Trainer and student course materials workbook

Copyright 2016 © Adrian Ilewicz

ASP.NET MVC 5 with Development Patterns

Printed slides and hands-on-labs inside

# Workbook information

Title: ASP.NET MVC 5 with Development Patterns

Revision: A

First release: January 2016

Adrian Ilewicz - All rights reserved

[Adrian.ilewicz@gmail.com](mailto:Adrian.ilewicz@gmail.com)

<https://uk.linkedin.com/in/ailewicz>

# Course overview

This course focuses on developing ASP.NET MVC 5 application by implementing well-known development patterns and best practises.

This training has been designed to be a 5-days trainer-led classroom course.

## Course outline

1. Designing MVC 5 applications using Development patterns
   1. Methodology and project phases
   2. Designing models
   3. Application layers – example designs
      1. Repository pattern
      2. Unit of work / data façade pattern
   4. Dependency Injection
      1. Entourage anti-pattern
      2. Stairway
2. Developing MVC models
   1. POCO and EF classes for schema design
   2. Entity Framework Code First
      1. Domain application layer – domain objects
      2. Client application layer - ViewModel objects
      3. Factory pattern
   3. Entity Framework code migrations
   4. Validation via attributes
   5. Validation with Fluent Validation
3. Developing MVC controllers
   1. Routing and controllers
   2. Model binders
   3. Action Filters
4. Views
   1. HTML Helpers
   2. Partial Views
   3. Layout and strongly typed views
5. Testing and debugging
   1. MVC unit tests
   2. Test driven development
   3. Mocking with Moq
      1. Mocking dependencies
      2. Mocking HttpContext
   4. Global error handling
      1. Saving error data into database
   5. MVC health monitoring
6. Application structure
   1. Information architecture
   2. Routing
7. Designing and styling interface
   1. Views layout
   2. Responsive layout using CSS3 and Bootstrap
8. Application performance and responsiveness
   1. jQuery AJAX calls to MVC
   2. Dynamic partial modals
   3. Output Cache and Object cache
   4. Application State
      1. Session object
      2. Application object
9. ASP.NET MVC Identity 2.1 framework
   1. Implementing Identity in layered application
   2. MVC security layers
   3. Controlling access to resources
   4. OAuth – integrating your application with Facebook
   5. Designing secure applications
10. Web Services
    1. Idea behind SOA
       1. Scalable applications
       2. Resilient applications
    2. OData using WCF Data Services
11. WEB API
    1. Implementing RESTfull applications
12. Globalisation and localisation of MVC applications
13. Deploying MVC
    1. Deploy to local
    2. Deploy to cloud
14. Optional topics
    1. jQuery DataTables with server side processing
    2. Task automation with Node.js/npm using bower and grunt
    3. Other useful frameworks

## Machine specification

For the purpose of this training a following software is required on student’s machine.

* Visual Studio 2013 Professional or better
  + Visual Studio Community Edition will do
  + https://www.visualstudio.com/en-us/downloads/download-visual-studio-vs.aspx
* SQL Server 2012 Express with Management Studio
  + <https://www.microsoft.com/en-gb/download/details.aspx?id=42299>
  + Choose Express and Tools
* Internet Information Services (IIS) server
  + Use any version available in Windows
* Any GIT client for downloading source code from Github
  + <https://www.sourcetreeapp.com/>
* Google Chrome browser

Trainer machine will additionally require:

* Microsoft PowerPoint 2010 or better
* ZoomIT
  + <https://technet.microsoft.com/en-us/sysinternals/zoomit.aspx>

Student requirement

Trainer requirement

At least 5 years of hands-on experience with C#.NET Framework

# Lab Overview

## Introduction

KOMSKY Ltd is a technology company producing electronic 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 tickets to Komsky to be resolved. Each ticket should have Ticket 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 ticket
* Customer users shall be able to view their own company's tickets
* Agents shall be able reply to customer tickets
* Agents shall be able to pick ticket from ticket pool and assign them to themselves.
* Agents can reply to any ticket 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
* Tickets shall have discussion board where Customer users and agents discuss the tickets
* Only authorized users shall be able to login to portal
* Every unhandled exception must be logged into database
* A modern template must be used and page must be able to adapt to device
* Non-functional requirement – a development and architecture patterns must be used for easy application extensibility and maintenance.

As a mid-level Web Developer you want to apply well-known patterns during development phase for the software to be maintainable in the future.

## 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 Repository

I use GitHub to store project files for this training and it’s publicly available here

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

Use master branch to checkout latest LAB task versions (eg. LAB starter and LAB solution project states).

Feel free fork this project, or to contact me at [adrian.ilewicz@gmail.com](mailto:adrian.ilewicz@gmail.com) with project issues.

## Conventions

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

*Italics* – solution specific names and additional  
explanations

## Lab Workflow and Code Examples

On the beginning, every piece of code for implementing patterns and code constructs is provided just below the task description, but as the labs proceed further and further student is expected to assimilate code constructs and use them without code examples. In case of trouble, a code solution project is provided on the github source.

## Snippets

Because some of the commands and code writing labour are time-intensive tasks, a group of snippets has been prepared for some of the commands. If a lab command contains a [Snippet: CustomerDomain] text, it means a snippet is available. Snippets must be imported prior usage, and are grouped with lab number prefix, eg CustomerDomain snippet from Lab 02 will be named lab02\_CustomerDomain.

Snippets folder: \Materials\Snippets\Lab XX where xx is lab number

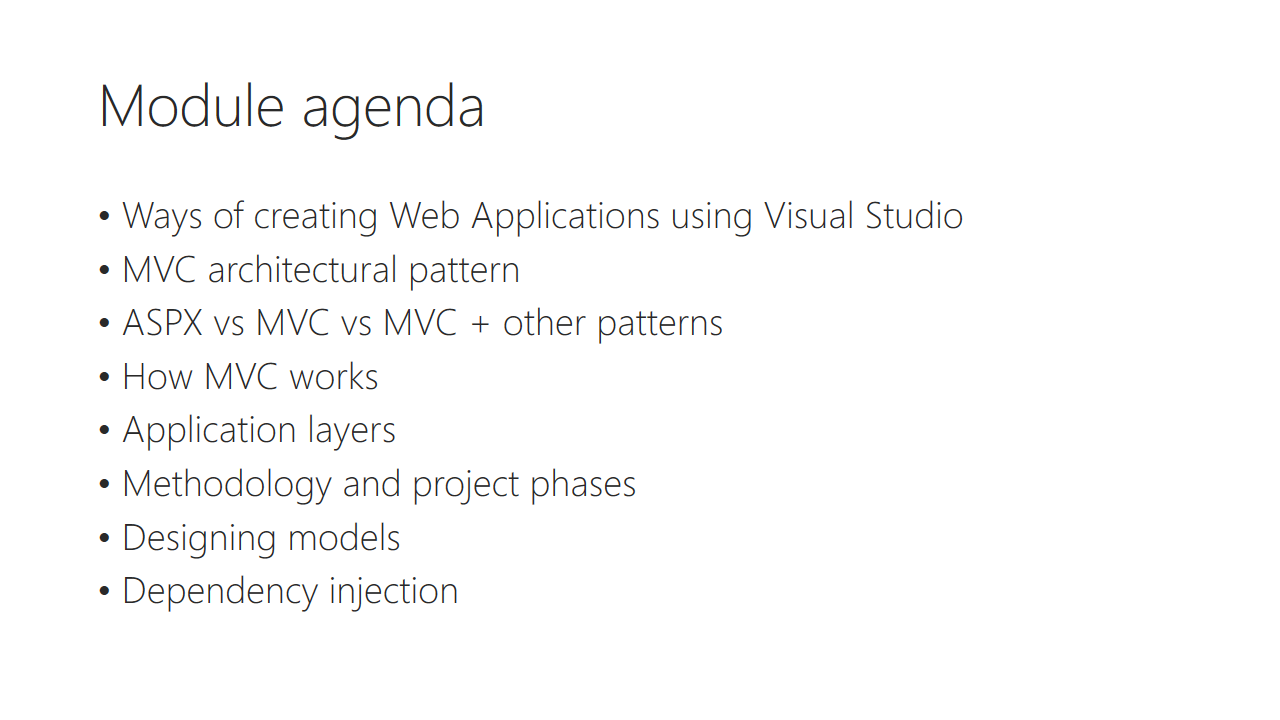
Visual Studio snippets import command: Tools => Code Snippet Manager => Import

## Demo trainer notes:

If you are a trainer, you might notice, that demonstration steps for most of the demonstration are just general or not provided at all. Feel free to run demonstrations as you wish, but please mind that some of them might require to complete previous demonstrations (eg. factories require layers and at least one entity and one domain class). The best way to prepare for and run demonstration is this:

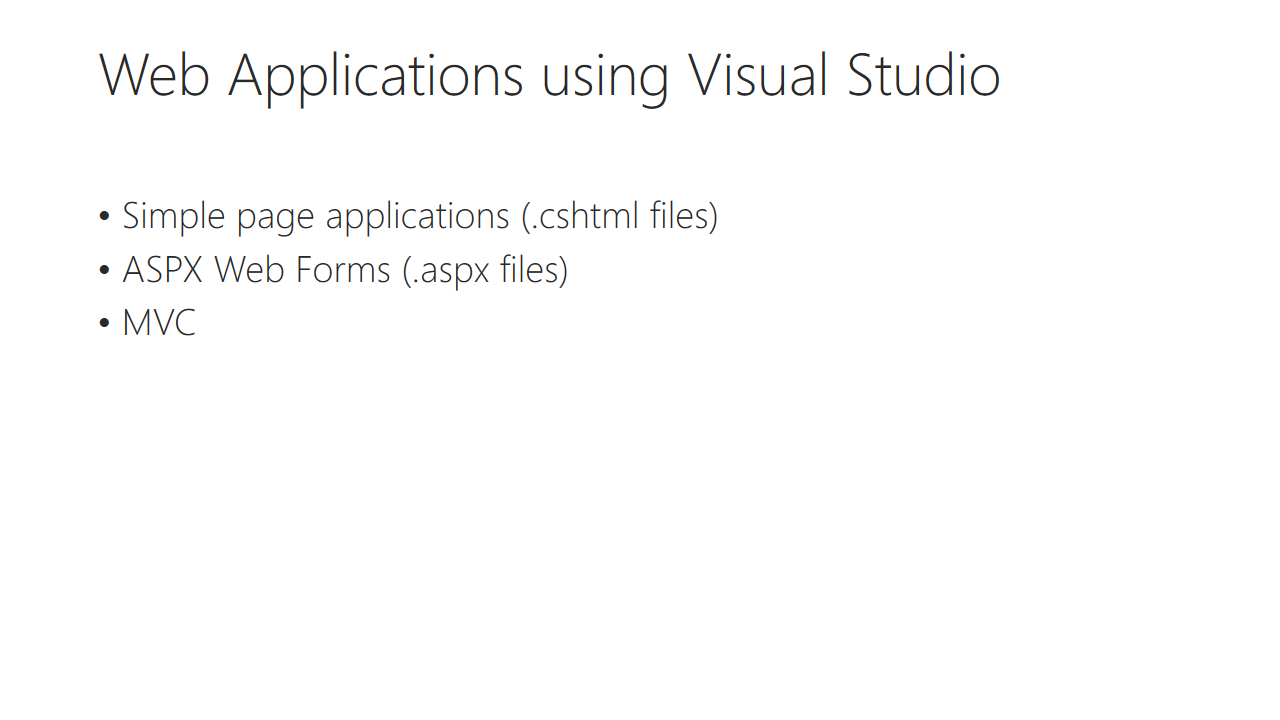
* Read through the content and complete all the labs
* Start first demonstration in module by opening previous labs solution project. Use git master branch to checkout spe
* Show students part of the labs as demonstration.
* Remember that some snippets are available.

# Module 01 Development Patterns



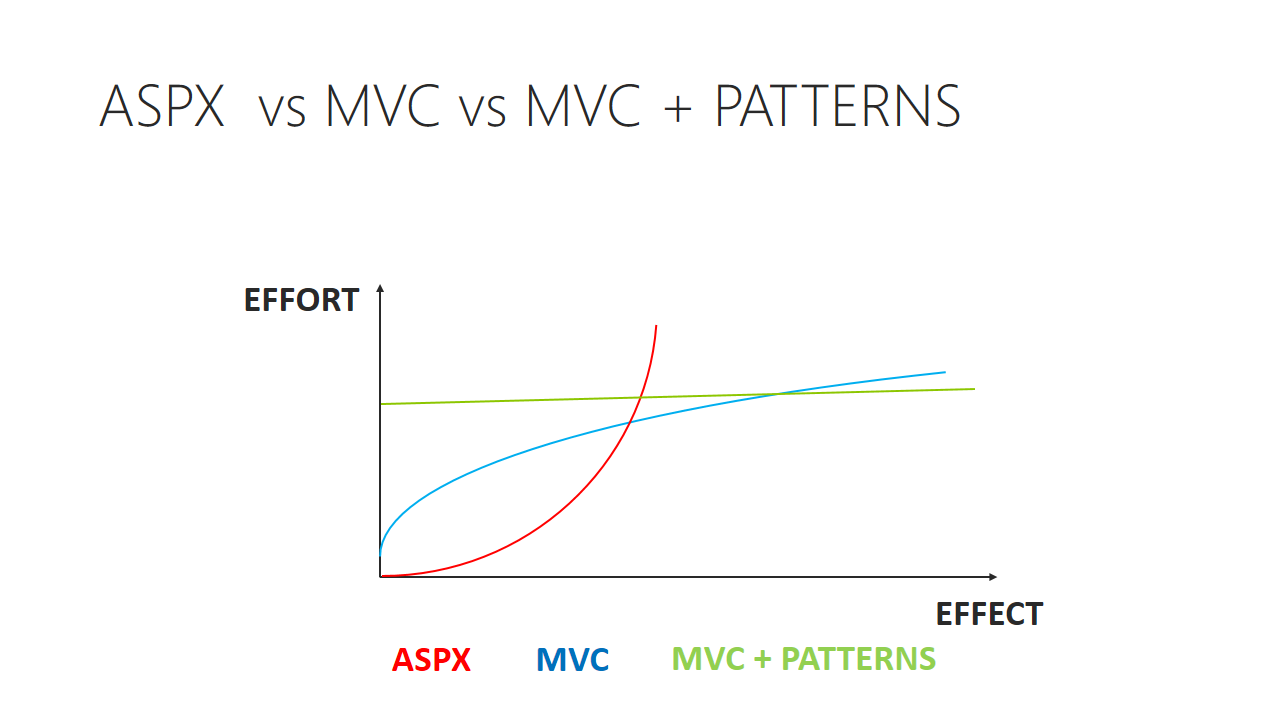
Trainer notes:

Student notes:



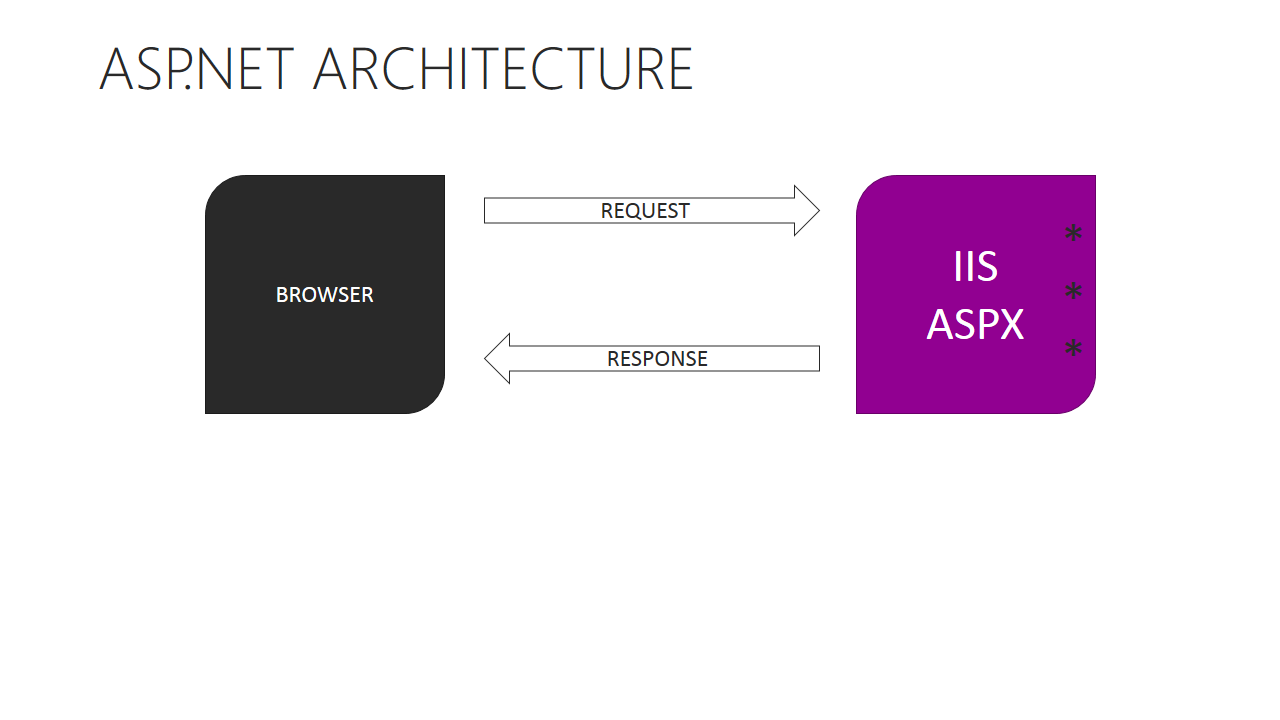
Trainer notes:

Student notes:



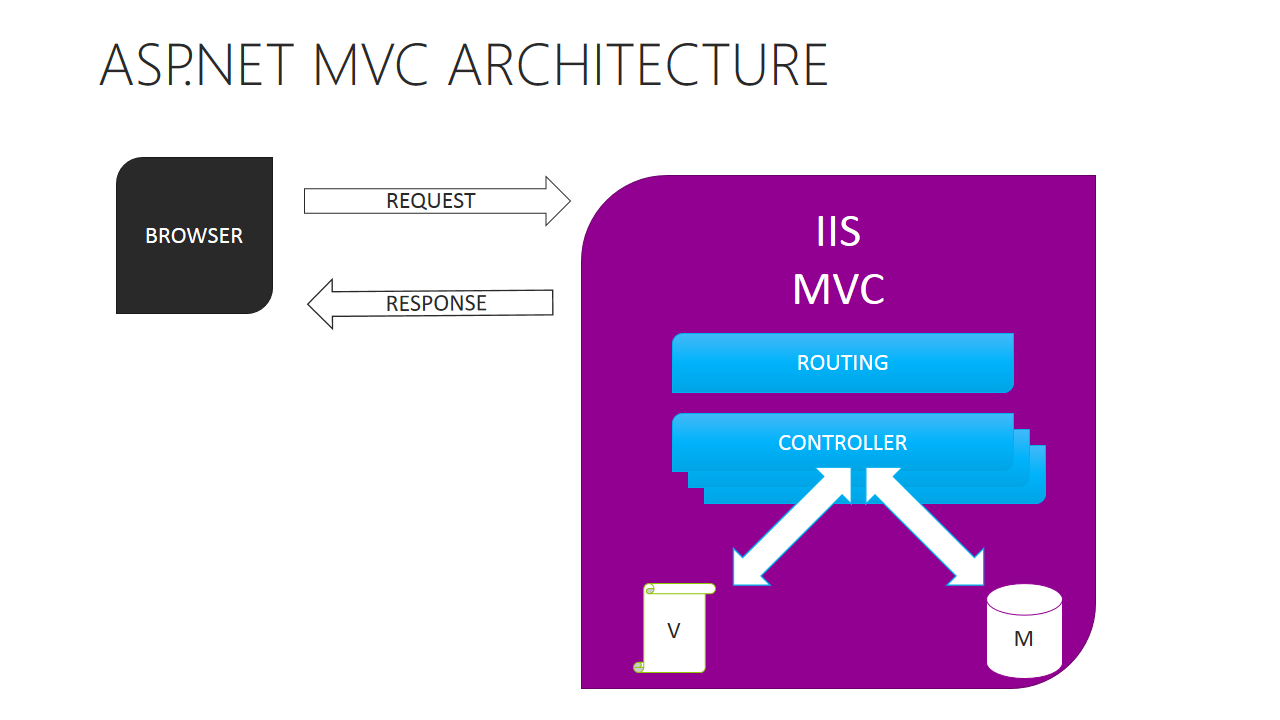
Trainer notes:

Student notes:



Trainer notes:

Student notes:



Trainer notes:

Student notes:



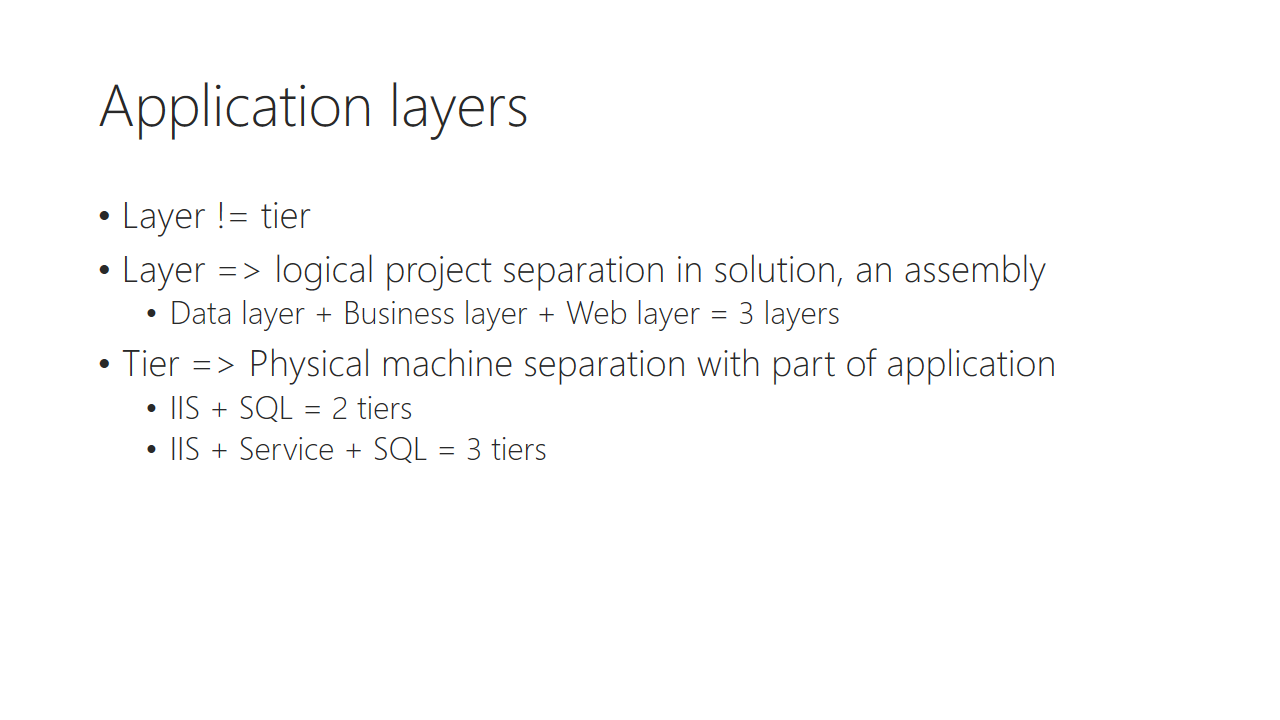
Trainer notes:

Student notes:

