab 01

# Content:

Out target for this lab is to create a layered solution of projects to minimize refactoring of code later on. The three main layers of the project are going to be: Web (Interface Layer), Domain (Service Layer)

# General steps:

1. Create container solution with starter MVC 5 Web Application Project
2. Create specific project for solution layers used for patterns
3. Refactor solution to create repository pattern
4. Refactor solution to use unit of work / data façade pattern
5. Remove dependencies from home controller and test solution

## Task 1 – CREATE SAMPLE WEB APPLICATION USING MVC 5 TEMPLATE

1. You might want to create your own repository for your solution in GIT or other version control system
2. Create new ASP.NET Application
   1. Name: **Komsky.Web**
   2. Solution: **Komsky**
   3. Add unit tests project to your solution
   4. Note the default authentication mode: Individual User Accounts
   5. Leave other options as default
3. Run the project and make sure you are able to register, log in and log out.

At this point you should have working sample MVC application using standard template.

## Task 2 - setup layers of your project

1. Add to your solution following projects:
   1. Kind: Class Library | Name: **Komsky.Services**
   2. Kind: Class Library | Name: **Komsky.Domain**
   3. Kind: Class Library | Name: **Komsky.Data.DataAccess**
   4. Kind: Class Library | Name: **Komsky.Data.Entities**
   5. Further layer solution by creating folder Data and move two data projects inside this folder
   6. Further layer solution by creating folder Services and move Services and Domain projects inside this folder
   7. Create folder Tests and move there **Komsky.Web.Test** project
2. Copy file **IdentityModels.cs** from **Komsky.Web/Models** to **Komsky.Data.DataAccess**
3. Delete original **IdentityModels.cs** file from Web project
4. Copy file **IdentityModels.cs** to **Komsky.Data.Entities** project
   1. Rename this file to ApplicationUser
   2. Delete ApplicationDbContext class
   3. Update namespace inside **ApplicationUser.cs** to resemble current file location
5. Go back to **IdentityModels.cs** inside **Komsky.Data.DataAccess** project
   1. Rename this file to ApplicationDbContext
   2. Delete ApplicationUser class
   3. Update namespace inside **ApplicationDbContext.cs** to resemble current file location
6. Install nugget Microsoft.AspNet.Identity.EntityFramework for **Komsky.Data.DataAccess** and **Komsky.Data.Entities** using Package Manager console with following instruction:

PM> install-package Microsoft.AspNet.Identity.EntityFramework

1. Add **Komsky.Data.Entities** reference to **Komsky.Data.DataAccess** and to **Komsky.Web**
2. Add **Komsky.Data.DataAccess** reference to **Komsky.Web**
3. Update using clauses in solution
4. Build and run your solution, debug any errors.

At this point you should have layered solution, ready to code refactoring. We need to make further changes to implement Repository and Unit of Work patterns

## Task 3 – ADD REPOSITORY PATTERN TO DATA ACCESS LAYER

1. Create new project **Komsky.Data** in the **Data** folder
2. Add **Repositories** folder to **Komsky.Data** project
3. Add new interface IRepository<T> to Repositories folder. The code for this interface is provided below.

public interface IRepository<T> where T : class

{

IQueryable<T> GetAll();

T GetById(int id);

T GetById(Guid id);

T GetById(string id);

void Add(T entity);

void Update(T entity);

void Delete(T entity);

void Delete(int id);

void Delete(Guid id);

void Delete(string id);

}

1. Add GenericRepository class to your Data Access project
   1. Add folder Repositories
   2. Add new class Generic Repository<T> to this folder. The code for GenericRepository class is provided below.

public class GenericRepository<T> : IDisposable, IRepository<T> where T : class

{

public GenericRepository(DbContext dbContext)

{

if (dbContext == null)

{

throw new ArgumentNullException("dbContext");

}

DbContext = dbContext;

DbSet = DbContext.Set<T>();

}

protected DbContext DbContext { get; set; }

protected DbSet<T> DbSet { get; set; }

public virtual IQueryable<T> GetAll()

{

return DbSet;

}

public virtual T GetById(int id)

{

return DbSet.Find(id);

}

public virtual T GetById(Guid id)

{

return DbSet.Find(id);

}

public virtual T GetById(string id)

{

return DbSet.Find(id);

}

public virtual void Add(T entity)

{

DbEntityEntry dbEntityEntry = DbContext.Entry(entity);

if (dbEntityEntry.State != EntityState.Detached)

{

dbEntityEntry.State = EntityState.Added;

}

else

{

DbSet.Add(entity);

}

}

public virtual void Update(T entity)