Demo: MVC Template components

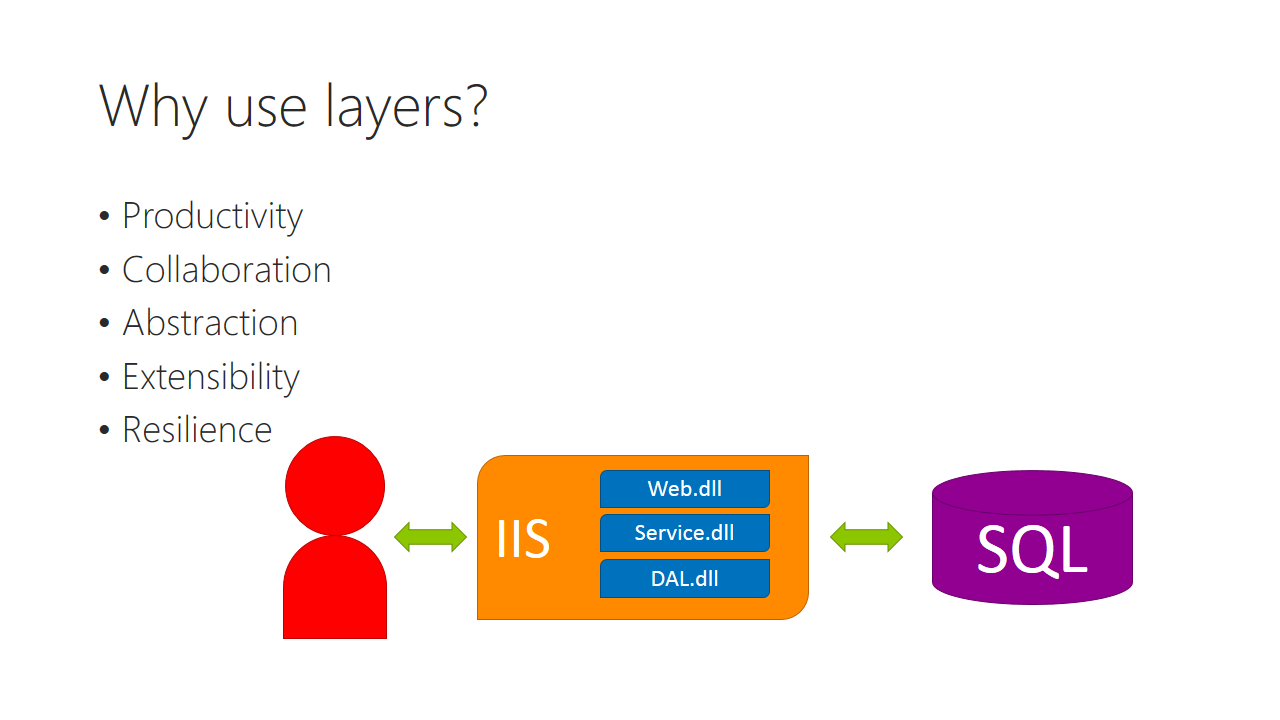
Trainer notes:

Student notes:



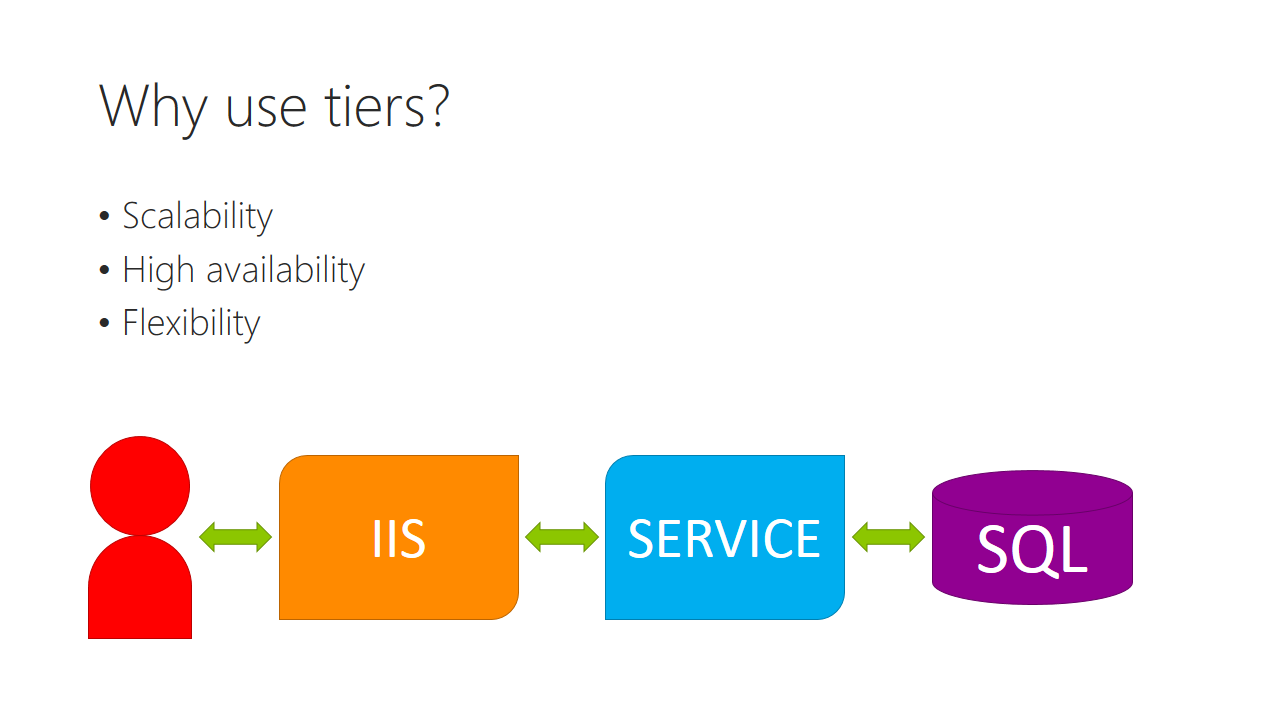
Trainer notes:

Student notes:



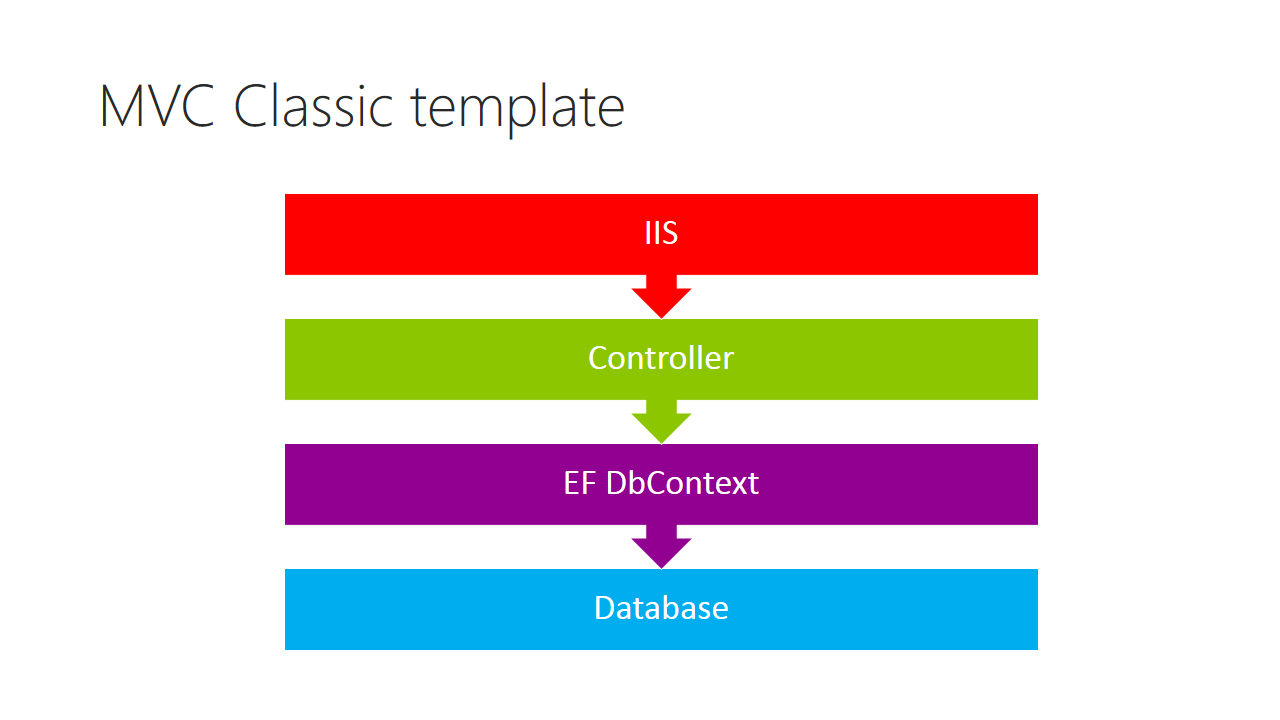
Trainer notes:

Student notes:



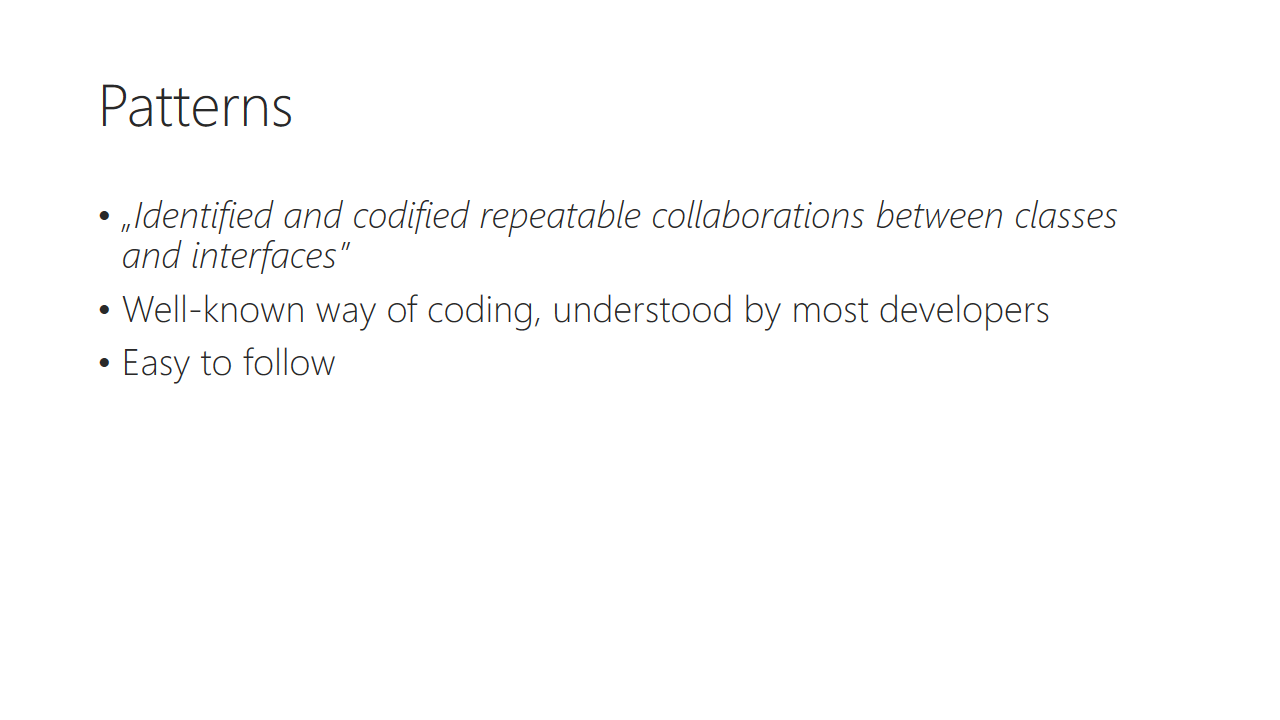
Trainer notes:

Student notes:



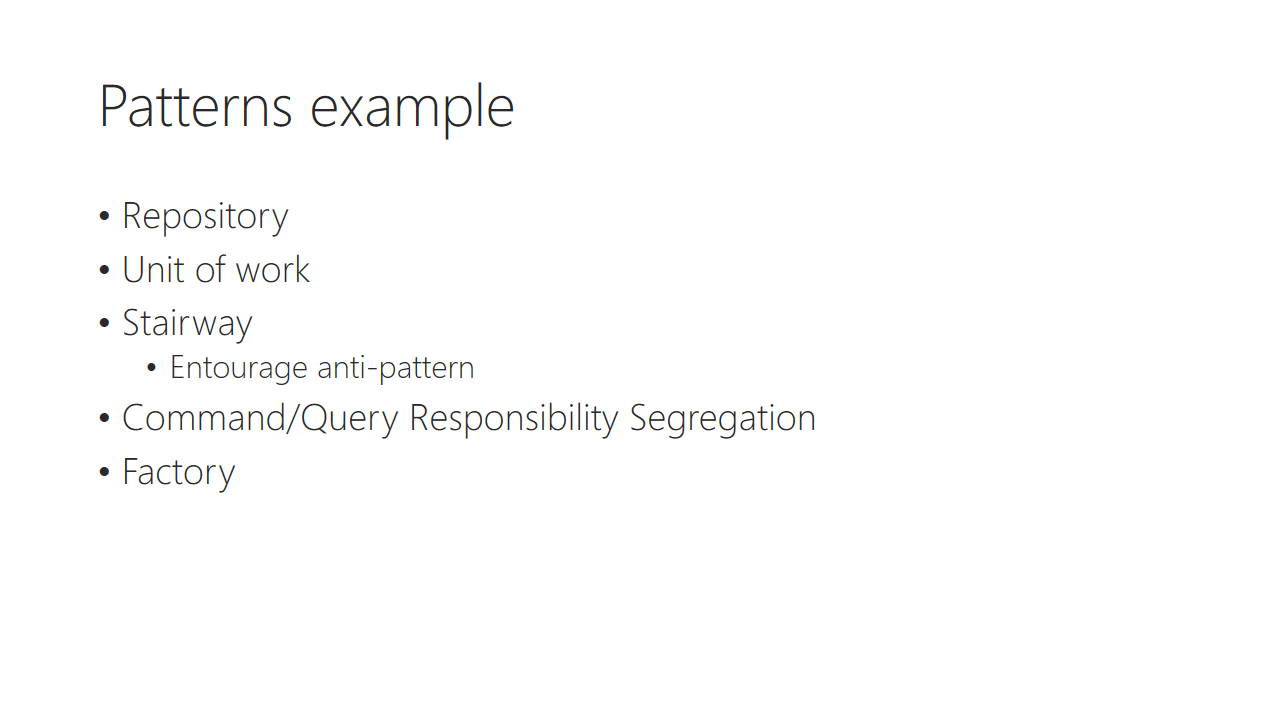
Trainer notes:

Student notes:



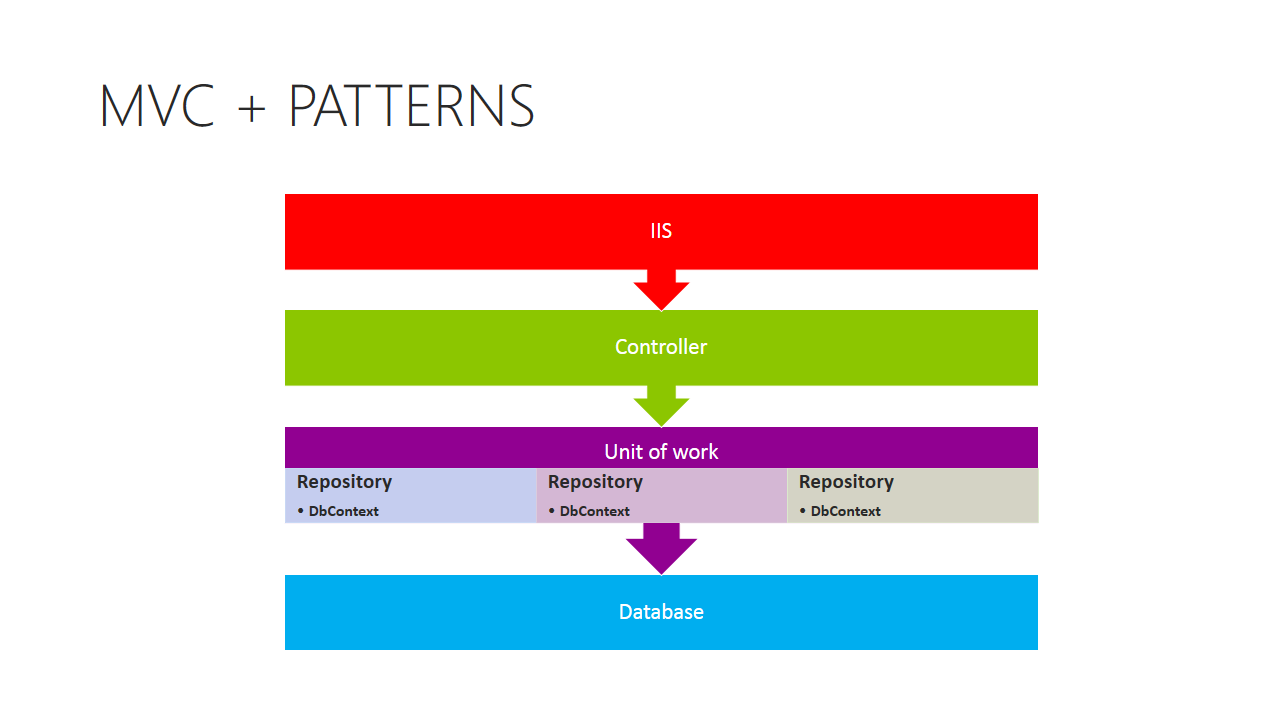
Trainer notes:

Student notes:



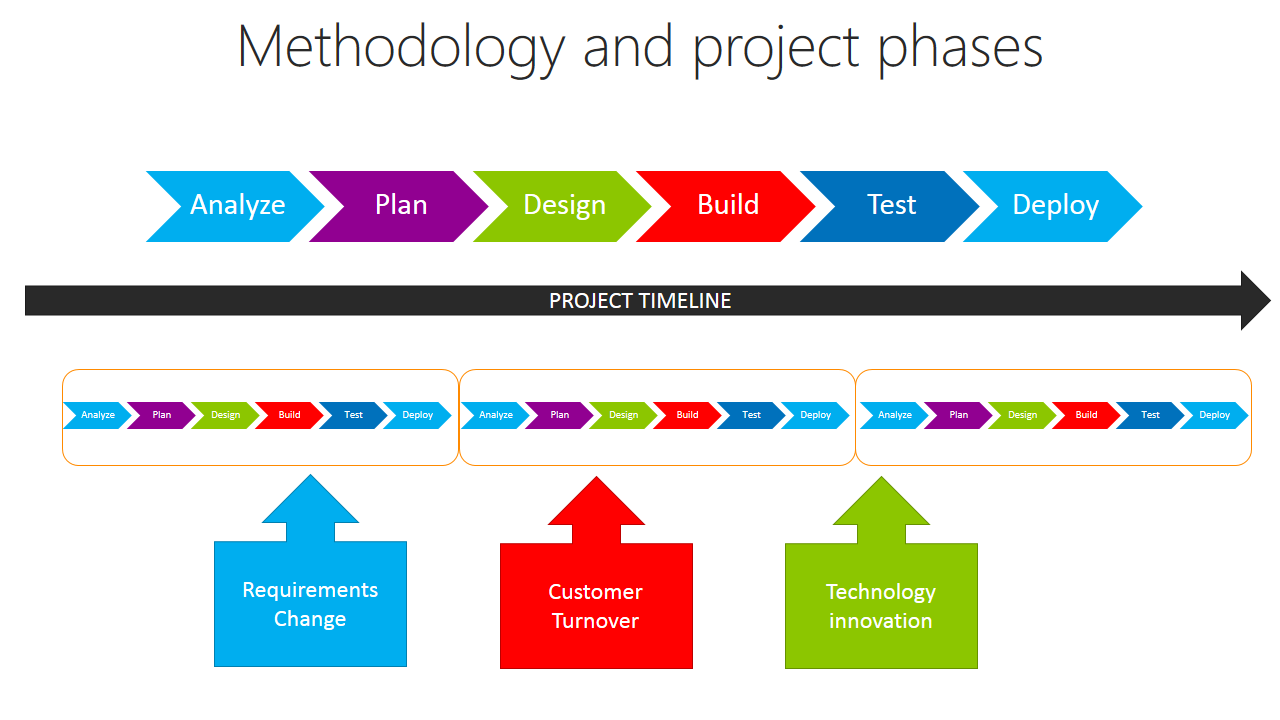
Trainer notes:

Student notes:



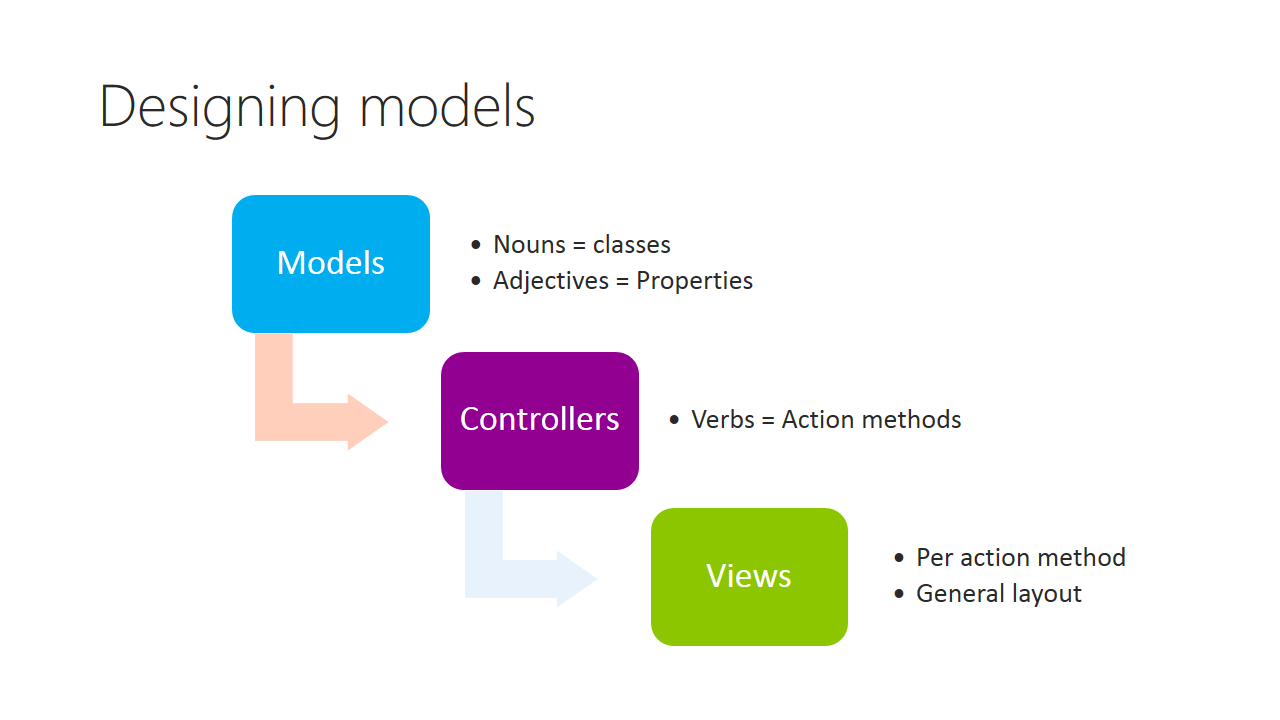
Trainer notes:

Student notes:



Trainer notes:

Student notes:



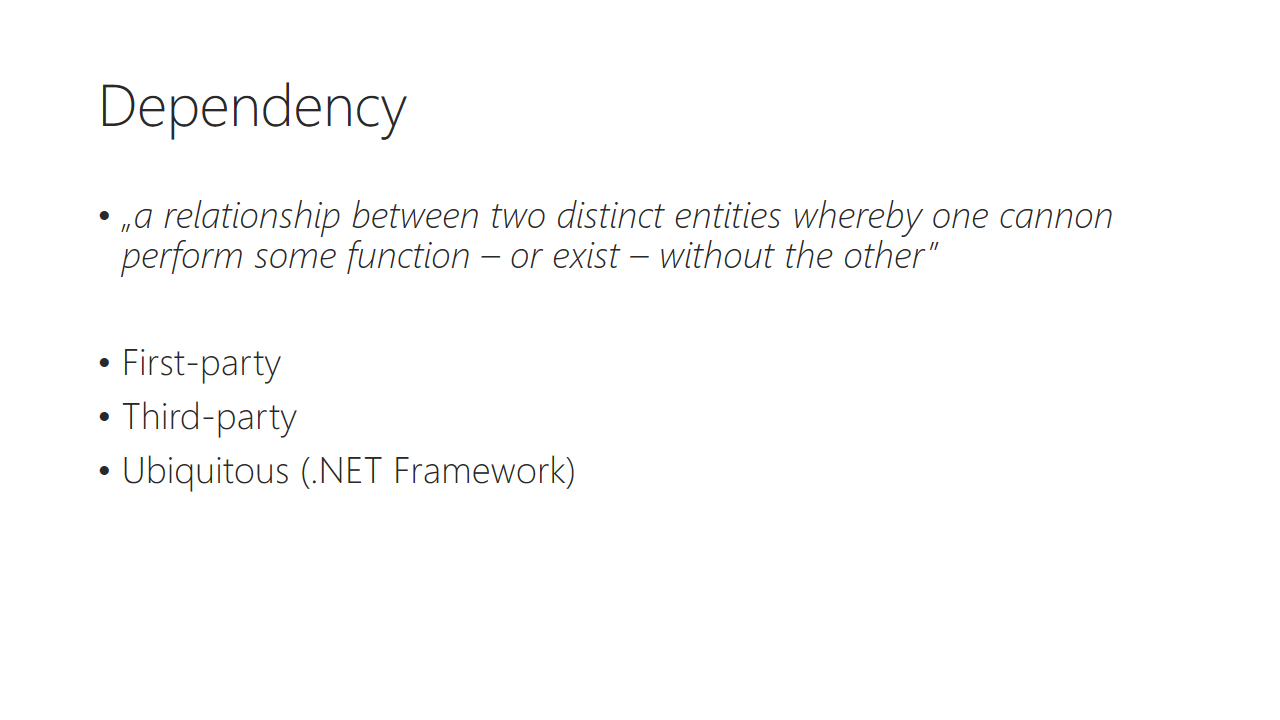
Trainer notes:

Student notes:

Demo: Scaffolding models

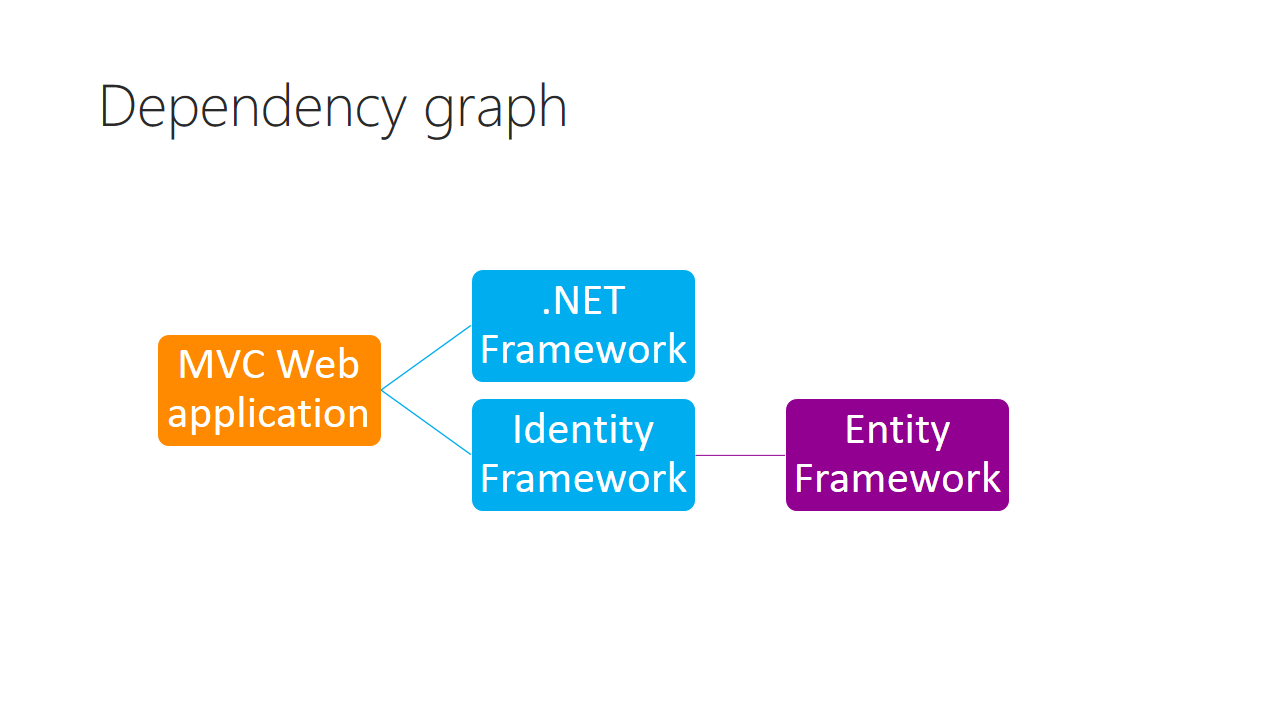
Trainer notes:

Student notes:



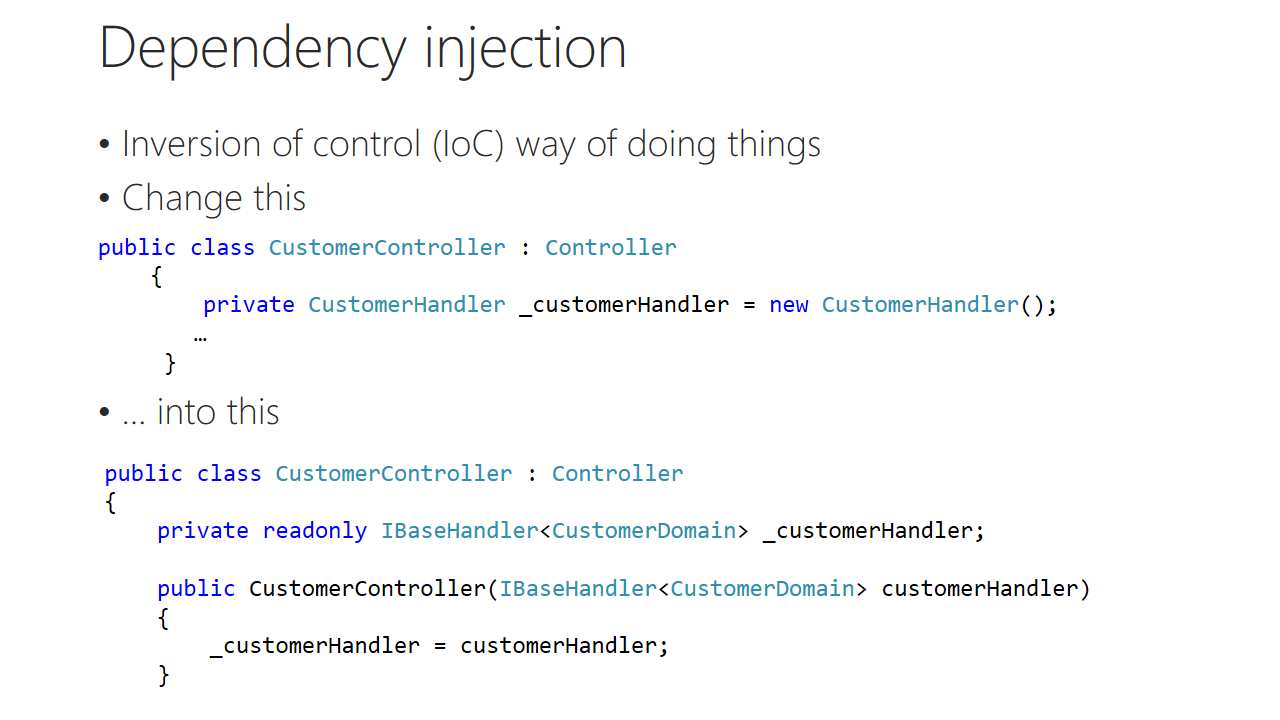
Trainer notes:

Student notes:



Trainer notes:

Student notes:



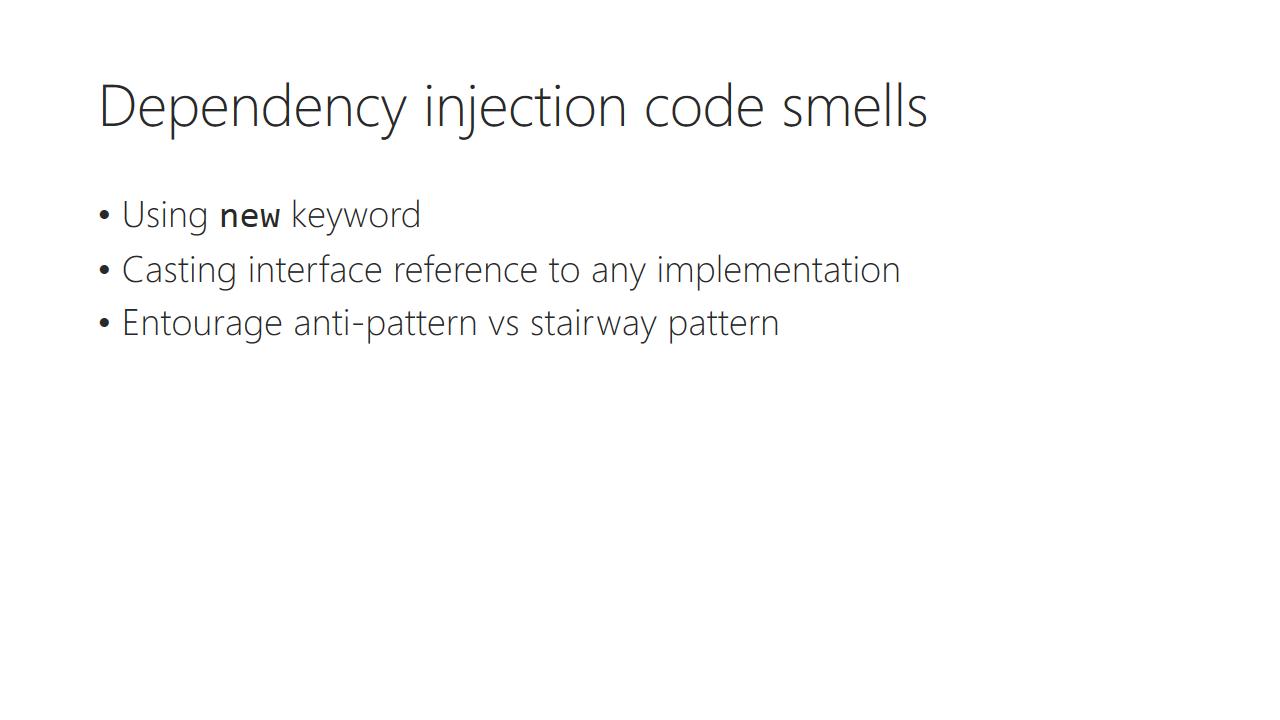
Trainer notes:

Student notes:

Demo: Refactoring MVC template to use DI

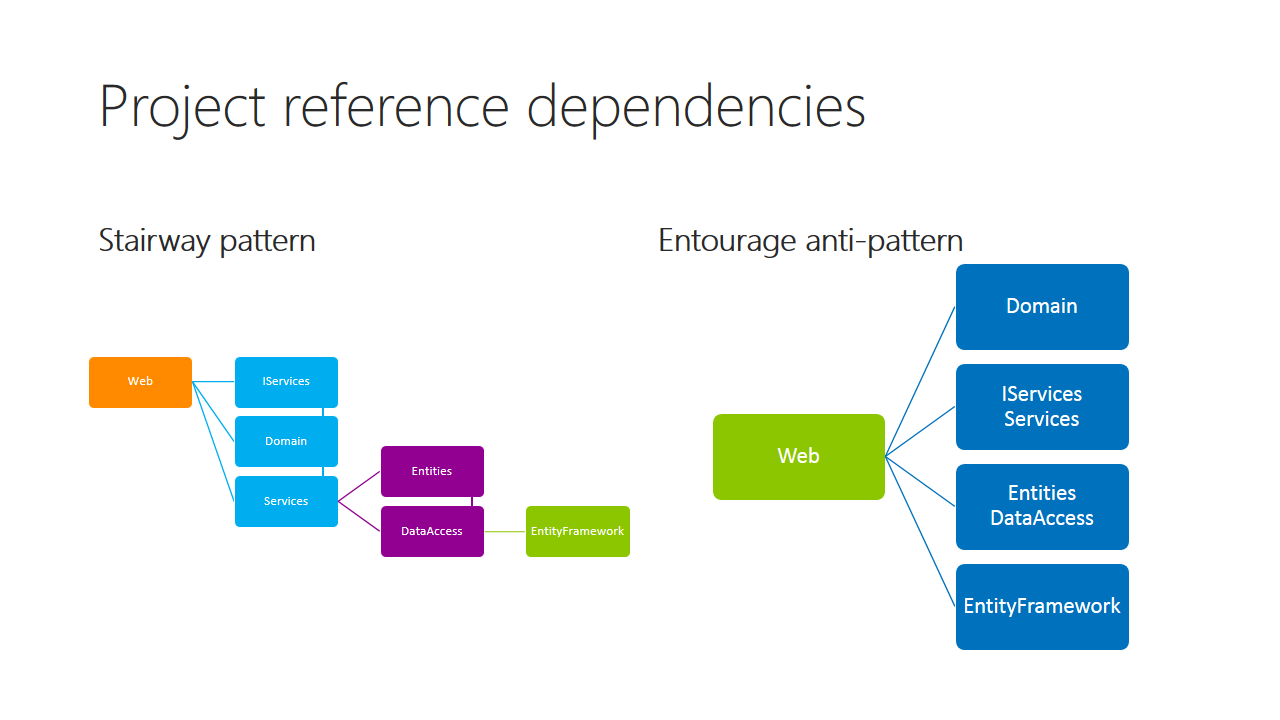
Trainer notes:

Student notes:



Trainer notes:

Student notes:



Trainer notes:

Student notes:

Demo: Project references using stairway

Trainer notes:

* Create Data folder and add Komsky.Data.DataAccess class library
* Create Services folder and add Komsky.Domain.Services class library
* In Komsky.Web add reference to Komsky.Domain.Services
* In Komsky.Domain.Services add reference to Komsky.Data.DataAcces
* If you have Code Maps Visual Studio feature, show dependencies using Code Maps:
  + Right click on Komsky.Web and Show on Code Map.
  + Right click on Komsky.Domain.Services and Add to Code Map
  + Right click on Komsky.Data.DataAccess and Add to code map

Student notes:

Demo: Installing and using Ninject

Trainer notes:

* Install Ninject to Komsky.Web using Ninject.MVC5
* Show NinjectWebCommon file.
* Don’t refactor your application to use it yet, it will be explained in the labs

Student notes:

# 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) and Data Access 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 MVC5 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 nuget 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 Class Library 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.
   3. In **Komsky.Data.DataAccess** add reference to **Komsky.Data**, so IRepository is accessible. 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. Add missing Komsky.Data.Entites reference to Komsky.Services project
4. Install missing nuget 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?

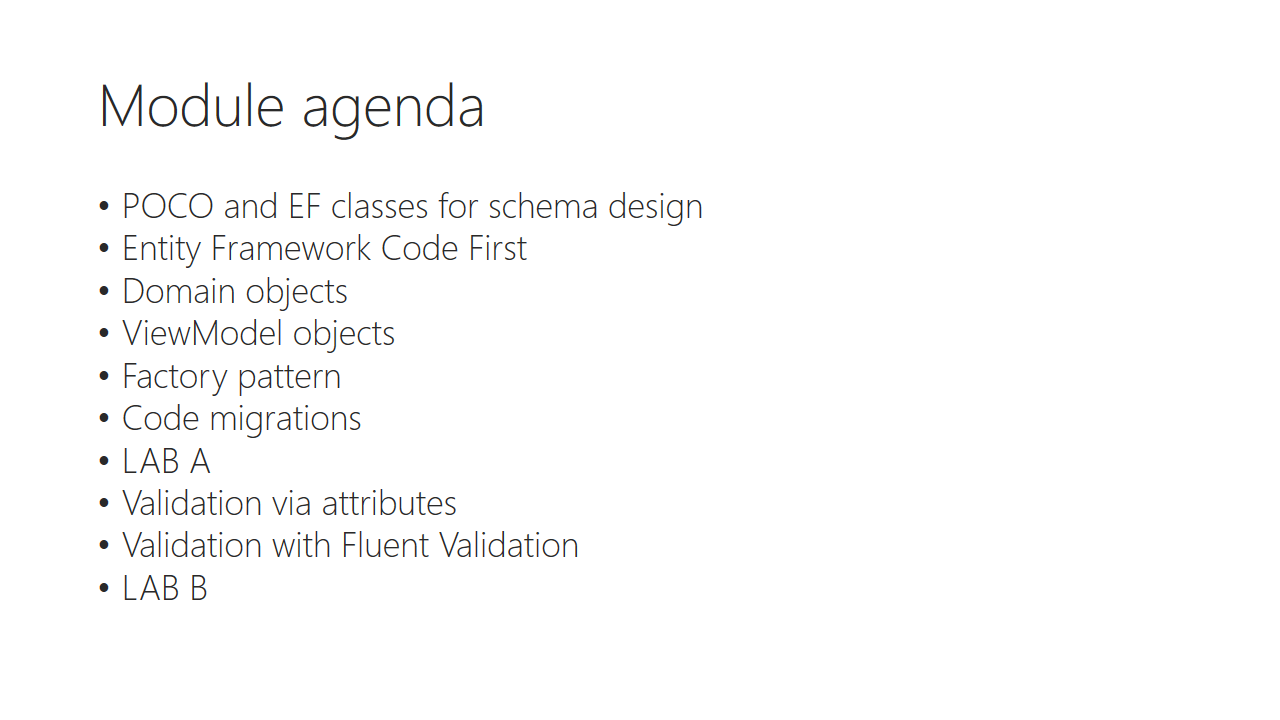
At this point we have ended with independent HomeController, what is very important for maintaining, testing and further development of new features.

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

## Lab 01 Summary

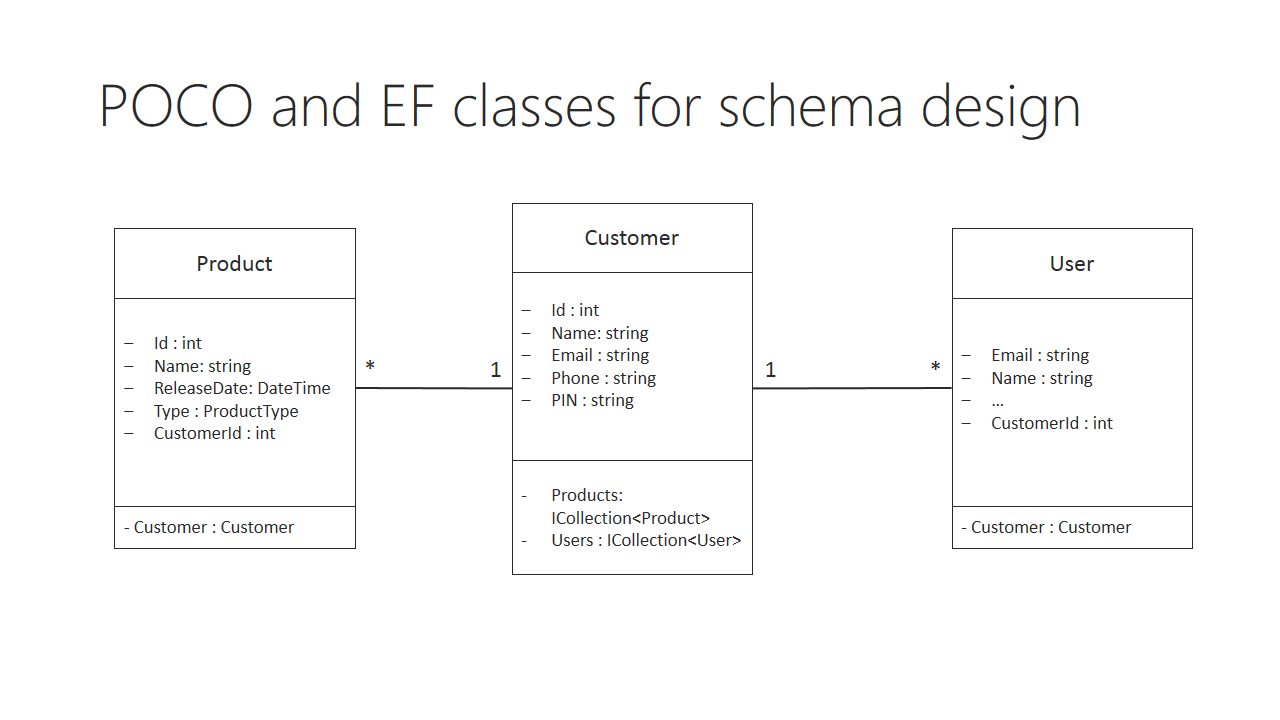
In this lab we have heavily refactored default MVC template to remove dependencies and use two standard development patterns – Repository and Unit of Work. We have ended with nice generic template that will work for many smaller and bigger projects. We will use it as a scaffold for out ticketing system.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

# Module 02 Models



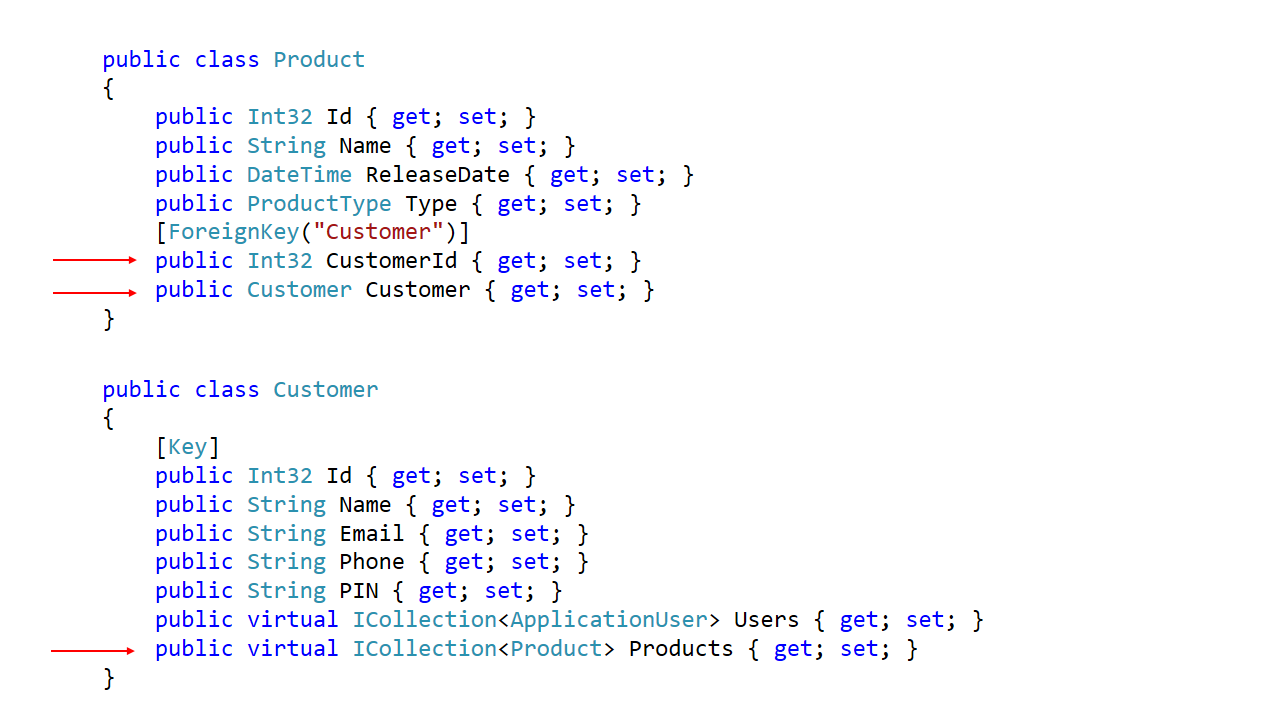
Trainer notes:

Student notes:



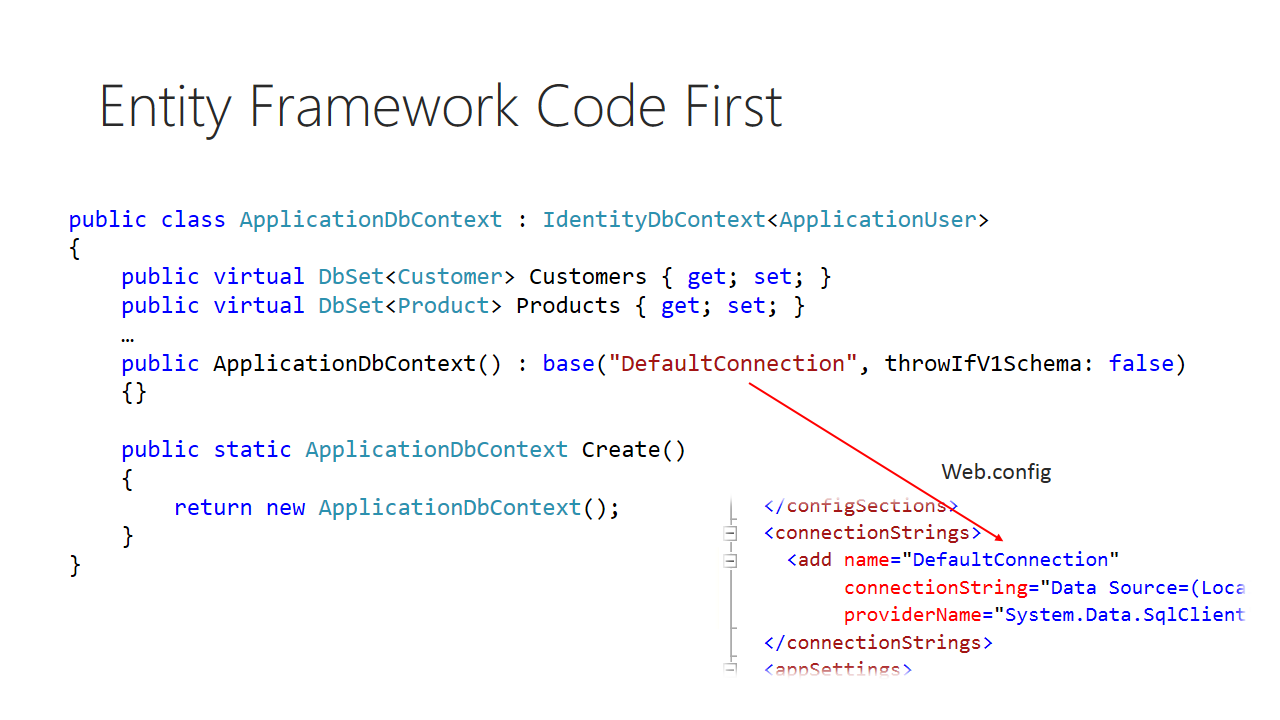
Trainer notes:

Student notes:



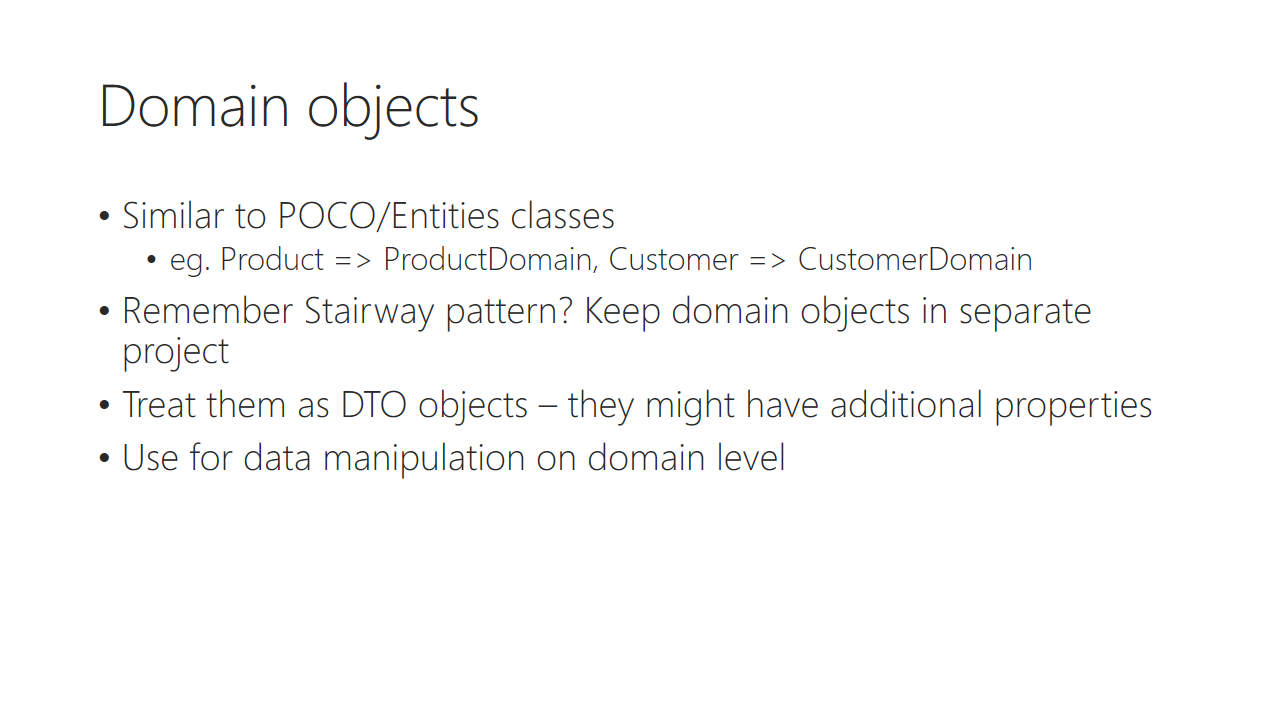
Trainer notes:

Student notes:



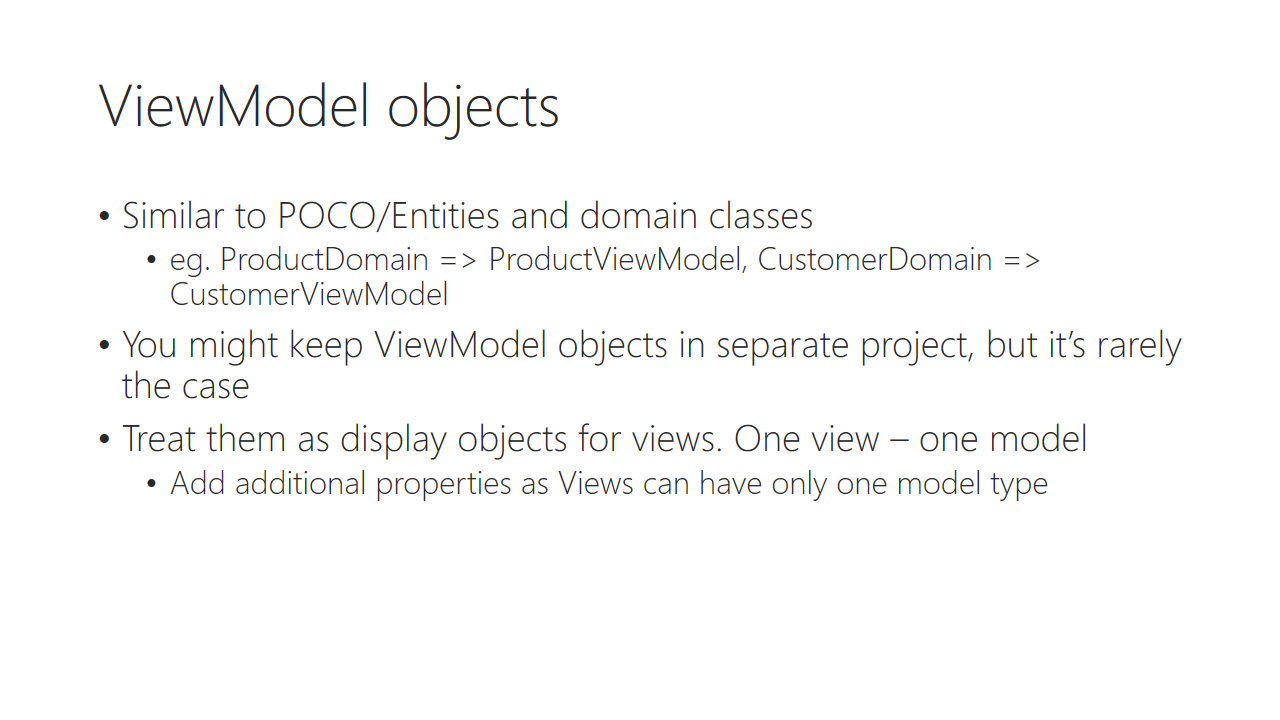
Trainer notes:

Student notes:



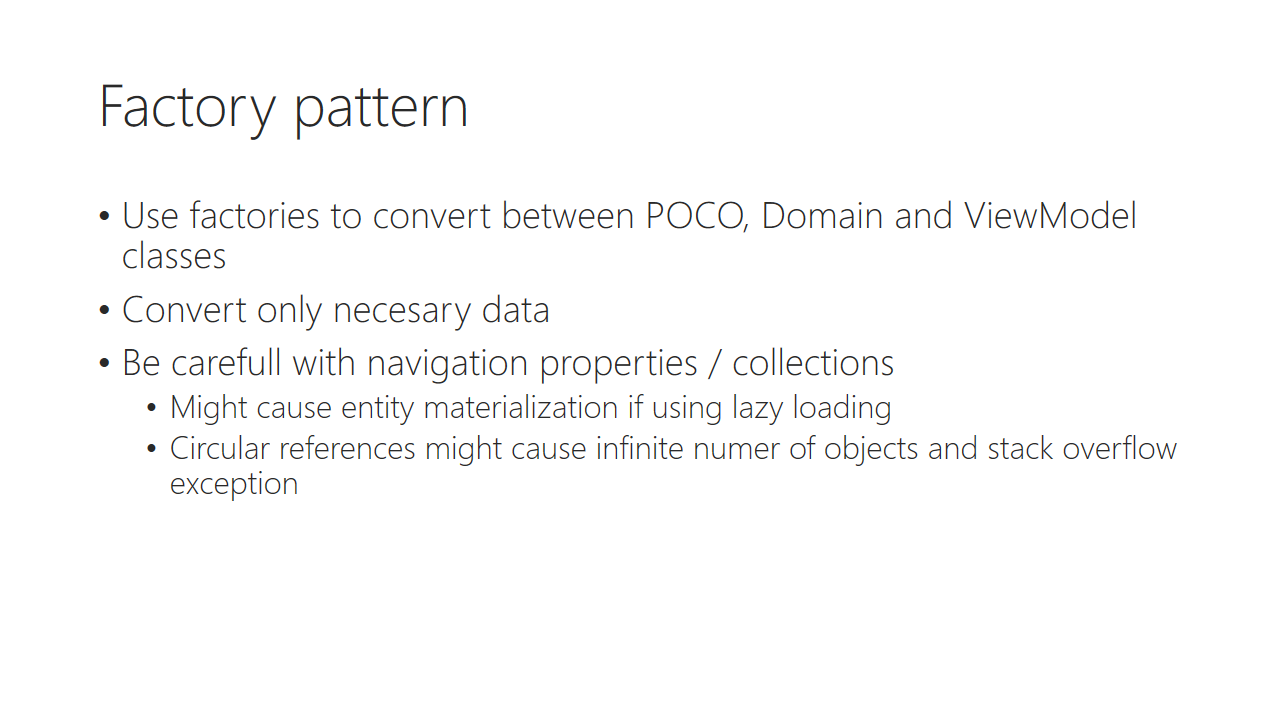
Trainer notes:

Student notes:



Trainer notes:

Student notes:



Trainer notes:

Student notes:



Trainer notes:

Student notes:

Discussion: Where you should keep factories?

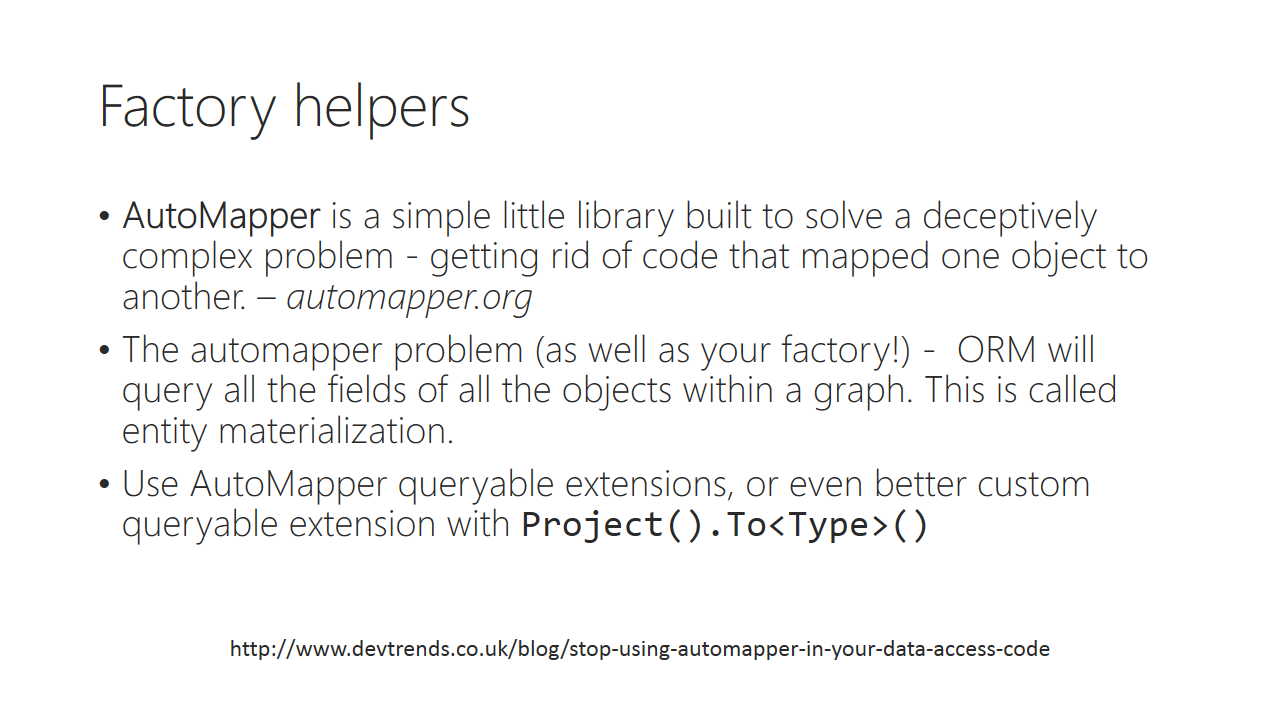
In separate project?

In domain service project?

In Domain object project?

Trainer notes:

Student notes:



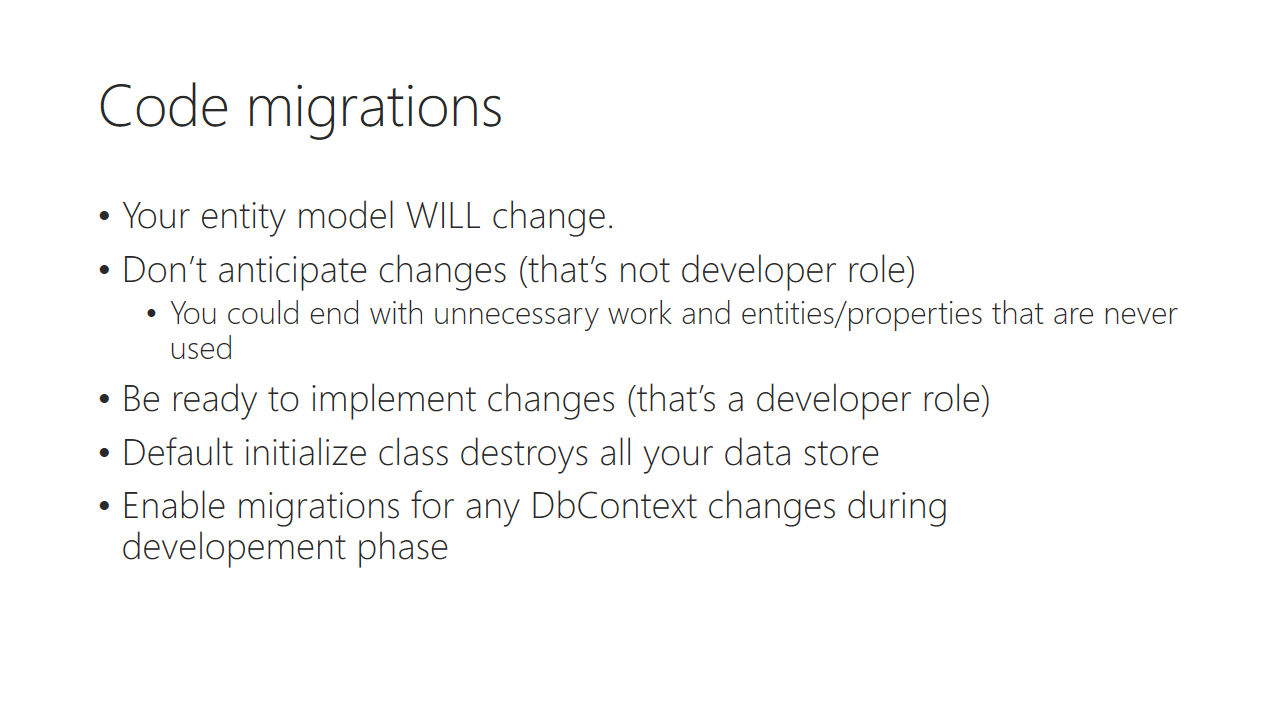
Trainer notes:

Student notes:

EF POCO => Domain => ViewModels

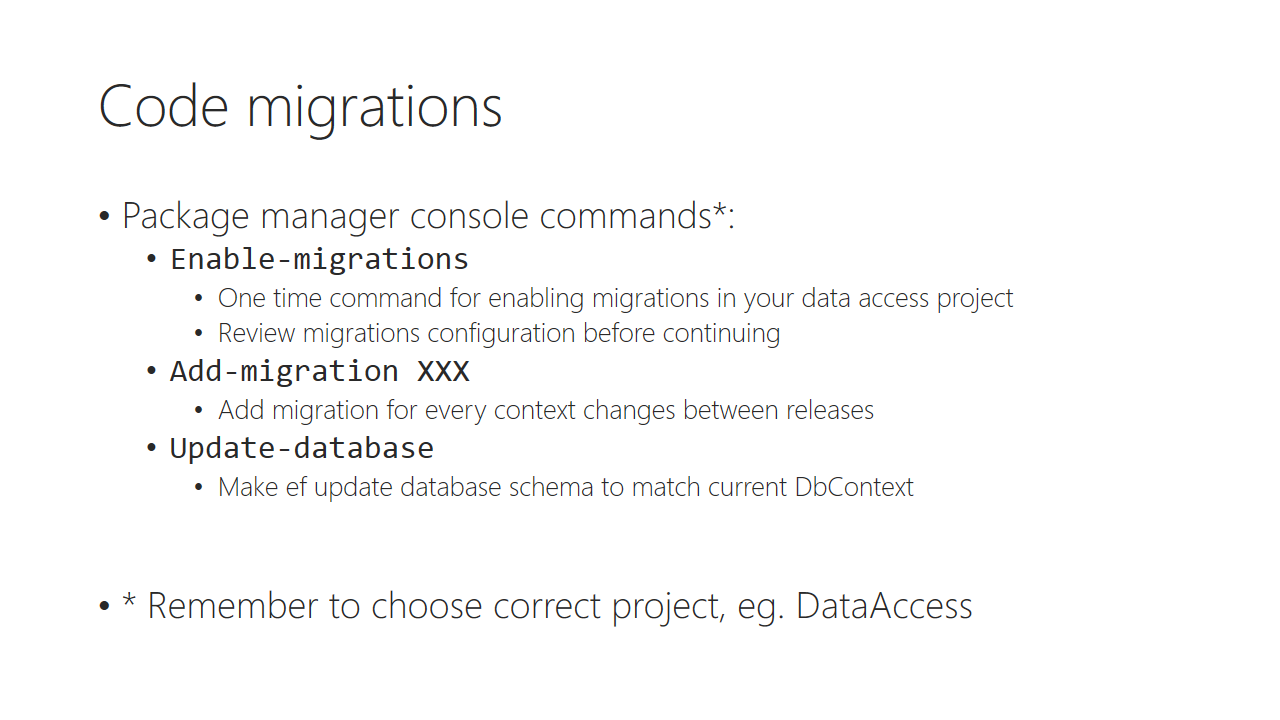
Trainer notes:

Student notes:



Trainer notes:

Student notes:



Trainer notes:

Student notes:

Demo: EF code migrations

Trainer notes:

Student notes:

# Lab 02 A

## 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. And still, as agile-lovers, we want to deliver usable piece of software on the end of this lab.

## 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. Add Entity Framework Code Migrations

## Task 1 – Creating Customer Data Layer

1. Add **Models** folder to **Komsky.Domain** project
2. Add new CustomerDomain class to Models folder [Snippet: CustomerDomain]
   1. Field type: Int32, name: Id
   2. Field type: String, name: Name
3. Add **Handlers** folder to **Komsky.Services** project
4. Add new IBaseHandler interface to Handlers folder. [Snippet: IBaseHandler]  
   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);

void Commit();

Task CommitAsync();

}

1. Implement this interface in a generic abstract 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 Dispose() method, and dispose DataFacade object in it.
   4. Implement Commit() method and call DataFacade.Commit() inside.
   5. Implement CommitAsync() method and return DataFacade.CommitAsync() inside.
   6. Keep rest of the methods as abstract
   7. [Snippet: BaseHandler]
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).
   2. [Snippet: CustomerHandler]
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 – Creating Customer Business Layer

1. Open CustomerHandler class in **Komsky.Services/Handlers** project and continue implementing required methods.

*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!*

1. Go to **Komsky.Services** project and create folder **Factories**
2. Add new public static class called CustomerDomainFactory to this location
3. 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.   
   [Snippet: CustomerHandlerAddUpdateDelete]. 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 – Creating 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<CustomerDomain> 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. Run through all the commented methods and implement them using \_customerHandler instead of db. Remember to call Commit() method after any change like creating, updating or deleting object. Full CustomerController class could look like this:

public class CustomerController : Controller

{

private readonly IBaseHandler<CustomerDomain> \_customerHandler;

public CustomerController(IBaseHandler<CustomerDomain> customerHandler)

{

\_customerHandler = customerHandler;

}

public ActionResult Index()

{

return View(\_customerHandler.GetAll().Select(CustomerViewModelFactory.Create));

}

public ActionResult Details(int? id)

{

if (id == null)

{

return new HttpStatusCodeResult(HttpStatusCode.BadRequest);

}

CustomerViewModel customerViewModel = \_customerHandler.GetById(id.Value).CreateCustomerViewModel();

if (customerViewModel == null)

{

return HttpNotFound();

}

return View(customerViewModel);

}

public ActionResult Create()

{

return View();

}

[HttpPost]

[ValidateAntiForgeryToken]

public ActionResult Create([Bind(Include = "Id,Name")] CustomerViewModel model)

{

if (ModelState.IsValid)

{

\_customerHandler.Add(model.CreateCustomerDomain());

\_customerHandler.Commit();

return RedirectToAction("Index");

}

return View(model);

}

public ActionResult Edit(int? id)

{

if (id == null)

{

return new HttpStatusCodeResult(HttpStatusCode.BadRequest);

}

CustomerViewModel customerViewModel = \_customerHandler.GetById(id.Value).CreateCustomerViewModel();

if (customerViewModel == null)

{

return HttpNotFound();

}

return View(customerViewModel);

}

[HttpPost]

[ValidateAntiForgeryToken]

public ActionResult Edit([Bind(Include = "Id,Name")] CustomerViewModel model)

{

if (ModelState.IsValid)

{

\_customerHandler.Update(model.CreateCustomerDomain());

\_customerHandler.Commit();

return RedirectToAction("Index");

}

return View(model);

}

public ActionResult Delete(int? id)

{

if (id == null)

{

return new HttpStatusCodeResult(HttpStatusCode.BadRequest);

}

CustomerViewModel customerViewModel = \_customerHandler.GetById(id.Value).CreateCustomerViewModel();

if (customerViewModel == null)

{

return HttpNotFound();

}

return View(customerViewModel);

}

[HttpPost, ActionName("Delete")]

[ValidateAntiForgeryToken]

public ActionResult DeleteConfirmed(int id)

{

\_customerHandler.Delete(id);

\_customerHandler.Commit();

return RedirectToAction("Index");

}

protected override void Dispose(bool disposing)

{

if (disposing)

{

\_customerHandler.Dispose();

}

base.Dispose(disposing);

}

}

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 – Entity Framework Code Migrations

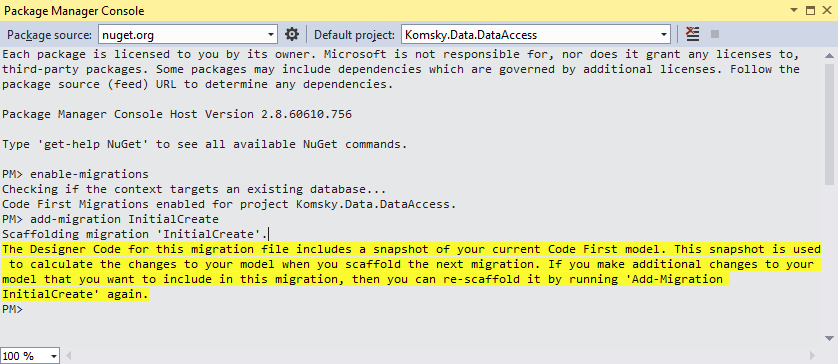
1. Open **Web.config** file from **Komsky.Web** project and edit *DefaultConnection* connection string:
   1. Remove *AttachDbFilename* section
   2. Rename *Initial Catalog* to *komskydb*
2. Open Package Manager Console and run following commands for **Komsky.Data.DataAccess** project:

PM> enable-migrations

…

PM> add-migration InitialCreate

* 1. After enabling migrations, stop and analyse **Configuration.cs** file that has been added to your project. You’ll find very important Seed() method, that will allow you to add necessary values to your Database, like first admin users, default products etc. For now, we will leave this method empty, but we might use it in the future.
  2. After adding InitialCreate migration, stop and analyse **xxx\_InitialCreate.cs** files content. You’ll find here two methods – Up() and Down(). Up() method will be run, when you want to migrate database to the newest version by running update-database command.



1. OK, now it’s time to update our database. Open again Package Manager Console and run update-database command.

PM> update-database

*If you have any problems with this command, read through your error message. You’ll might need to log in into (LocalDb)\v11.0 SQL Server instance via Management Studio, delete existing database and repeat update-database command.*

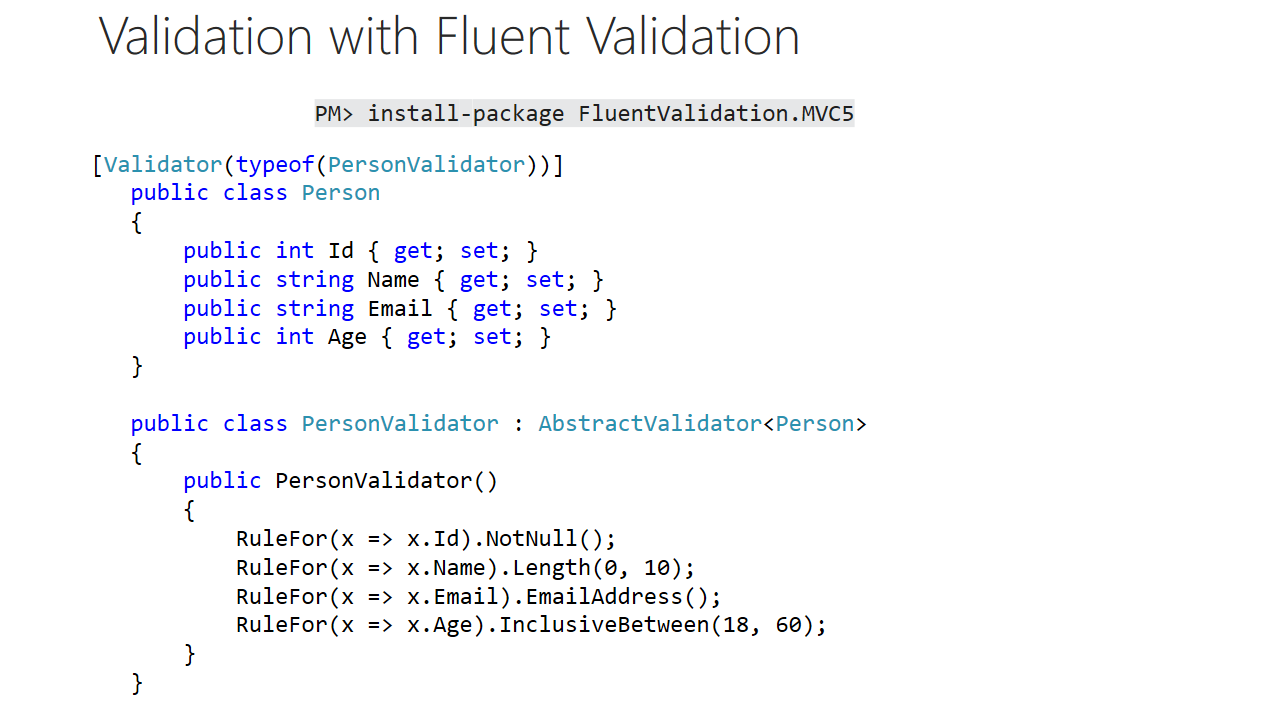
1. Build your project and run your application.
2. Navigate to Customers menu item to display Index – a customer’s list
3. The list is empty, but it works! Try to play with this new feature of your application – add, edit and delete some Customers.

At this point we have implemented most of the necessary layout and patterns, so we can focus on developing features required by clients. Think of the previous work as an investment – now, introducing new features, maintaining and working in a team to make this app better will be easier!



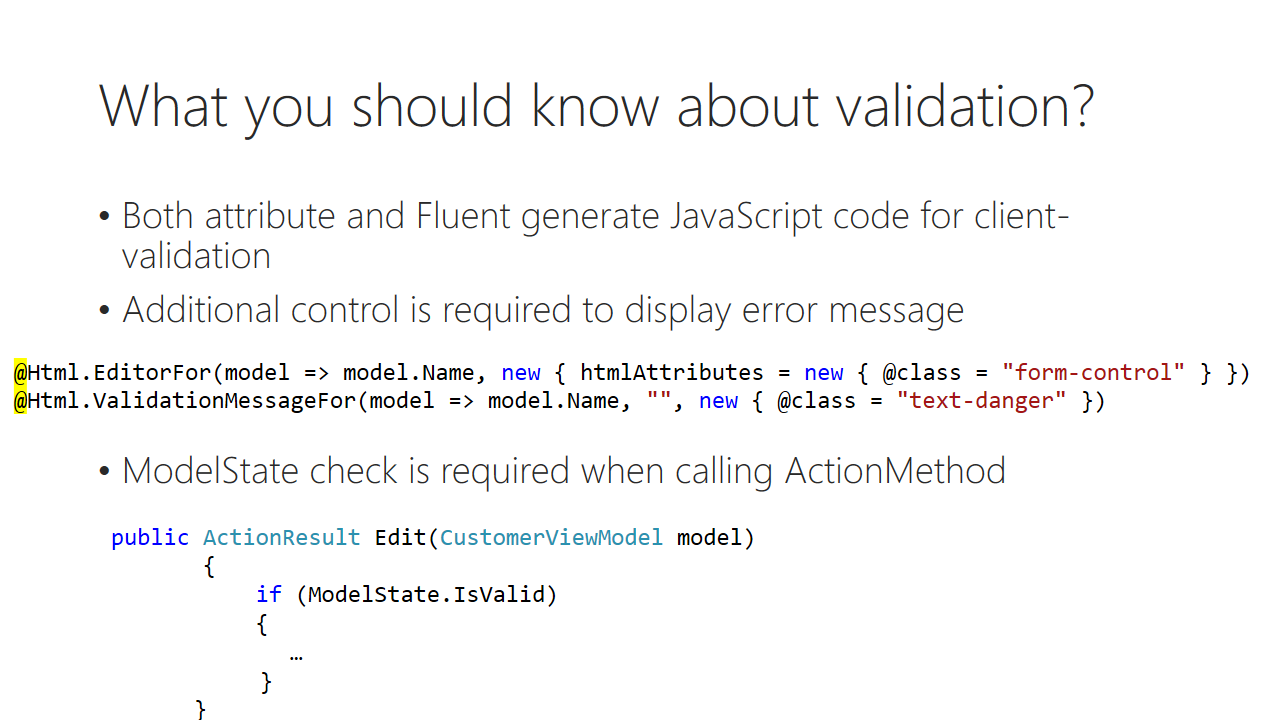
Trainer notes:

Student notes:



Trainer notes:

Student notes:



Trainer notes:

Student notes:



Trainer notes:

Student notes:

Demo: Model validation

Trainer notes:

Student notes:

# Lab 02 B

## Content:

This part of Lab focuses on entity validation by using attributes as well as fluent validation

## General steps:

1. Add new properties to Customer class
2. Validate new properties using attributes
3. Validate new properties using Fluent Validation

## Task 1 – Add new properties to customer class

1. Go to **Komsky.Data.Entities** and add new POCO class called Product. Add following properties:
   1. Int32 Id
   2. String Name
   3. DateTime ReleaseDate
   4. ProductType Type (this is enum, see instructions below)
      1. Add new Enum to Enums folder called ProductType with values: Software, Hardware, Other
   5. Customer Customer
2. Open the Customer definition and add following properties
   1. String Email
   2. String Phone
   3. String PIN
   4. virtual ICollection<Product> Products
3. Go to **Komsky.Data.DataAccess** and add new DbSet with products to ApplicationDbContext

public virtual DbSet<Product> Products { get; set; }

1. Go to **Repositories** folder and add empty IProductRepository interface, implementing IRepository<Product>

public interface IProductRepository : IRepository<Product> {}

1. Add new ProductRepository class, inheriting from GenericRepository<Product> and implementing IProductRepository

public class ProductRepository : GenericRepository<Product>, IProductRepository

{

public ProductRepository(DbContext dbContext) : base(dbContext)

{

}

}

1. Go to **IDataFacade.cs** file in **UnitOfWork** folder, and add ProductRepository to this interface.

ProductRepository Products { get; }

1. Implement this property in DataFacade class the same way as CustomerRepository.

<The code is not provided here on purpose. Please try implementing this by yourself, it should be easy by now. In case of trouble, ask your fellow student or trainer. On the second though – don’t ask trainer, he’s busy reading news, please check the solution project ☺ >

1. Use the *Package Manager Console* to add Products migration to your database.

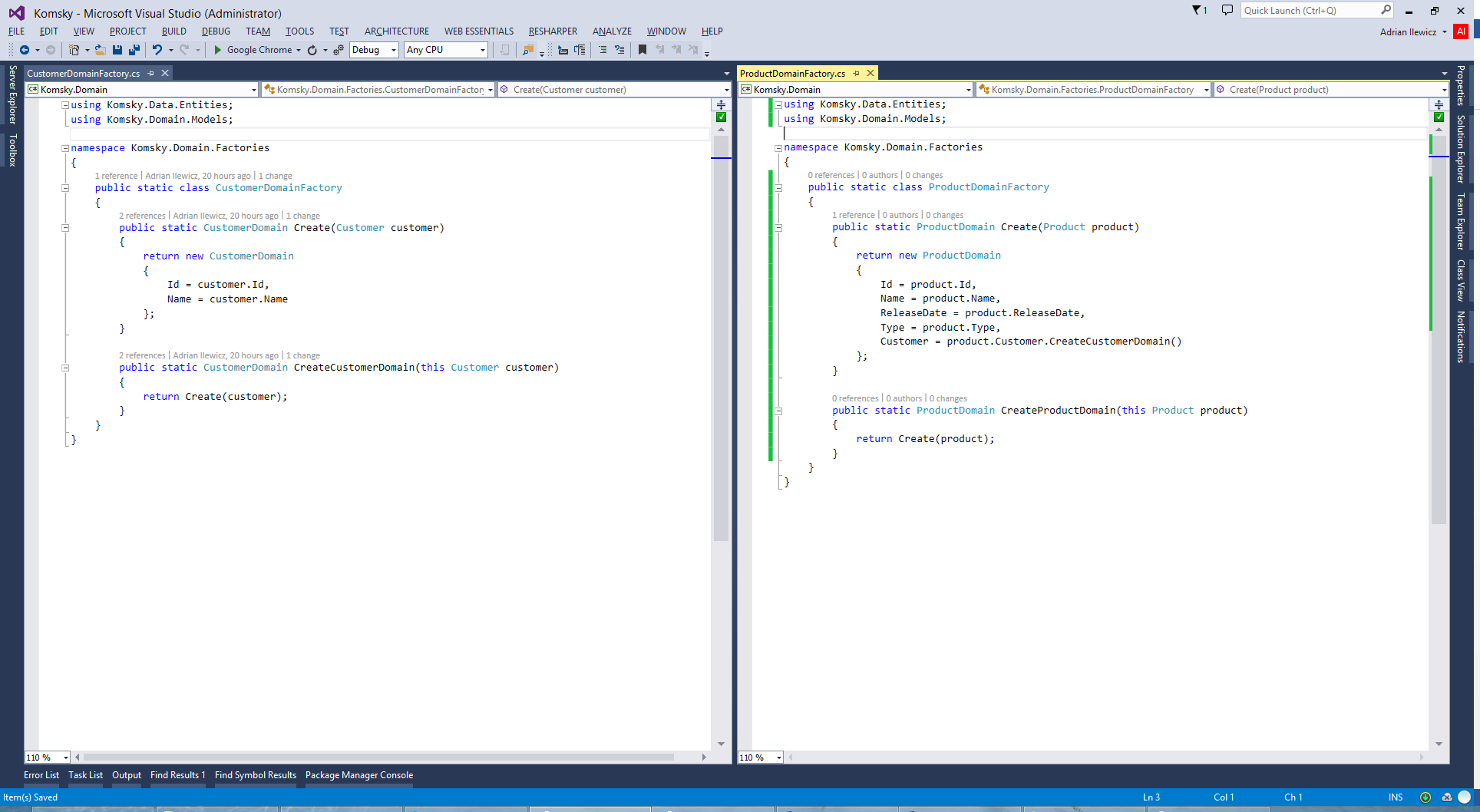
PM> add-migration Products

1. Update database schema with update-database command

PM> update-database

1. Go to **Komsky.Domain** project and add ProductDomain class to **Models** folder mimicking Product entity
   1. You can use the same ProductType enum for Type
2. Update CustomerDomain class adding list of products. Use IEnumerable<ProductDomain> collection type.
3. Add two new factories to **Komsky.Services/Factories** project folder – Product factory and Product domain factory. Try mimicking existing CustomerDomainFactory for ProductDomainFactory and CustomerFactory for ProductFactory.

<I’m pretty sure you’ll handle this task without any code reference here. One small hint – to create CustomerDomain from Customer object use existing CreateCustomerDomain extension method from CustomerDomainFactory. Use VisualStudio split code view to help yourself.>



1. Update CustomerViewModel class by adding missing members from CustomerDomain class.
2. Update Customer factories to correctly create Customer objects.
3. Create ProductHandler class in **Komsky.Services/Handlers** project folder. Inherit from BaseHandler<ProductDomain> and implement all abstract method. You can mimic existing CustomerHandler class.
4. Bind IBaseHandler<ProductDomain> with ProductHandler class in **NinjectWebCommon.cs** RegisterServices method.
5. Go to **Komsky.Web/Models** project folder and add ProductViewModel class. Implement necessary properties.
6. Go to **Komsky.Web/Models/Factories** project folder and add ProductViewModelFactory as well as ProductDomainFactory. Implement functionalities for both of those classes.
7. Build your project and debug any errors.

Our business layer is ready to handle product requests. At this point we have usable ProductHandler, ready to handle our object modifications requests. Please make sure you’ve understood what we have coded here – it’s a standard way to implement handler in our solution, and soon we will implement them even more.

## Task 2 – Validate Customer class using attributes

1. Build, run your project and navigate to Customers list.
2. Go ahead, and create a customer without any name (leave Name property empty). Why is this possible, to create a customer without any name? We need to fix this.
3. Navigate to CustomerViewModel class in **CustomerViewModel.cs** file. Add [Required] attribute just above Name property declaration. The attribute comes from System.ComponentModel.DataAnnotations namespace. Code should look like this:

**[Required]**

public String Name { get; set; }

1. Build and run your app with debbuging. Try again to create empty customer. Is it still possible?
2. Don’t stop debugging. Navigate to **CustomerController.cs** file and find Create method with [HttpPost] attribute and CustomerViewModel model parameter. Hit F9 to add a breakpoint and again try to add new empty customer. Does breakpoint hit? How to explain that behaviour?
3. Stop debugging.
4. Finish implementing missing fields from Customer and CustomerDomain classes. Browse for and add appropriate attributes for those properties. You can use [Snippet:CustomerViewModel] find a full list here:

<https://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.aspx>

public class CustomerViewModel

{

public Int32 Id { get; set; }

[Required]

[MinLength(3)]

public String Name { get; set; }

[Required]

[EmailAddress]

public String Email { get; set; }

[Required]

[Phone]

public String Phone { get; set; }

[StringLength(8, MinimumLength = 4)]

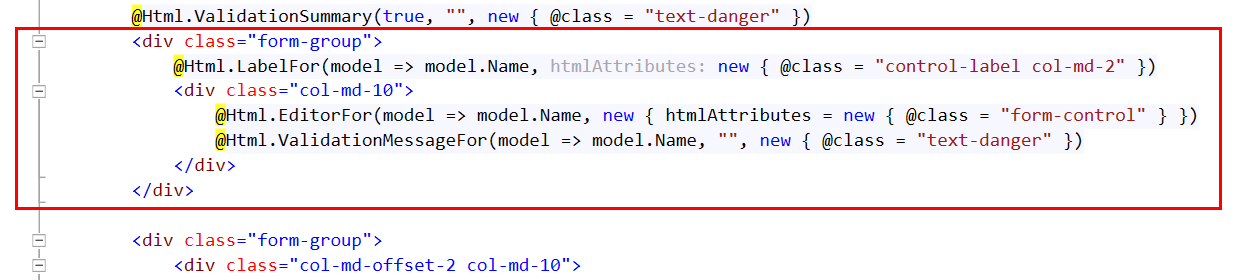
public String PIN { get; set; }

public IEnumerable<ApplicationUser> Users { get; set; }

public IEnumerable<ProductViewModel> Products { get; set; }

}

1. Unfortunately that’s not all. We still need to display and validate those fields on the client. Navigate to **Views/Customer/Create.cshtm** View file.
2. Within this file, copy div section marked by form-group class, that contain display and edit code for Name property. When still marked and copied into clipboard, paste this four times, each for one property in ViewModel (Name, Email, Phone and PIN).



1. Update code to display those other fields instead of Name. Remember to change all three Html Helper methods, that is LabelFor, EditorFor and ValidationMessageFor
2. Remove Bind directive from Create method signature to allow those new fields be binded into model.
3. Build and run you code. Navigate to Customers and add new customer. Try inserting invalid data. Play with valid and invalid values.

## Task 3 – Validate Product class using Fluent Validation

1. Open Package Manager Console and install FluentValidation.MVC5 package for **Komsky.Web** project.

PM> install-package FluentValidation.MVC5 -Version 5.6.2

*If you are using Visual Studio 2015 you can skip the version info*

1. Add following line (as the last) to Application\_Start() method in **Global.asax** file

protected void Application\_Start()

{

...

**FluentValidationModelValidatorProvider.Configure();**

}

1. Navigate to **ProductViewModel.cs** file in **Models** folder.
2. Add new class in this file called ProductViewModelValidator, and inherit from AbstractValidator<ProductViewModel> generic class.
3. Implement default constructor with RuleFor(delegate) entry for each respective validation rule. [Snippet:ProductViewModelValidator] and example below:

public class ProductViewModelValidator : AbstractValidator<ProductViewModel>

{

public ProductViewModelValidator()

{

RuleFor(x => x.Name).NotEmpty();

...

RuleFor(x => x).Must(CustomeRule);

}

private bool CustomeRule(ProductViewModel arg)

{

//validate ProductViewModel in any way you like here

return true;

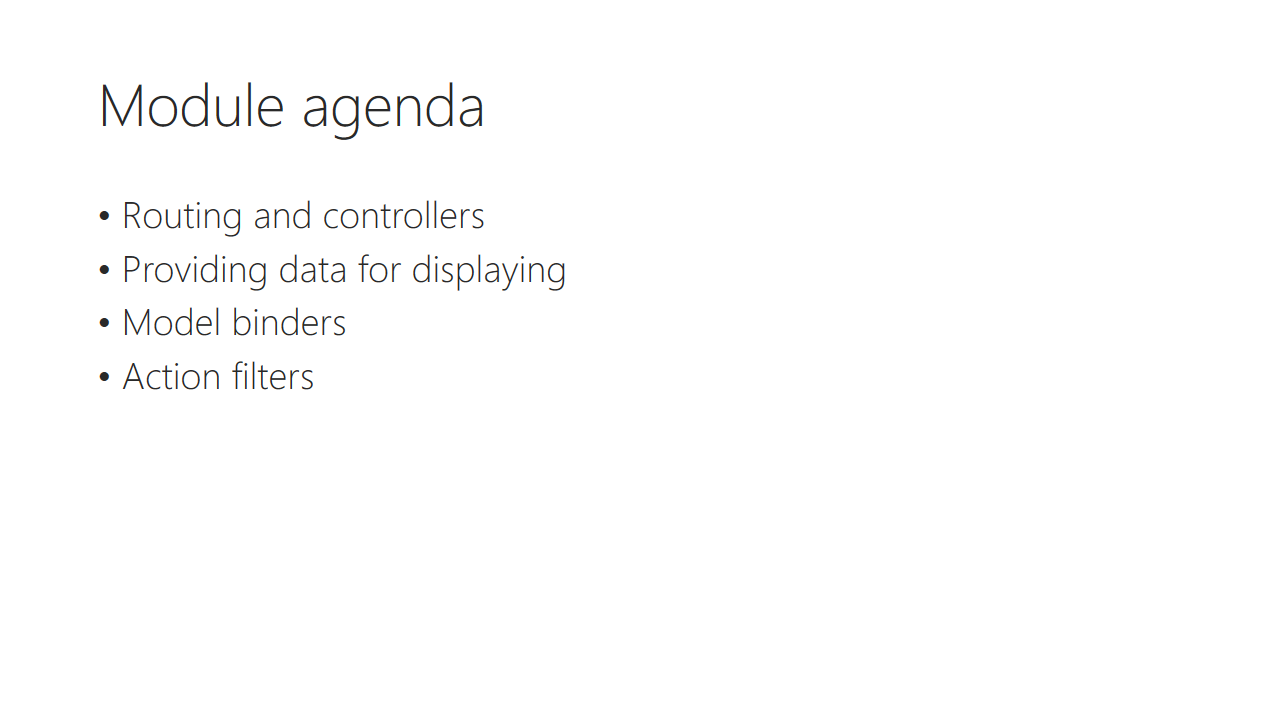
}

}

1. Decorate ProductViewModel class with [Validator(typeof(ProductViewModelValidator))] attribute
2. Build your app. Solution is ready.
3. If you have finished before group, use spare time to adding some customers objects into database. You might also add some products for those customers, but since we don’t have Controller class for this type yet, you can only achieve this by using Server Explorer window in Visual Studio.

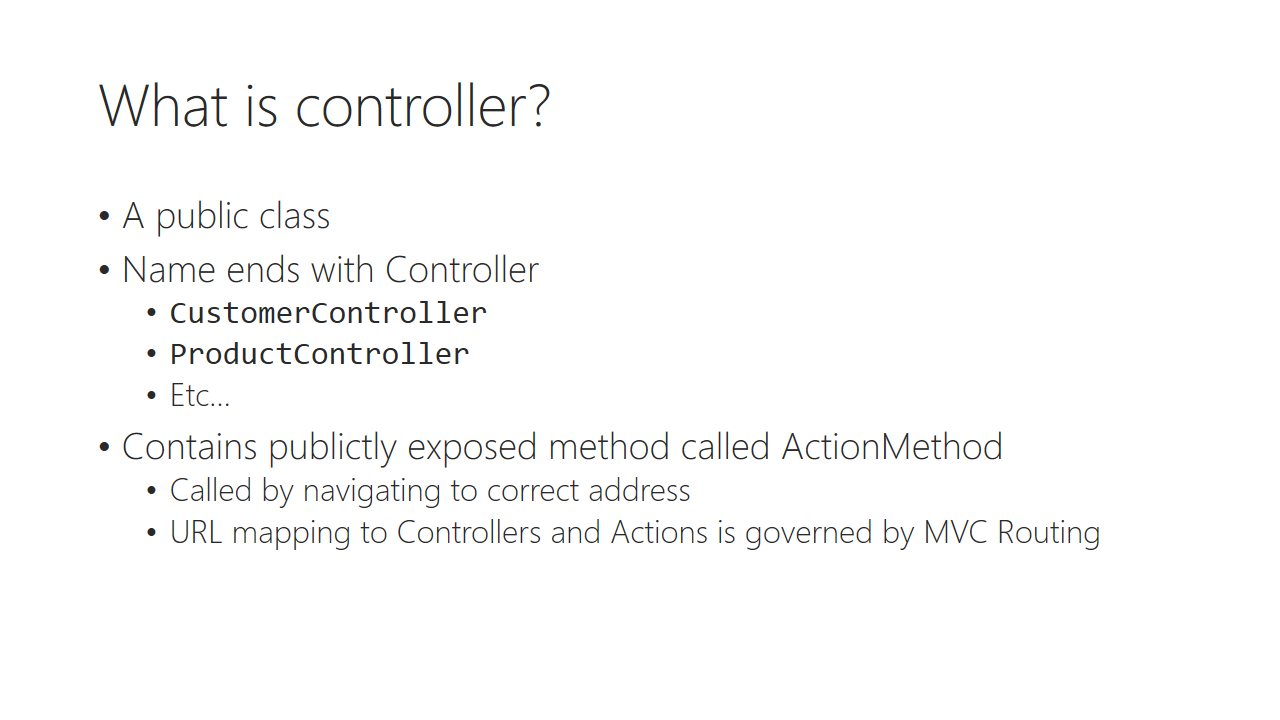
At this point we have working application with simple validation on both client and server. We will test Product validation later on, because we first need to create ProductController.