{

DbEntityEntry dbEntityEntry = DbContext.Entry(entity);

if (dbEntityEntry.State == EntityState.Detached)

{

DbSet.Attach(entity);

}

dbEntityEntry.State = EntityState.Modified;

}

public virtual void Delete(T entity)

{

DbEntityEntry dbEntityEntry = DbContext.Entry(entity);

if (dbEntityEntry.State != EntityState.Deleted)

{

dbEntityEntry.State = EntityState.Deleted;

}

else

{

DbSet.Attach(entity);

DbSet.Remove(entity);

}

}

public virtual void Delete(int id)

{

var entity = GetById(id);

if (entity == null)

{

return;

}

Delete(entity);

}

public virtual void Delete(Guid id)

{

var entity = GetById(id);

if (entity == null)

{

return;

}

Delete(entity);

}

public virtual void Delete(string id)

{

var entity = GetById(id);

if (entity == null)

{

return;

}

Delete(entity);

}

public void Dispose()

{

DbContext.Dispose();

}

}

1. Add new interface IApplicationUserRepository to **Komsky.Data.DataAccess.Repositories**
   1. This interface should inherit from IRepository<ApplicationUser>
   2. For now we will add only one method to find a user – by email. Add GetByEmail method to our interface as below:

namespace Komsky.Data.DataAccess.Repositories

{

public interface IApplicationUserRepository : IRepository<ApplicationUser>

{

ApplicationUser GetByEmail(string email);

}

}

1. Now we need to implement this interface in class called ApplicationUserRepository.
   1. Add new class ApplicationUserRepository to **Repositories** folder
   2. Inherit from GenericRepository<ApplicationUser>
      1. Implement missing constructor (use Implement option from Visual Studio or ReSharper Implement missing members)
   3. Implement IApplicationUserRepository GetByEmail method. See the implementation below.

public ApplicationUser GetByEmail(string email)

{

return GetAll().SingleOrDefault(x => x.Email.ToLower() == email.ToLower());

}

1. Build your solution and debug any errors. You don’t have to run your solution yet – we haven’t done any changes to Web project so far.

At this point you should have simple repository implemented with only one method – finding application user by email. We could use it directly in our Web project, but best practise is further wrapping it by unit of work piece of code. We will use *DataFacade* pattern to achieve it.

## Task 4 – ADD DATA FAÇADE PATTERN TO DATA ACCESS LAYER

1. Add **UnitOfWork** folder to **Komsky.Data.DataAccess** project
2. Add new interface IDataFacade, implementing IDisposable interface. The code is provided below.

public interface IDataFacade : IDisposable

{

void Commit();

Task CommitAsync();

ApplicationUserRepository ApplicationUsers { get; }

}

1. Implement this interface in DataFacade class. The code is provided below.

public class DataFacade : IDataFacade

{

#region Fields

private ApplicationDbContext \_dbContext;

private ApplicationUserRepository \_applicationUsers;

#endregion

#region Constructors

public DataFacade()

{

CreateDbContext(null);

}

public DataFacade(ApplicationDbContext dbContext)

{

CreateDbContext(dbContext);

}

protected void CreateDbContext(ApplicationDbContext dbContext)

{

\_dbContext = dbContext ?? new ApplicationDbContext();

\_dbContext.Configuration.ProxyCreationEnabled = false;

\_dbContext.Configuration.LazyLoadingEnabled = false;

\_dbContext.Configuration.ValidateOnSaveEnabled = false;

}

#endregion

public ApplicationUserRepository ApplicationUsers

{

get { return \_applicationUsers ?? (\_applicationUsers = new ApplicationUserRepository(\_dbContext)); }

}

public void Commit()

{

\_dbContext.SaveChanges();

}

public Task CommitAsync()

{

return \_dbContext.SaveChangesAsync();

}

#region Dispose pattern

private bool \_disposed;

protected virtual void Dispose(bool disposing)

{

if (!\_disposed)

{

if (disposing)

{

\_dbContext.Dispose();

}

}

\_disposed = true;

}

public void Dispose()

{

Dispose(true);

GC.SuppressFinalize(this);

}

#endregion

}

1. Build your solution and debug any errors. You don’t have to run your solution yet – we haven’t done any changes to Web project so far.

At this point you should have implemented *DataFacade* pattern. Next, we are going to use *DataFacade* in our Web project instead of directly manipulating DbContext.