# Module 03 Controllers



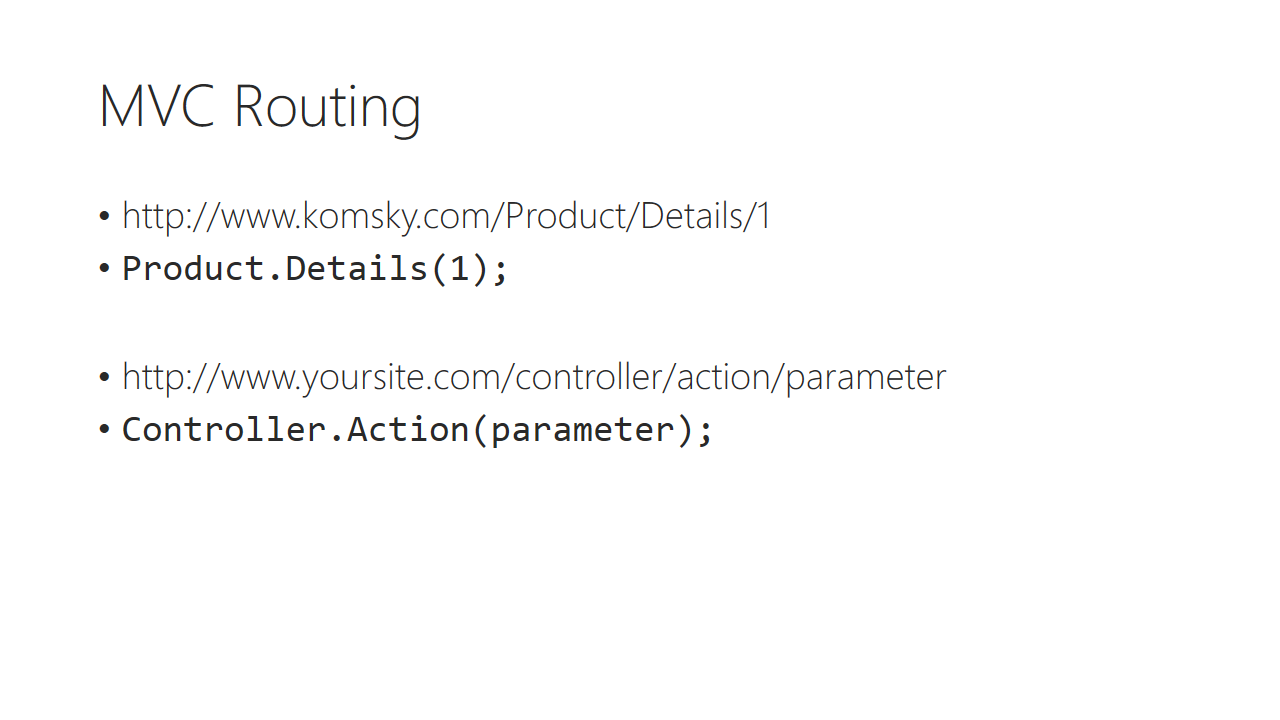
Trainer notes:

Student notes:



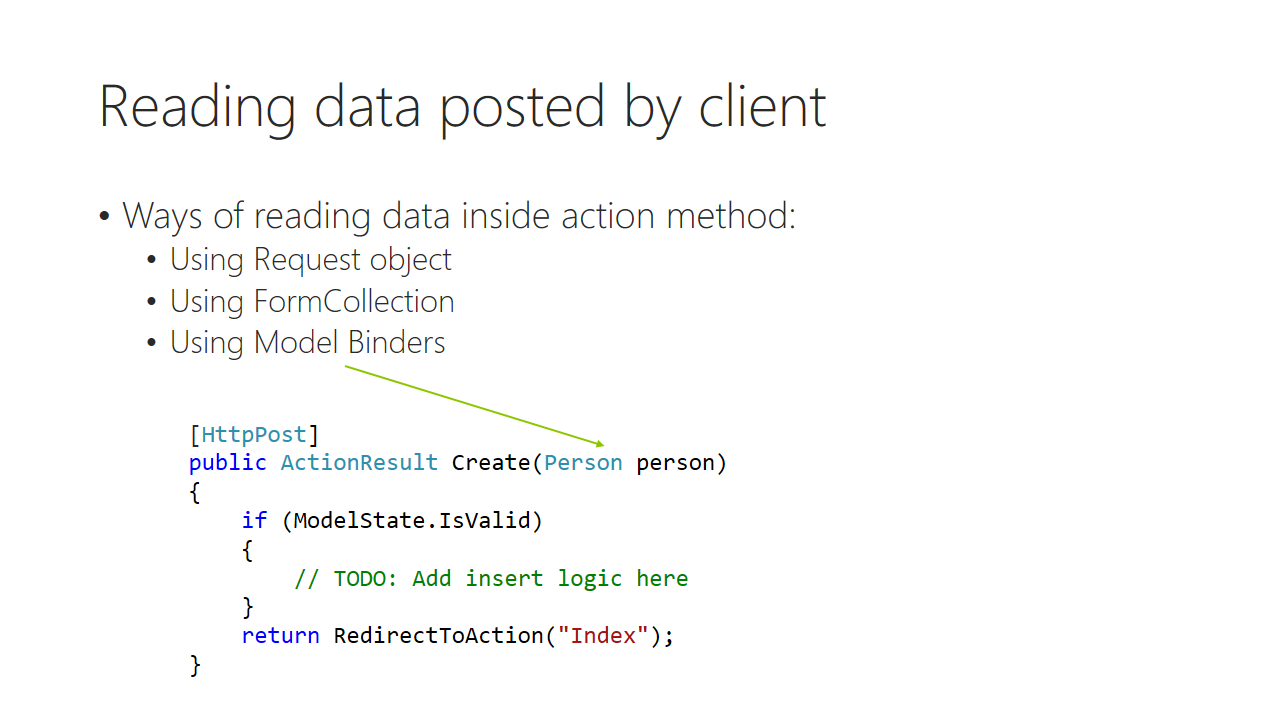
Trainer notes:

Student notes:



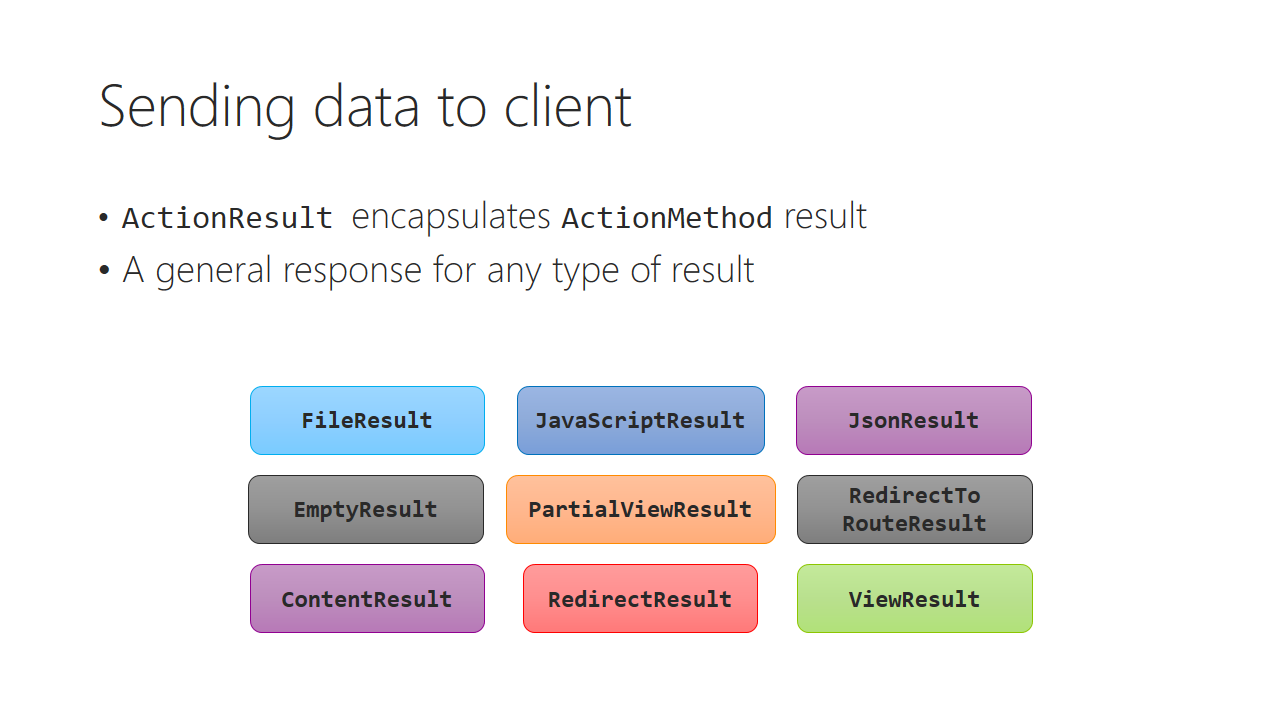
Trainer notes:

Student notes:



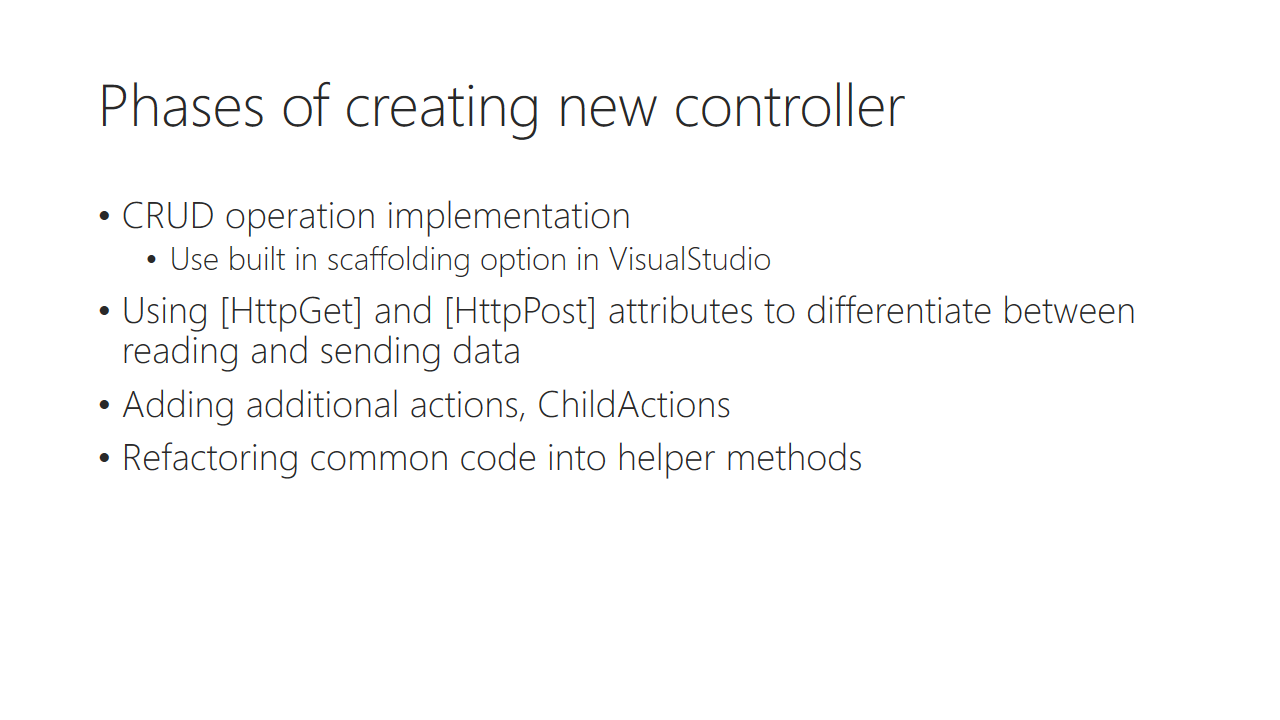
Trainer notes:

Student notes:



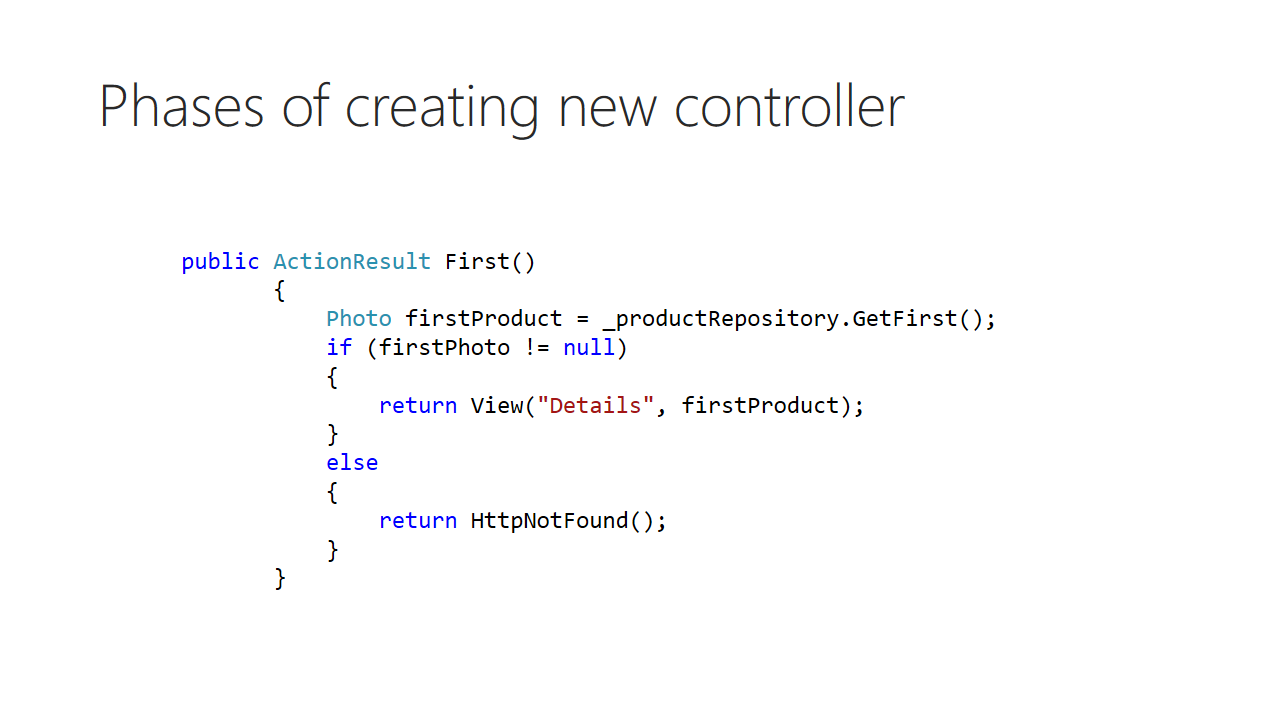
Trainer notes:

Student notes:



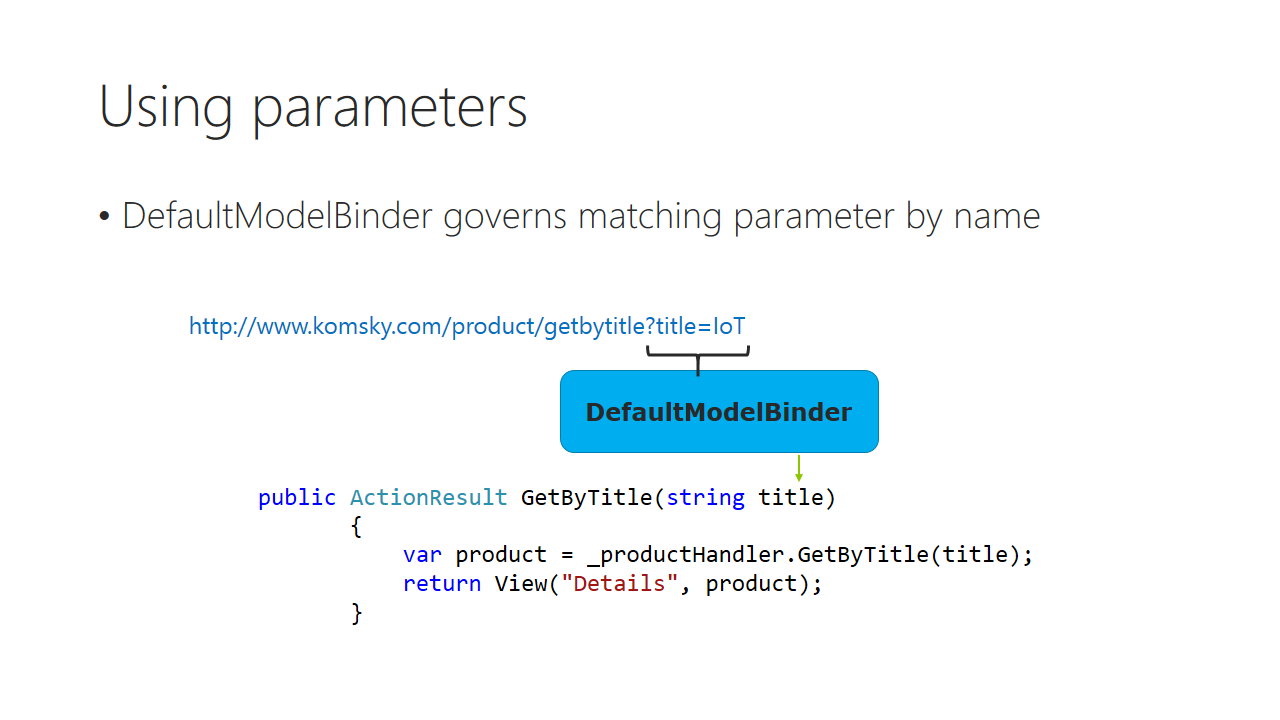
Trainer notes:

Student notes:



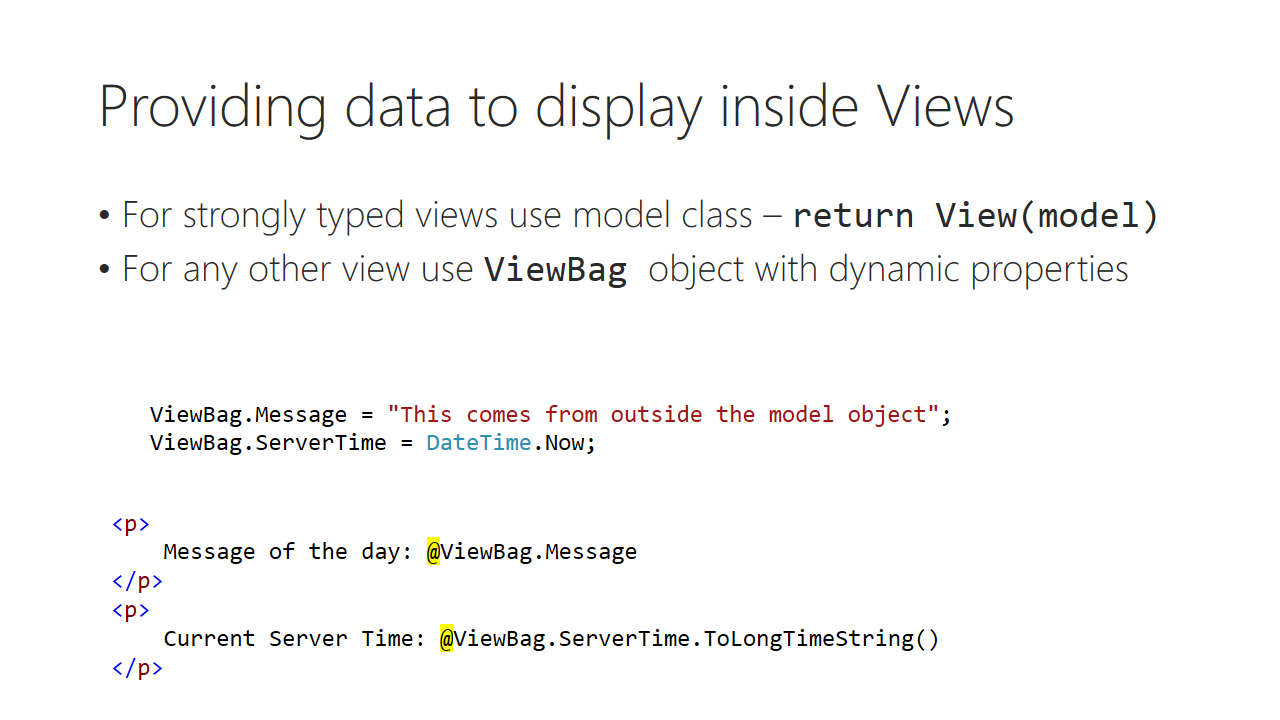
Trainer notes:

Student notes:



Trainer notes:

Student notes:



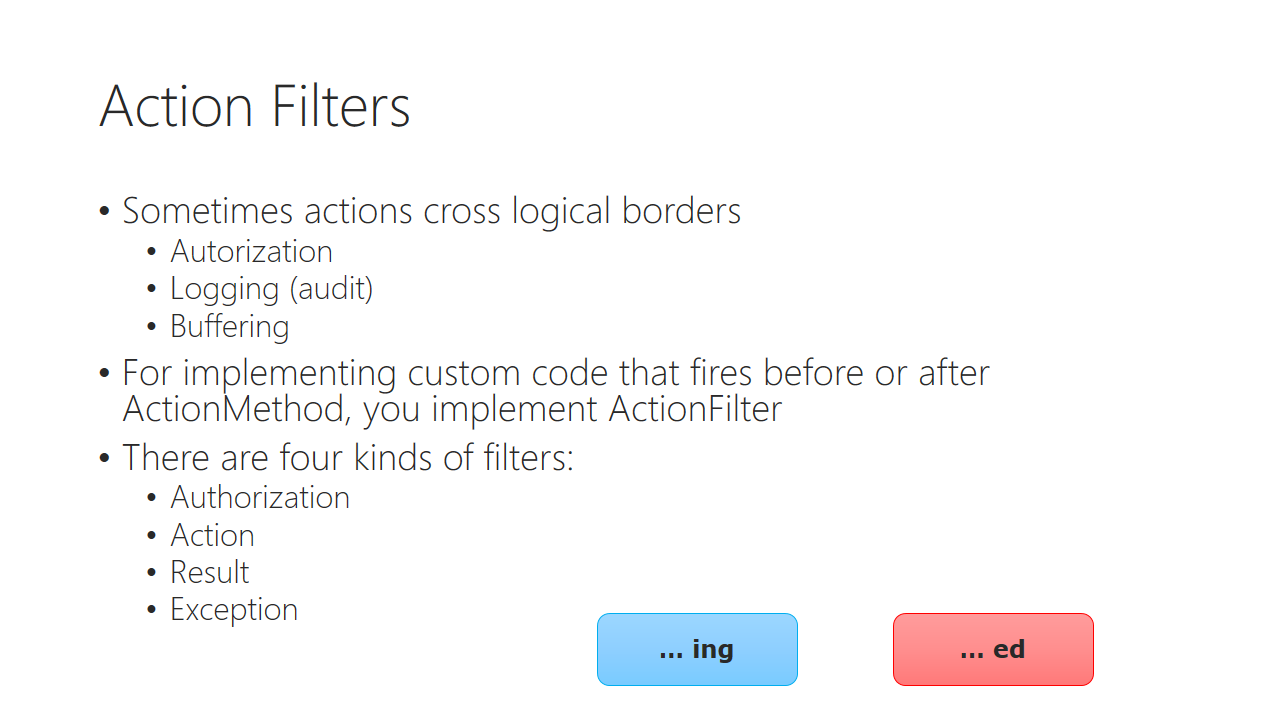
Trainer notes:

Student notes:

Demo: Scaffolding new controller

Trainer notes:

Student notes:



Trainer notes:

Student notes:



Trainer notes:

Student notes:

Discussion: When you might need a custom filter?

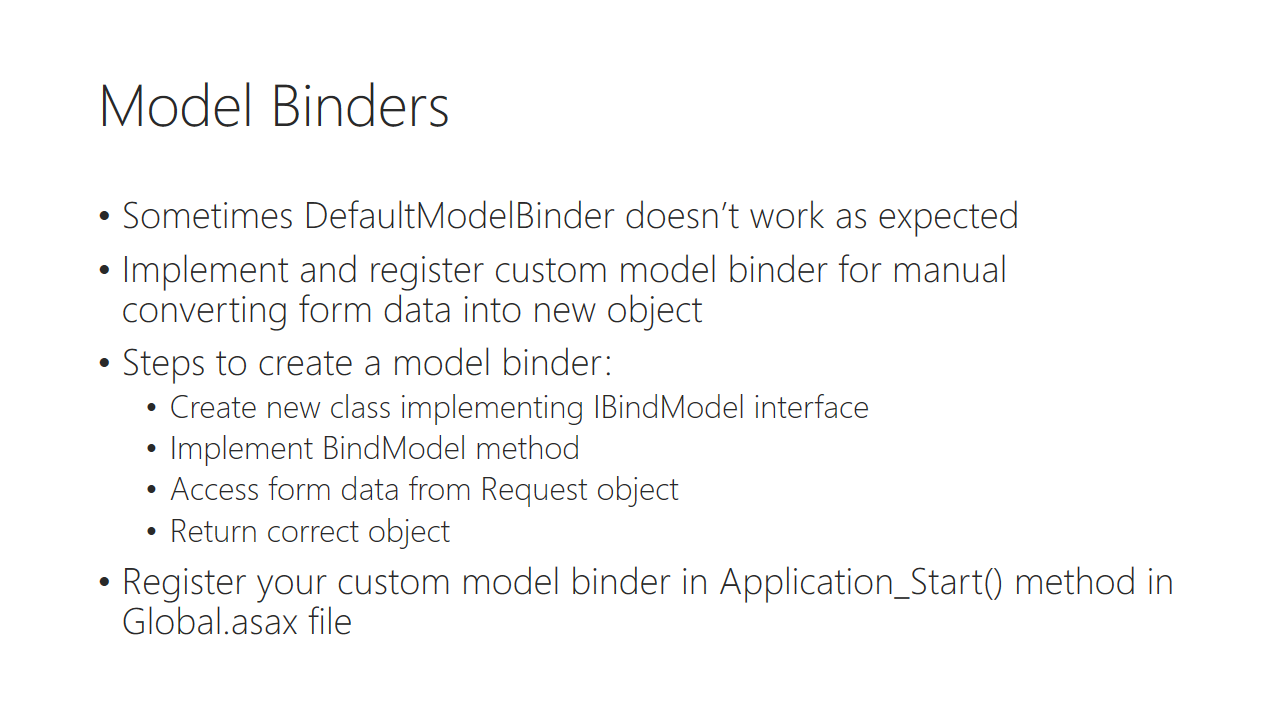
Trainer notes:

Student notes:

Demo: Custom action filters

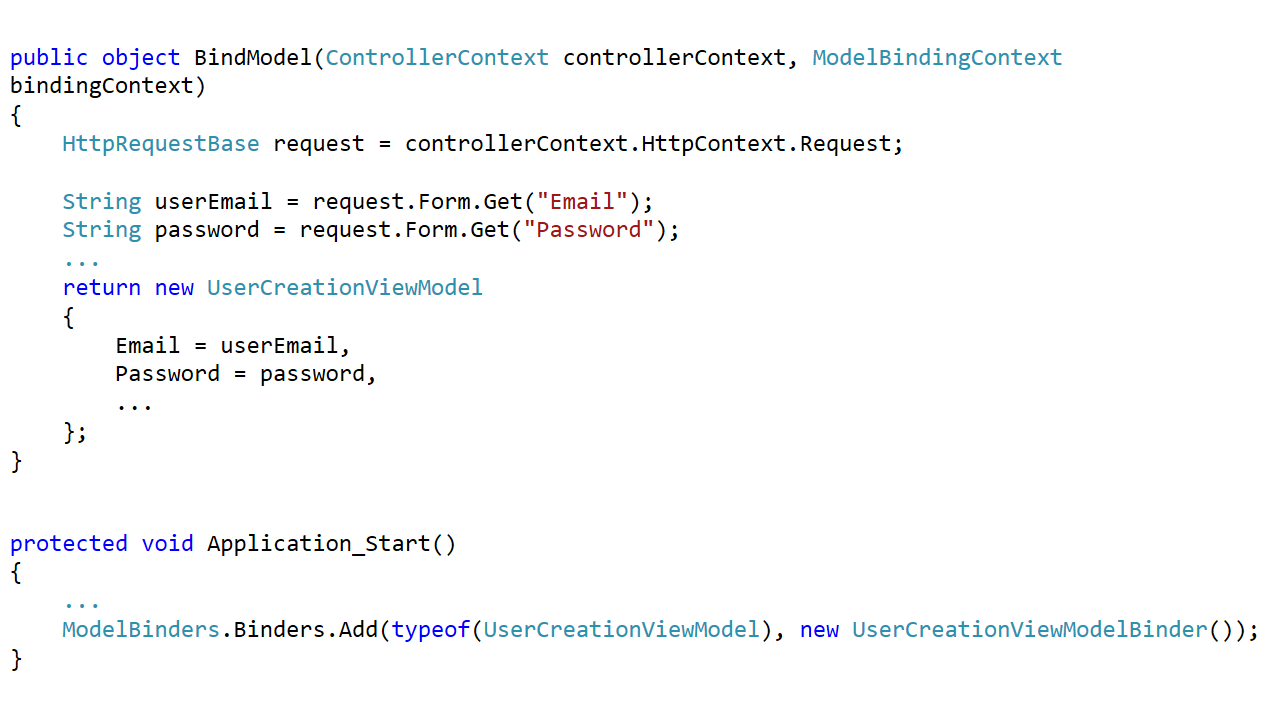
Trainer notes:

Student notes:



Trainer notes:

Student notes:



Trainer notes:

Student notes:

Demo: Custom model binders

Trainer notes:

Student notes:

# Lab 03

## Content

In this lab we focus on developing new Ticket controller for products, so that a ticket might be created for particular product. We also implement new route for searching tickets by ID

## General steps

1. Creating new Ticket entity with repository and handler
2. Creating new Ticket controller for handling ticket management
3. Implementing product controller class
4. Refactoring tickets for allowing product choose with drop down list
5. Creating custom model binder

## Task 1 – Create new Ticket class, provide repository and handler for this type

1. Create new folder in your solution called Common.
2. Add new Class Library project to this folder called **Komsky.Enums**
3. Add new enum type to this project called TicketState, with following options:
   1. Created
   2. Assigned
   3. InProgress
   4. Solved
   5. Reopened
   6. [Snippet:TicketState]
4. Add new enum type called TicketPriority, with following options:
   1. Normal
   2. Highest
   3. High
   4. Low
   5. Lowest
   6. [Snippet:TicketPriority]
5. Navigate to **Komsky.Data.Entities** project, and add reference to **Komsky.Enums**
6. Move ProductType enum to **Komsky.Entites**. Delete **Komsky.Data.Entites/Enums** folder. Update namespaces where applicable, build and debug any errors.
7. Create new class called Ticket in **Komsky.Data.Entites** project. Add following properties to it:
   1. Id : int
   2. Title : String
   3. Description: String
   4. TicketPriority: TicketPriority
   5. TicketState: TicketState
   6. AgentReply: String
   7. OwnerId : String. Decorate with [ForeignKey("Owner")]
   8. Owner: ApplicationUser
   9. AssignedAgentId: string; Decorate with [ForeignKey("AssignedAgent")]
   10. AssignedAgent: ApplicationUser
   11. [Snippet:TicketEntity]
8. Navigate to **Komsky.Data.DataAccess** project and add DbSet<Ticket> Tickets property to ApplicationDbContext class
9. Create empty ITicketRepository interface, inheriting from IRepository<Ticket>
10. Implement this new interface in TicketRepository class
    1. Inherit from GenericRepository<Ticket>
    2. Implement ITicketRepository
    3. See other repositories to check if your code is correct
11. Add your new ticket repository to IDataFacade interface, and implement it correctly in DataFacade class
12. Navigate to **Komsky.Domain** project, and reference **Komsky.Enums** if you haven’t already when you were refactoring enums)
13. Add TicketDomain class with similar properties as original Ticket POCO class. [Snippet:TicketDomain]
14. Because we haven’t still removed Identity dependency, temporarily we must reference **Komsky.Data.Entities** in **Komsky.Domain** project. We will remove this later.
    1. Komsky.Data.Entites require Microsoft.AspNet.Identity.EntityFramework to be referenced. Update this as well by installing Microsoft.AspNet.Identity.EntityFramework to this project. Right now you’ve just experienced entourage problem first hand.
15. Navigate to **Komsky.Services/Handlers** folder and add new TicketHandler class
16. Inherit from BaseHandler<TicketDomain> class and implement all methods. You’ll need two new factories – Ticket and TicketDomain. Implement them as well.
    1. Don’t hesitate to see how other factories and handlers were implemented. This code is very repeatable. [Snippet:TicketHandler]
17. Open the Package Manager Console, make sure the **DataAccess** project is selected and run following commands:
    1. add-migration Tickets
    2. Update-database

At this point we have working data and business layer. Now’s the time to create UX for that feature.

## Task 2 – Create Ticket Controller with model class and Views

1. Navigate to **Komsky.Web/Models** folder and add new TicketViewModel class. Implement it the same way as TicketDomain class.[Snippet:TicketViewModel]
2. Decorate TicketViewModel properties with [DisplayName("xxx")] attribute, providing text, that will be displayed as field name in views, eg. AgentReply => Agent’s Reply, etc. [Snippet has already did it for you]
3. Implement appropriate factories for this type. [Snippet:TicketViewModelFactories – they are created in one file for simplicity]
4. Scaffold new TicketController with **Add** => **Controller** => **MVC 5 Controller with read/write actions**
5. Add TicketHandler handler the ‘dependency injection way’. Just to remind
   1. IBaseHandler<TicketDomain> interface only
   2. Initialize via public constructor
   3. [Snippet: TicketControllerTicketHandler]
   4. Bind to class using Ninject Bind() method.
6. Go to **Views/Ticket** folder. Add following views, using TicketViewModel model class:
   1. Name: Index, Template: List
   2. Name: Details, Template: Details
   3. Name: Create, Template: Create
   4. Name: Edit, Template: Edit
7. Go back to TicketController and implement action methods in the following way [Snippet:TicketControllerMethods – replace all generated methods]:
   1. Index => Get all Tickets using \_ticketHandler, convert them to ViewModel types and pass to View method.

public ActionResult Index()

{

var model = \_ticketHandler.GetAll().Select(TicketViewModelFactory.Create);

return View(model);

}

* 1. Details => use \_ticketHandler to get ticket by id, convert it to ViewModel type and pass to View method
  2. Create() => leave empty
  3. Create(FormCollection collection) => this is more complex. Follow points bellow:
     1. Change input parameter to accept (TicketViewModel model) parameter
     2. Validate the model using ModelState.IsValid property. If not, just return View(model)
     3. If model state is valid, convert model into ModelDomain type, add to \_ticketHandler and commit changes, then navigate to Index method by using RedirectToAction method.

public ActionResult Create(TicketViewModel model)

{

if (ModelState.IsValid)

{

\_ticketHandler.Add(model.CreateTicketDomain());

\_ticketHandler.Commit();

return RedirectToAction("Index");

}

return View(model);

}

* 1. Edit(int id) => Find this ticket by id, convert to domain and pass to view.

public ActionResult Edit(int id)

{

var model = \_ticketHandler.GetById(id);

if (model != null)

{

return View(model.CreateTicketViewModel());

}

return HttpNotFound("Ticket not found");

}

* 1. Edit(int id, FormCollection collection) => Similar to Create.
     1. Change signature to accept ViewModel class
     2. Validate model
     3. Save and return to Index

public ActionResult Edit(TicketViewModel model)

{

if (ModelState.IsValid)

{

\_ticketHandler.Update(model.CreateTicketDomain());

\_ticketHandler.Commit();

return RedirectToAction("Index");

}

return View(model);

}

* 1. Delete (id) => similar to Details

public ActionResult Delete(int id)

{

var model = \_ticketHandler.GetById(id).CreateTicketViewModel();

return View(model);

}

* 1. Delete(id, collection) => Change signature to accept ViewModel and just delete entity by id

public ActionResult Delete(TicketViewModel model)

{

\_ticketHandler.Delete(model.Id);

\_ticketHandler.Commit();

return RedirectToAction("Index");

}

1. Build your solution and debug any errors.
2. Navigate to **Views/Shared/\_Layout.cshtml** file and add new link to ticket controller

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

1. Run and test your solution.
2. If you have finished before the rest of the group, decorate TicketViewModel with validation attributes. If you still have bunch of time, implement ProductController

At this point we have working ticketing solution with CRUD operations. Unfortunately, some properties are not yet usable, and some are missing. Let’s fix some of those issues in next task.

## Task 3 – Implement Product controller

1. Mimicking previous task, implement ProductController. [Snippet: ProductController]

*What issue have you encountered? Probably you’re unable to create Product without providing CustomerId, but how regular user can know CustomerId? We need to rewrite create method using dropdown list. We will use Html.DropDownList helper method, that will be discussed in next module, but we desperately need it right now!*

1. Navigate to **Views/Product/Create.cshtml** file
2. Find section responsible for rendering CustomerId field, and change EditorFor into DropDownListFor method call. This method has quite different signature, so just rewrite code from below:

@Html.DropDownListFor(model => model.CustomerId, new SelectList(Model.AllCustomers, "Id", "Name", Model.CustomerId), new { @class = "form-control" })

*As you can see, this method requires from us to deliver list of all customers. We need to implement this in model and controller.*

1. Navigate to ProductViewModel definition and add new property of type IEnumerable<CustomerViewModel> named AllCustomers
2. Navigate to ProductController Create() method. We need to create a model with AllCustomers and pass this into View.
3. Because we don’t have CustomerHandler in product controller, add new handler called \_customerHandler. Update constructor respectively.

private readonly IBaseHandler<ProductDomain> \_productHandler;

**private readonly IBaseHandler<CustomerDomain> \_customerHandler;**

public ProductController(IBaseHandler<ProductDomain> productHandler, **IBaseHandler<CustomerDomain> customerHandler** )

{

\_productHandler = productHandler;

**\_customerHandler = customerHandler;**

}

1. Update Create() method to fill the model.AllCustomers property:

public ActionResult Create()

{

ProductViewModel model = new ProductViewModel();

model.AllCustomers = \_customerHandler.GetAll()

.Select(CustomerViewModelFactory.Create);

return View(model);

}

1. Build, debug and run your solution. Assuming you have at least one customer, you should be able to create new product using drop down list with customer list.
2. Update the same way Edit(int id) action and view.
3. Test your solution.

## Task 4 – Refactor Ticket feature to allow choosing a product when creating a ticket

1. Navigate to **Komsky.Data.Entities** and connect Ticket with Product by adding new properties in Ticket class:

[ForeignKey("Product")]

public int? ProductId { get; set; }

public Product Product { get; set; }

1. Add new migration and update database (see task 1 point 17)
2. Update domain class and factories respectively.
3. Update ViewModel class and factories respectively.
4. Navigate to TicketViewModel definition and add IEnumerable<ProductViewModel> AllProducts property.
5. Go to TicketController and in Create() method create new instance of TicketViewModel class called model.
6. Fill model.AllProducts with \_productsHandler.
   1. You will need to create \_productsHandler field. Implement it the DI way, that is the same as \_ticketHandler

public ActionResult Create()

{

TicketViewModel model = new TicketViewModel();

model.AllProducts = \_productHandler.GetAll().Select(ProductViewModelFactory.Create);

return View(model);

}

1. Go to **Create.cshtml** in **Views/Ticket** folder. Remove form-group DIV’s for AgentReply, OwnerId and AssignedAgentName. Only Title and description should be left. Don’t close the window tab yet, we will get back here in a minute.
2. Go back to TicketController, to Create(model) method. Add Bind directive to method signature with Title and Description. This makes sure, that nobody will inject values for the fields we have just deleted.
3. Just before adding model to \_ticketHandler, assign current user’s ID as OwnerId. The code should look like this:

public ActionResult Create([Bind(Include = "Title, Description")] TicketViewModel model)

{

if (ModelState.IsValid)

{

model.OwnerId = User.Identity.GetUserId();

\_ticketHandler.Add(model.CreateTicketDomain());

\_ticketHandler.Commit();

return RedirectToAction("Index");

}

return View(model);

}

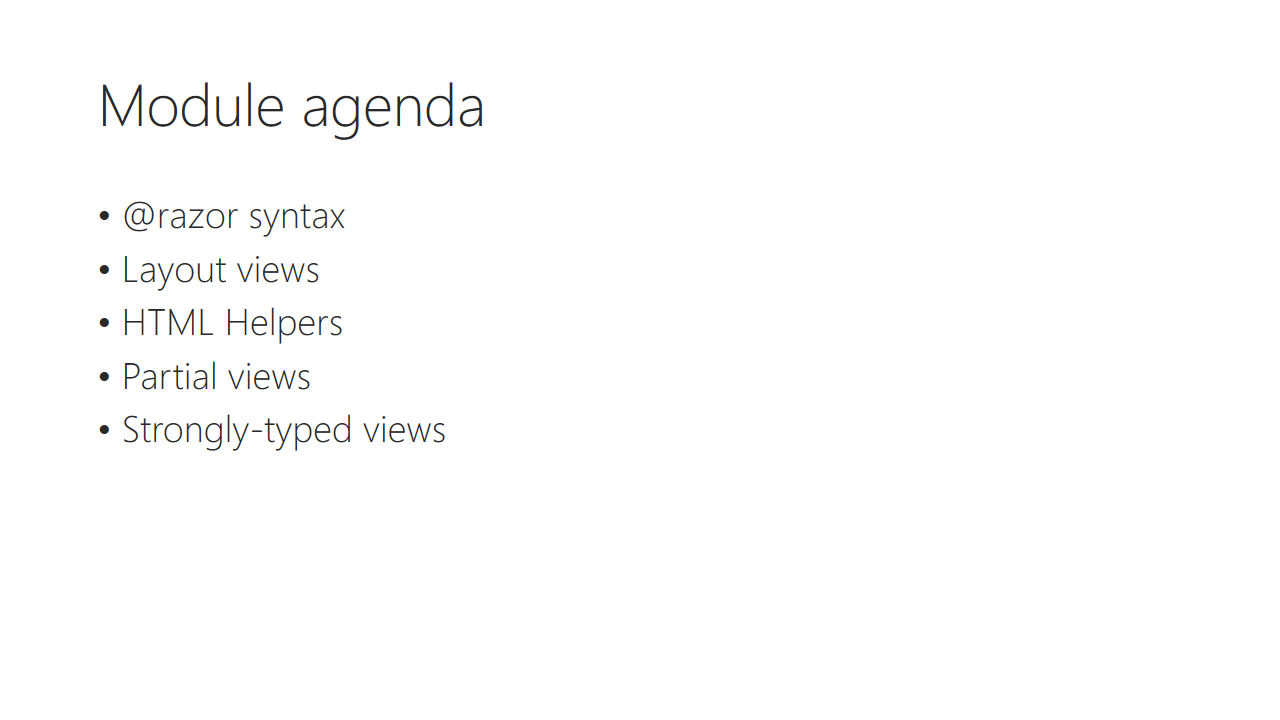
1. Build, debug and run your solution. Test adding new tickets to the system, as this should be possible now.

*Make sure, that you first register and login to the system, as without user’s ID this code will fail. To make sure that you don’t access it without authentication, add* [Authorize] *attribute on top of the* TicketController *class. We will discuss this attribute in module 9.*

1. Go back to **Create.cshtml** file of **Ticket** folder. Add new form-group div with dropdown list for app products. Mimic **Product/Create.html** file by copying-paste DropDownListFor AllCustomers.
2. Modify **Index.cshtml** view file, to display current product id for a ticket, by adding additional row to table with correct header and content.
3. Build and debug any errors.
4. Go ahead and test your application. Start by making sure, that you are registered and logged in user. Add some customers and add some products to database. Next, create a ticket for a product. Does it display correct product id?
5. Fix the Create method signature, by adding ProductId property name to Bind attribute. This will fix the problem. Test application again.

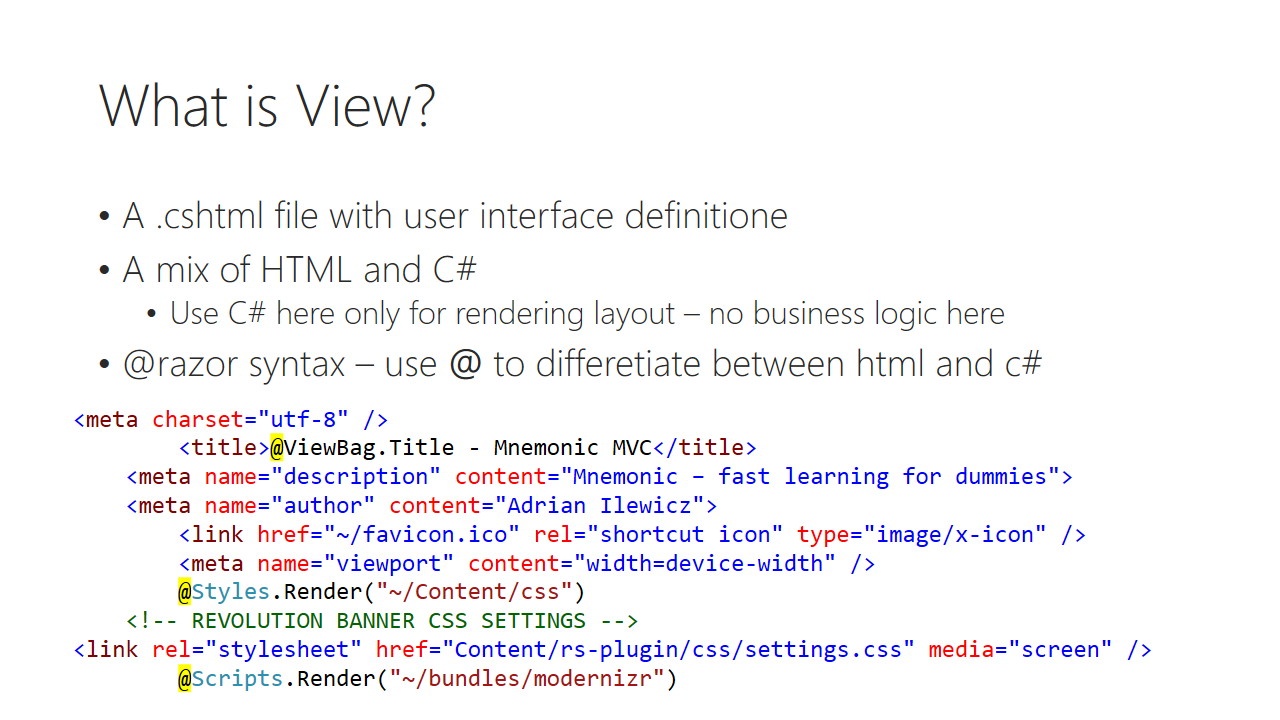
At this point we have created ticket controller, that allows for management tickets. This normally does not end’s work with controllers, but you have learned the basics. Usually the next step would be making sure that details and edit views allows for appropriate displaying and editing capabilities (eg. Edit should display product list in form of a dropdown as well, and Details view should provide with product name instead of a id. If you have finished before the rest of the group, please implement those).