## Task 5 – PREPARE WEB PROJECT CODE TO REMOVE DEPENDENCIES AND USE DATA FAÇADE

At this point we still have dependency on *EntityFramework* in *AccountController* and *ManageController* classes. Unfortunately, to remove them, we would have to implement from scratch *UserStore* from ASP.NET Identity on domain level, and we will take care of that in module 9. In the mean time, we will add ability to use *DataFacade* in our home controller just to display user name (the only method available now in repository is GetByEmail).

1. Move **IdentityConfig.cs** from **Komsky.Web/App\_Start** to **Komsky.Services** project
   1. Copy file from **Komsky.Web/App\_Start** to **Komsky.Services**
   2. Delete file from **Komsky.Web/App\_Start**
2. Add missing **Komsky.Data.DataAccess** reference to **Komsky.Services** project
3. Install missing nugget packages to **Komsky.Services** project

PM> install-package Microsoft.AspNet.Identity.EntityFramework

PM> install-package Microsoft.AspNet.Identity.Owin

1. Add **Komsky.Service** reference to **Komsky.Web** project
2. Build, run and test your solution, debugging any errors
3. Go to home controller, and add private readonly field of IDataFacade type
4. Create constructor for home controller with injecting IDataFacade object
   1. Do not create default constructor, to make sure that our injection constructor is used by application

private readonly IDataFacade \_dataFacade;

public HomeController(IDataFacade dataFacade)

{

\_dataFacade = dataFacade;

}

1. Modify Index Action Method code, to get user details, if he or she is logged in.

public ActionResult Index()

{

if (User.Identity != null && !String.IsNullOrEmpty(User.Identity.Name))

{

ViewBag.UserDetails =  
 \_dataFacade.ApplicationUsers.GetByEmail(User.Identity.Name).Email;

}

return View();

}

1. Modify Index view, to display user email, if it’s provided by Action Method

@if (ViewBag.UserDetails == null)

{

<h1>ASP.NET</h1>

}

else

{

<h1>@ViewBag.UserDetails</h1>

}

1. Run your application. You should end up with exception:

## ***No parameterless constructor defined for this object.***

1. Install Ninject from this package:

PM> install-package Ninject.MVC5

1. Notice new file in **App\_Start** folder: **NinjectWebCommon.cs**
   1. The last thing we need to do is bind IDataFacade interface provided in HomeController constructor to actual DataFacade class. Add following code to RegisterServices method:

kernel.Bind<IDataFacade>().To<DataFacade>();

1. Run and test your app. Home page should now display ASP.NET when you’re not logged in, and your email, when you are.
   1. If you have problems with building solution, just run the project instead of building it. This is due to compile error in UnitTest project.
   2. If you have finished before the rest of the group – please, fix the test project, to successfully run and pass the tests. Can you see some dependency we haven’t removed?

At this point we have ended with independent HomeController, what is very important for maintaining, testing and further development of new features.

Account controller as well as Manage controller still have dependencies, but we will solve that in module 9.

## LAB 01 SUMMARY

In this lab we have heavily refactored default MVC template to remove dependencies and use two standard development patterns – Repository and Unit of Work. We have ended with nice generic template that will work for many smaller and bigger projects. We will use it as a scaffold for out ticketing system. The reason we did this on the beginning of the project is that later on we will keep our controllers independent.

Lab 02

# Content:

Out target for this lab is to create ability to manage customers of Komsky Inc, that is Create Read Update and Delete (CRUD) operations. For that purpose we will create business objects with correct validation rules. We will also add and use new layer in this lab – a business layer called **Services,** as well as *Factory pattern*, to create new objects.

# General steps:

1. Create Customer data layer
2. Create Customer business layer with factory development pattern
3. Create Customer view model and manage objects on the Web application layer.
4. <TODO>

## Task 1 – CREATE CUSTOMER DATA LAYER

1. Add **Models** folder to **Komsky.Domain** project
2. Add new Customer class to Models folder
   1. Class name: CustomerDomain
   2. Field type: Int32, name: Id
   3. Field type: String, name: Name
3. Add **Handlers** folder to **Komsky.Services** project
4. Add new IBaseHandler interface to Handlers folder. The code for the interface is provided below:

public interface IBaseHandler<T> : IDisposable

{

IEnumerable<T> GetAll();

T GetById(Int32 id);

void Add(T domainObject);

void Update(T domainObject);

void Delete(T domainObject);

void Delete(int domainObjectId);

}

1. Implement this interface in a generic class called BaseHandler<T>. Can you try implementing this by yourself? Below are some hints, and full code on the next page.
   1. Use IDataFacade as your data source
      1. Use protected accessibility modifier
      2. Name it DataFacade
   2. Initialize this filed in constructors, and remember about dependency rules   
      – use two constructors, one default, and one for DI.
   3. Implement only Dispose() method, and dispose DataFacade object in it.
   4. Keep rest of the methods as abstract
2. Add new class called CustomerHandler to **Komsky.Services/Handlers** project.
   1. Implement BaseHandler<CustomerDomain> within that class.  
      (I know, it’s impossible at this point – DataFacade doesn’t provide Customers object yet, but let’s change this quicky in next step).
3. Go to IDataFacade interface and expose CustomerRepository named Customers
4. Go to **Komsky.Data.DataAccess/Repositories** and add CustomerRepository
   1. Implement GenericRepository and ICustomersRepository interface
      1. Implement IRepository<Customer>.
         1. Which in order require Customer object from Entities project.  
            OK, we need to slow down, here. Stop coding, and go to next point.
5. Add class to **Entites** project your Customer POCO class with the same properties as **CustomerDomain** object, that is Id and Name.
   1. Mark Id property as a Key property using [Key] attribute from System.ComponentModel.DataAnnotations namespace.
   2. Add additional property, as at this level we know, that a Customer contains a collection of Users.

[Key]

public Int32 Id { get; set; }

public String Name { get; set; }

public virtual ICollection<ApplicationUser> Users { get; set; }

1. Navigate to ApplicationUser definition class and add a Customer property, as follows:

public Customer Customer { get; set; }

1. Add a DbSet collection property to your ApplicationDbContext in **DataAccess** project, as follows:

public virtual DbSet<Customer> Customers { get; set; }

1. Now, our data access is ready, and we can proceed to finish implementing customer repository and data façade. Go to **Repositories** folder in **DataAccess** project and implement CustomerRepository, as follows:
   1. Inherit from GenericRepository<Customer> class
   2. Implement ICustomerRepository interface.
   3. Your class should look like this:

public class CustomerRepository : GenericRepository<Customer>, ICustomerRepository

{

public CustomerRepository(DbContext dbContext) : base(dbContext)

{

}

}

1. Add CustomerRepository to IDataFacade

CustomerRepository Customers { get; }

1. Implement missing members from IDataFacade in DataFacade class, as following:

private CustomerRepository \_customers;

public CustomerRepository Customers

{

get { return \_customers ?? (\_customers = new CustomerRepository(\_dbContext)); }

}

1. Build your project and debug any errors.

At this point we have finished working with data layer. Before we can use Customer object on the Web layer, instead of using it directly, we will create a service handler, to manage our business layer. This approach won’t introduce additional Data Layer dependency, so in future we will be ready to remove it completely.

## Task 2 – CREATE CUSTOMER BUSINESS LAYER

1. Create new class called CustomerHandler in **Komsky.Services/Handlers** project
   1. Inherit from BaseHandler<Customer> class and implement required methods
2. On this level we’ll use DataFacade object, but DataFacade operates on Customer entity and we need CustomerDomain object, that will be used. We can create this on the fly, or we can introduce new development pattern, that we can reuse. So, let’s create that factory class!
3. Go to **Komsky.Domain** project and create folder **Factories**
4. Add new public static class called CustomerDomainFactory to this location
5. Add Create static method with return type CustomerDomain
   1. Add input parameter Customer from **Entities** project
   2. Convert Customer object into CustomerDomain as follows

public static class CustomerDomainFactory

{

public static CustomerDomain Create(Customer customer)

{

return new CustomerDomain

{

Id = customer.Id,

Name = customer.Name

};

}

}

1. You can also introduce this additionally in form of extension method (this might be useful later on) as follows:

public static CustomerDomain CreateCustomerDomain(this Customer customer)

{

return Create(customer);

}

1. Let’s get back to our CustomerHandler class. Now, having factory ready, we can easily create CustomerDomain object from Customer. Implement all reading methods using this factory, as follows:

public override IEnumerable<CustomerDomain> GetAll()

{

return DataFacade.Customers.GetAll().Select(CustomerDomainFactory.Create);

}

public override CustomerDomain GetById(int id)

{

return DataFacade.Customers.GetById(id).CreateCustomerDomain();

}

1. But we still need CustomerDomain to Customer factory object! Head back to Factories folder and implement new factory class called CustomerFactory. Implement this the same way you’ve implemented CustomerDomain factory, that is:

public static class CustomerFactory

{

public static Customer Create(CustomerDomain customerDomain)

{

return new Customer{Id = customerDomain.Id, Name = customerDomain.Name };

}

public static Customer CreateCustomer(this CustomerDomain customerDomain)

{

return Create(customerDomain);

}

}

1. For now we will just ignore Users property of Customer object, but after refactoring Identity, we will get back to it.
2. Head back to CustomerHandler, and finish implementing add, update and delete methods. The code might look like this:

public override void Add(CustomerDomain domainObject)

{

DataFacade.Customers.Add(domainObject.CreateCustomer());

}

public override void Update(CustomerDomain domainObject)

{

DataFacade.Customers.Update(domainObject.CreateCustomer());

}

public override void Delete(CustomerDomain domainObject)

{

Delete(domainObject.Id);

}

public override void Delete(int domainObjectId)

{

DataFacade.Customers.Delete(domainObjectId);

}

1. Build your solution and debug any errors.

At this point, we have business layer ready to use. Let’s jump to Web layer, to finally have some new features available.

## Task 3 – CREATE WEB LAYER

1. We need to start with new view object model, to scaffold customer controller object with views. Go to **Models** folder in **Komsky.Web** project and add CustomerViewModel class with Id and Name properties.
2. Build the project before proceeding.
3. Scaffold new controller called Customer by right clicking **Controllers** folder in **Komsky.Web** project and choose Add -> Controller…
   1. Choose *MVC 5 Controller with views using Entity Framework*
   2. Model class: *Customer (Komsky.Data.Entities)*
   3. Data context class: *ApplicationDbContext (Komsky.Data.DataAccess)*
   4. Controller name: CustomerController

***Warning****! This approach lets us easily scaffold controller and introduce views to our project, but it’s possible only because we still have data access layer dependency in Web project. We will change Customer entity into CustomerViewModel and DbContext into CustomerHandler in next steps. Later on, we will have to create actions and views manually, because EF will not be available on this layer.*

1. Add **Komsky.Domain** reference do **Komsky.Web** project
2. Go to newly create CustomerController class and make following changes:
   1. Remove ApplicationDbContext field from class definition
   2. Introduce IBaseHandler<Customer> field called \_customerHandler and initialize this using DI approach. The code might looks like this:

private readonly IBaseHandler<CustomerDomain> \_customerHandler;

public CustomerController(IBaseHandler<CustomerDomain> customerHandler)

{

\_customerHandler = customerHandler;

}

1. Notice, that we don’t use CustomerHandler type here, but rather generic interface. This will have much impact later on in testing.
2. Go to **App\_Start/NinjectWebCommon.cs** and bind this interface with real CustomerHandler class in RegisterServices() method.

kernel.Bind<IBaseHandler<CustomerDomain>>().To<CustomerHandler>();

1. Go back to CustomerController and comment out body of every scaffolded action method. We will use that content as a hint, what kind of model is expected in view.
2. Change Dispose method body, so instead of db, our \_customerHandler field is disposed here.
3. Go to **Views/Customer** folder and change model for every view found there from Customer into CustomerViewModel.

@model Komsky.Web.Models.CustomerViewModel

1. Before using CustomerViewModel, we need to convert this object from CustomerDomain.
   1. Go to **Models** folder and create **Factories** folder
   2. Create CustomerViewModelFactory. The class might looks like this:

public static CustomerViewModel Create(CustomerDomain customerDomain)

{

return new CustomerViewModel{Id = customerDomain.Id, Name = customerDomain.Name};

}

public static CustomerViewModel CreateCustomerViewModel(this CustomerDomain customerDomain)

{

return Create(customerDomain);

}

* 1. Create CustomerDomainFactory class. The class might looks like this:

public static CustomerDomain Create(CustomerViewModel customerViewModel)

{

return new CustomerDomain { Id = customerViewModel.Id, Name = customerViewModel.Name };

}

public static CustomerDomain CreateCustomerDomain(this CustomerViewModel customerViewModel)

{

return Create(customerViewModel);

}

1. Go to **Views/Shared** folder and edit **\_Layout.cshtml** file
   1. Add additional link to main menu, pointing to out new Customer controller.

<li>@Html.ActionLink("Home", "Index", "Home")</li>

**<li>@Html.ActionLink("Customers", "Index", "Customer")</li>**

<li>@Html.ActionLink("About", "About", "Home")</li>

1. Build your solution and debug any errors.
2. Run your solution and navigate to *Customers* from top menu.
   1. Yes, that’s true. Finally, you can see effects of your work!
   2. But I’m sorry that you see this ugly Entity Framework exception. What happened? We have modified our code-first file, but forgot to actually apply this to database.

At this point we have beautiful, nicely layered app, but we still need to update database file by adding new *Customers* table. We tackle that problem in last task of this lab by introducing Entity Framework code migrations.

## Task 4 – ADD ENTITY FRAMEWORK CODE MIGRATIONS

1. Gone