# Module 04 Views



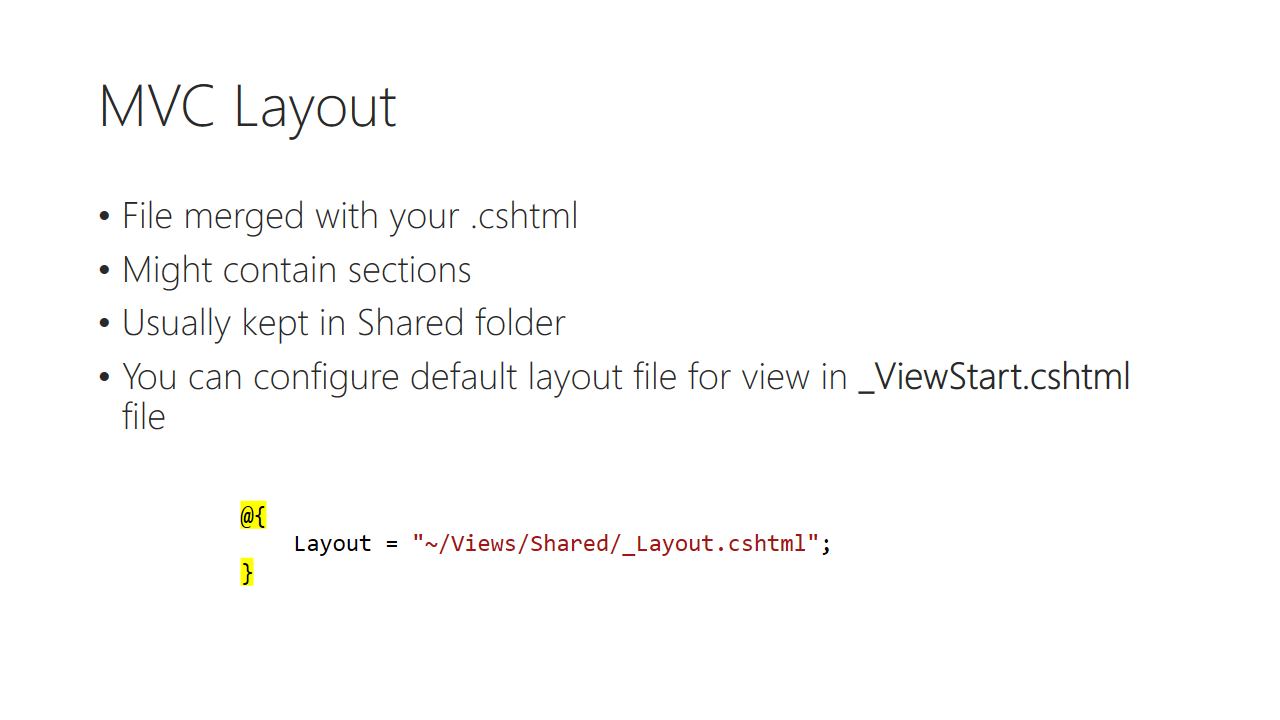
Trainer notes:

Student notes:



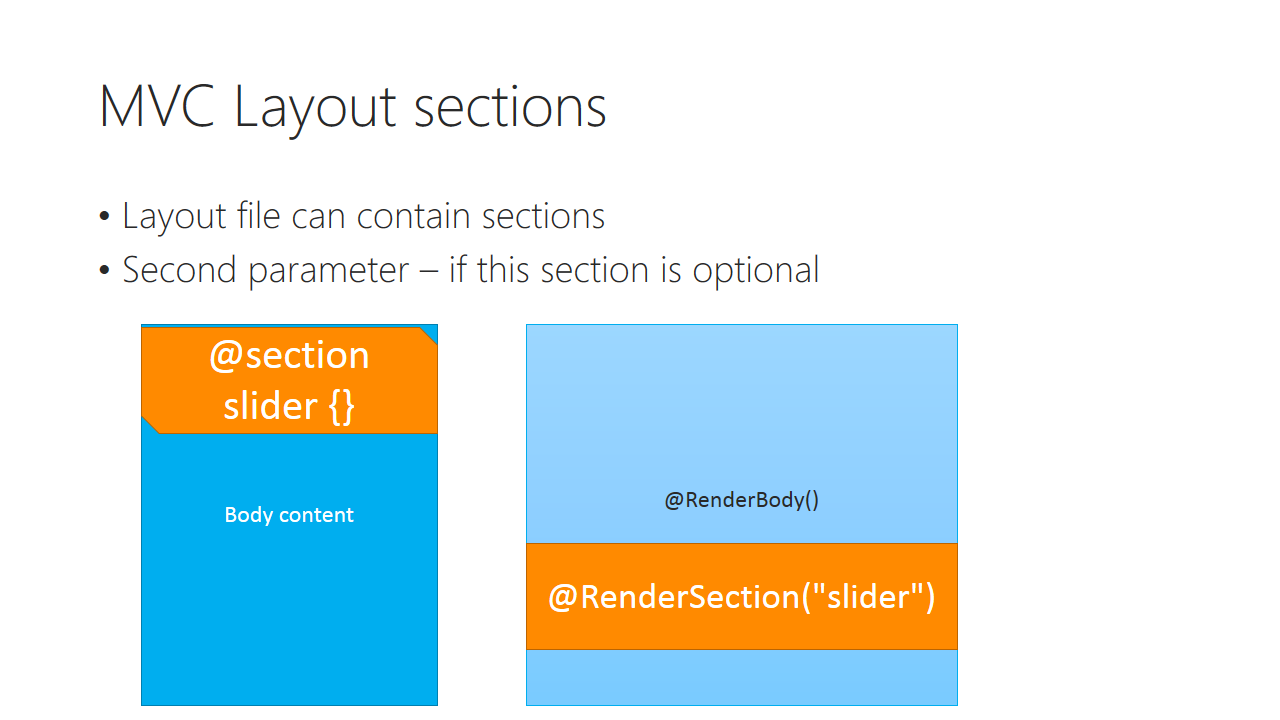
Trainer notes:

Student notes:



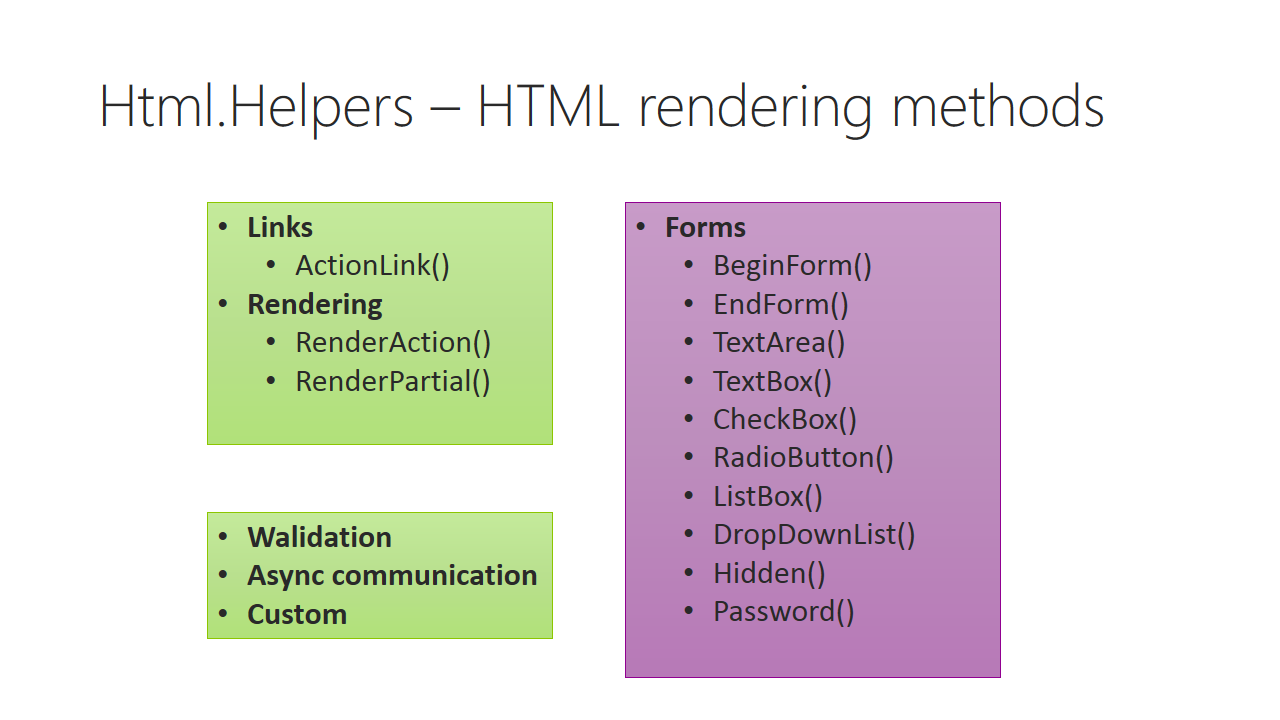
Trainer notes:

Student notes:



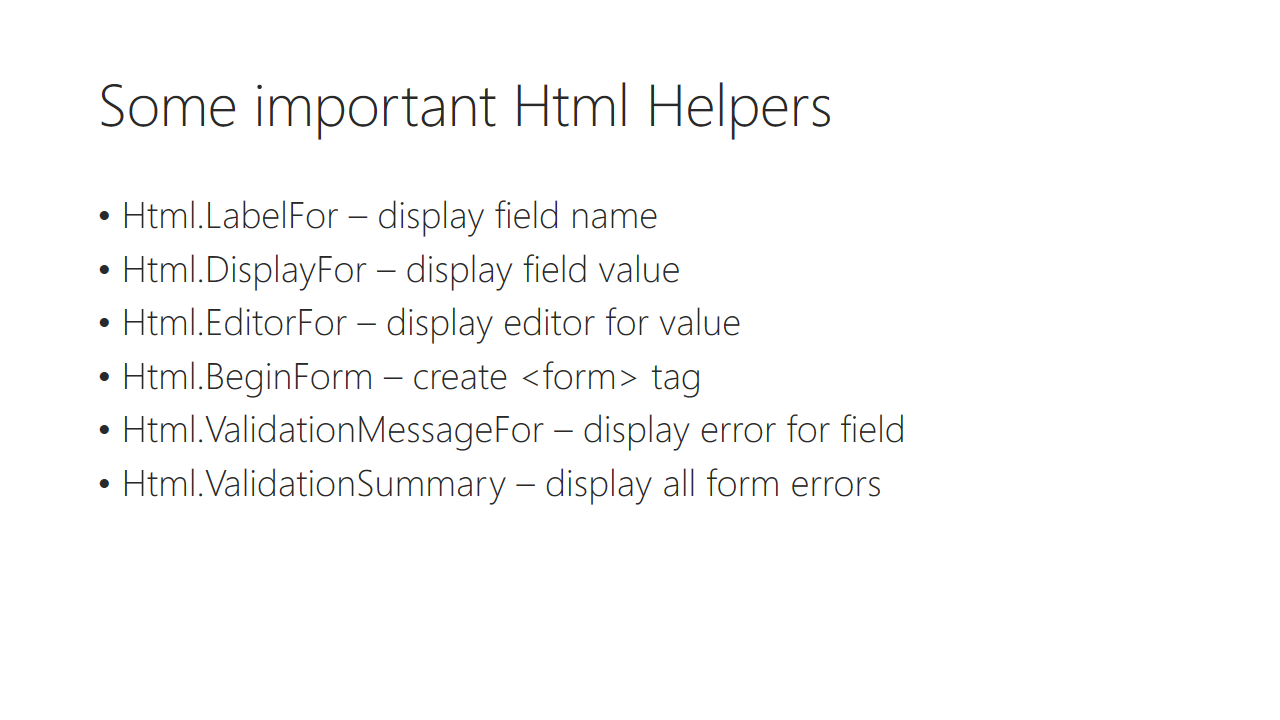
Trainer notes:

Student notes:



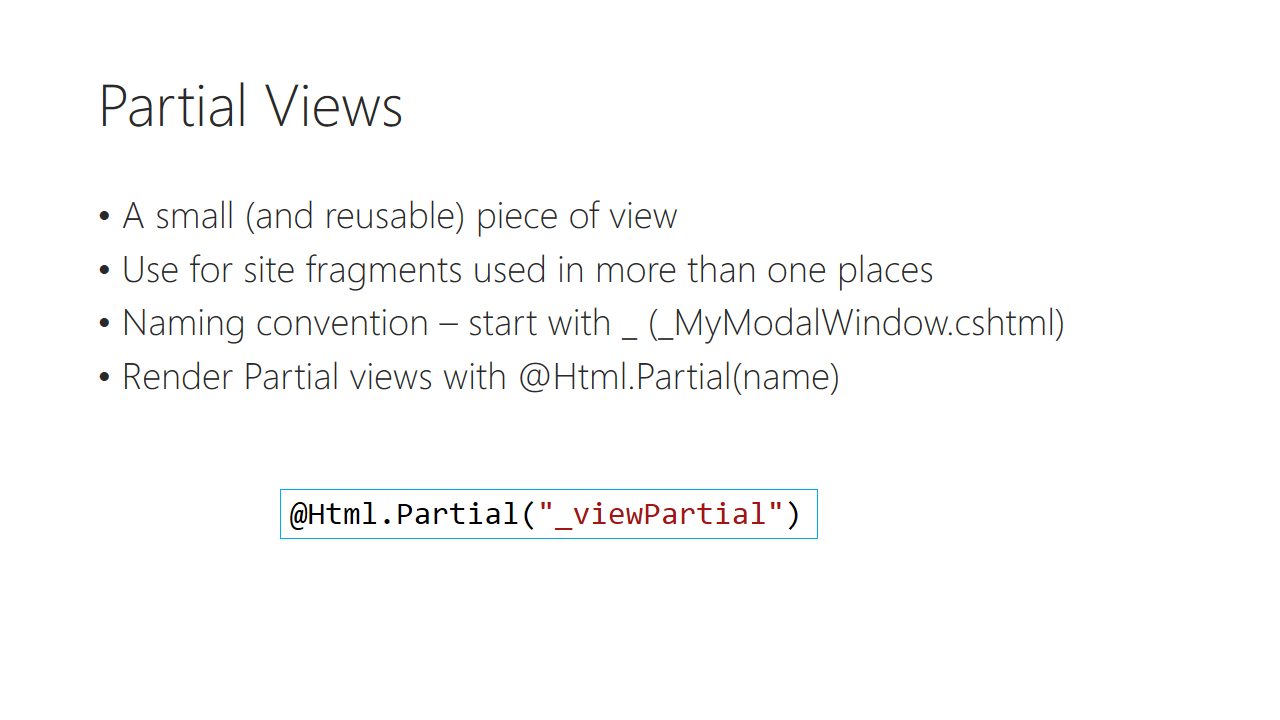
Trainer notes:

Student notes:



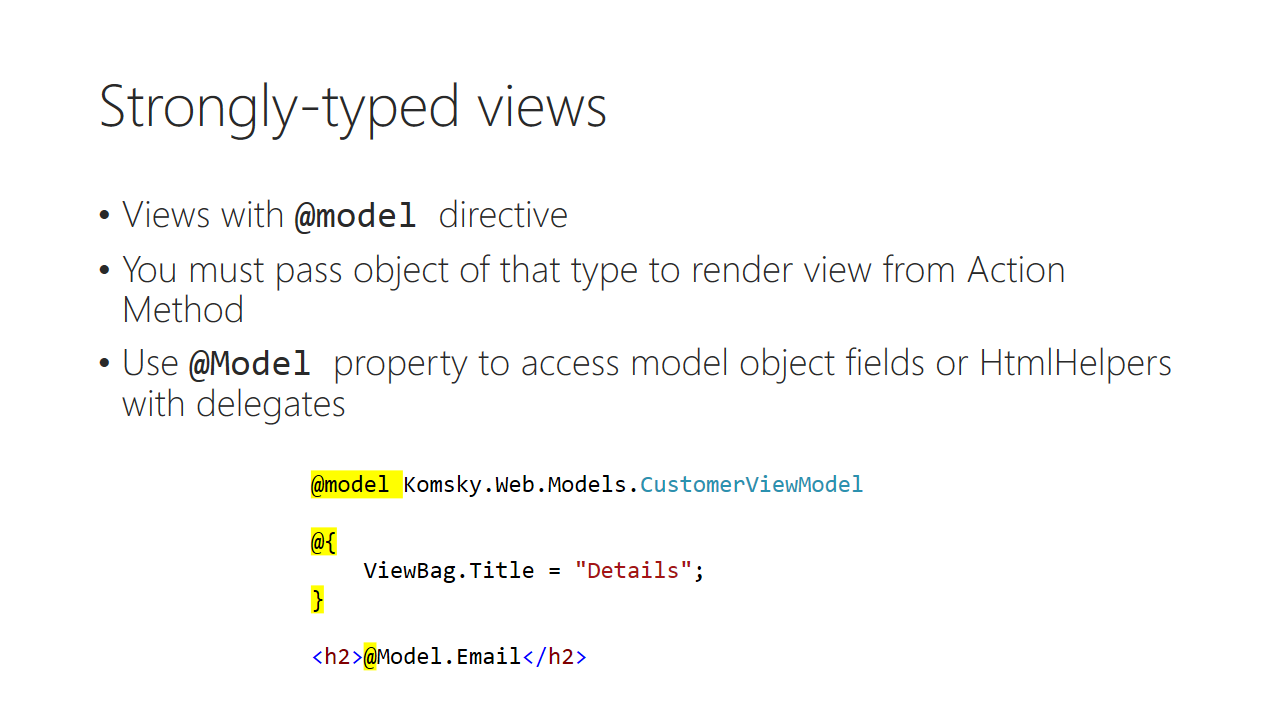
Trainer notes:

Student notes:



Trainer notes:

Student notes:



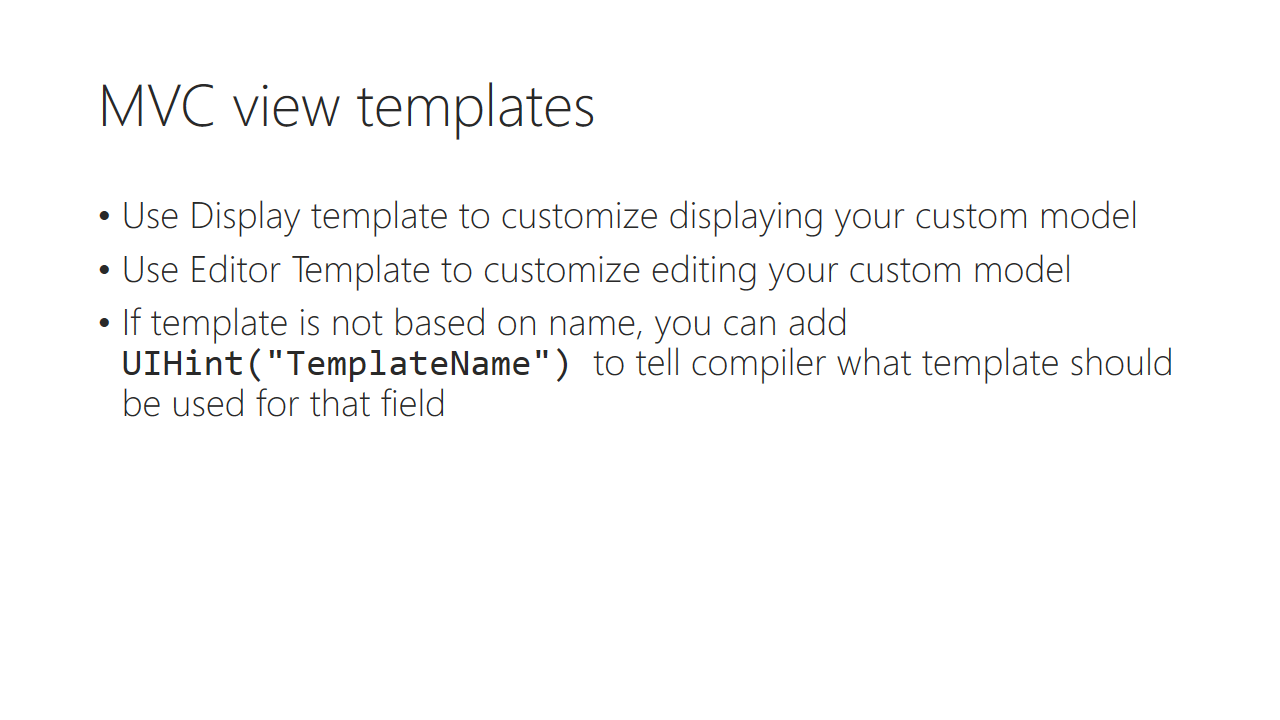
Trainer notes:

Student notes:

Demo: scaffolding views

Trainer notes:

Student notes:



Trainer notes:

Student notes:

# Lab 04

## Content

Out target for this lab is to learn how to use Views along with PartialViews and with HtmlHelpers a

## General steps

1. Working with T4MVC
2. Working with HtmlHelpers
3. Working with Partial Views
4. JavaScript bundling and minification

## Task 1 – Refactoring existing code to use T4MVC

1. Install T4MVC from nuget
2. Use *Build => Transform All T4 templates* Visual Studio command, to generate strongly-typed classes. After installing and generating those templates, you gain access to MVC static class with strongly typed names of controllers and actions in your solution.
3. Navigate to **\_Layout** file, and change navigation links. We need to refactor them using T4MVC, because using simple strings is prone to human errors – you have no compile time checking. The code should look like this:

@Html.ActionLink("Application name", MVC.Home.ActionNames.Index, MVC.Home.Name, new { area = "" }, new { @class = "navbar-brand" })

</div>

<div class="navbar-collapse collapse">

<ul class="nav navbar-nav">

<li>@Html.ActionLink(**MVC.Home.Name, MVC.Home.ActionNames.Index, MVC.Home.Name**)</li>

<li>@Html.ActionLink(**MVC.Ticket.Name, MVC.Ticket.ActionNames.Index, MVC.Ticket.Name**)</li>

<li>@Html.ActionLink(**MVC.Product.Name, MVC.Product.ActionNames.Index, MVC.Product.Name**)</li>

<li>@Html.ActionLink(**MVC.Customer.Name, MVC.Customer.ActionNames.Index, MVC.Customer.Name**)</li>

<li>@Html.ActionLink(**MVC.Home.ActionNames.About, MVC.Home.ActionNames.About, MVC.Home.Name**)</li>

<li>@Html.ActionLink(**MVC.Home.ActionNames.Contact, MVC.Home.ActionNames.Contact, MVC.Home.Name**)</li>

</ul>

@Html.Partial(MVC.Shared.Views.\_LoginPartial)

</div>

1. From now, everywhere in project, where a string of some kind is required, use MVC static object instead of string itself. It will not speed up your work, but it will make it safer.

## Task 2 – Working with Htmlhelpers and C# razor syntax

1. Open **Views/Ticket/Index.cshtml** and change **ViewBag.Title** on top of the page – assign a ‘Tickets’ value instead of Index
2. Change content of <h2> tag to ‘List of tickets’
3. Open **Views/Shared/\_Layout.cshtml** and add section called footer. Make it optional (parameter required: false). Place it above copyright tag:

**@RenderSection("footer", required: false)**

<p>&copy; @DateTime.Now.Year - My ASP.NET Application</p>

1. Open Web.config file and add two following lines in <appSettings> section:

<add key="Komsky:FullApplicationName" value="KOMSKY MVC Ticketing System"/>

<add key="Komsky:ShortApplicationName" value="KOMSKY"/>

1. Install *T4AppSettings* nuget. Use *–version 1.0.0* in case of compilation errors
2. Transform all T4 templates in Visual Studio Build command
3. In **\_Layout.cshtml** make following changes:
   1. In <title> tag replace with My ASP.NET Application with @AppSettings.Komsky\_ShortApplicationName
   2. In first Html.ActionLink replace application name with same @AppSettings.Komsky\_ShortApplicationName
   3. In footer copyright section tag replace name with @AppSettings.Komsky\_FullApplicationName
4. Navigate to http://www.w3schools.com/aspnet/mvc\_htmlhelpers.asp and spend some time analysing other Html Helpers available. We will need them later

## Task 3 – Working with partial views

In this task, we will refactor **Index.cshtml** for displaying tickets.

1. Go to **Views/Ticket** folder and create new empty view called \_Ticket. Tick 'Create as a partial view', so no layout page will be referenced here.
2. Copy content of Details view into Ticket. Take a notice on model directive on top of the page – it will require a TicketViewModel to be provided for rendering.
3. Remove C# razor code part with ViewBag.Title
4. Remove <h2> tag
5. Remove first <dt> tag with DisplayNameFor Title
6. Move content of first <dd> content into <h4> tag – replace TicketViewModel with model’s Title. Delete <dd> tag when it’s empty.
7. Change <p> tag on the end of the page with edit and index links.
   1. Wrap <p> tag with additional <div>
8. Go to **Index.cshtml** and delete <table> tag with all its content
9. Add new foreach loop for models elements, start by typing @
10. Call @Html.Partial with \_Ticket name. Remember to use MVC.Ticket… etc.
    1. T4MVC template must be regenerated to add \_Ticket definition, so you probably don’t see it now. Go to Build => Transform all T4 templates to update it.
    2. If you’re working in debugging session, you will need to rebuild your app, since transforming T4 templates creates new static entries into MVC class, and this must be compiled. Without that, IntelliSense will provide correct code, but you’ll end up with runtime error.
11. Finish writing Html.Partial call. Pass loop item element, as a model parameter into partial view.

@foreach (var item in Model)

{

@Html.Partial(MVC.Ticket.Views.\_Ticket, item)

}

1. Build and debug your app. Run it to see effects. Looks nice, isn’t it?
2. Let’s additionally align element’s horizontally, so they would use available space more effectively. Wrap your @Html.Partial call with <div style="display: inline-block">
3. Enjoy your new, nice ticket list!

Have you wondered what is the difference with Html.Partial and Html.RenderPartial? After Dr1Ku on StackOverflow:

Html.Partial returns a String, Html.RenderPartial calls Write internally, and returns void. While one can store the output of Html.Partial in a variable or return it from a method, one cannot do this with Html.RenderPartial. The result will be written to the Response stream during execution/evaluation. This also applies to Html.Action and Html.RenderAction.

## Task 4 – JavaScript bundling and minification

1. Go to **Komsky.Web/Scripts** folder and add two Javascript files called **global.js** and **init.js**
2. Inside **global.js** create new GLOBAL object and initialize it if it’s empty.
3. Add namespace into GLOBAL namespace object called Search.
4. Add new function called Search into Search namespace and put there console.log('Search') command to make sure the code run successfully.
5. Add new function called Init. Use console.log('Search Init') inside.
6. The GLOBAL object will be used later, but now on it provides a basic scaffold for implementing JavaScript client features.

var GLOBAL = GLOBAL || {};

GLOBAL.Search = GLOBAL.Search ||

{

Search: function() {

console.log('Search');

},

Init: function() {

console.log('Search init');

}

}

1. Go to **init.js** file and add following code:

$(document).ready(function() {

GLOBAL.Search.Init();

});

Code above provides single point of JavaScript initialization using jQuery’s ready function.

1. Navigate to BundleConfig class in **App\_Start** folder.
2. Duplicate one of the ScriptBundles, and provide your values for global.js and init.js files:

bundles.Add(new ScriptBundle("~/bundles/global").Include(

"~/Scripts/global.js",

"~/Scripts/init.js"));

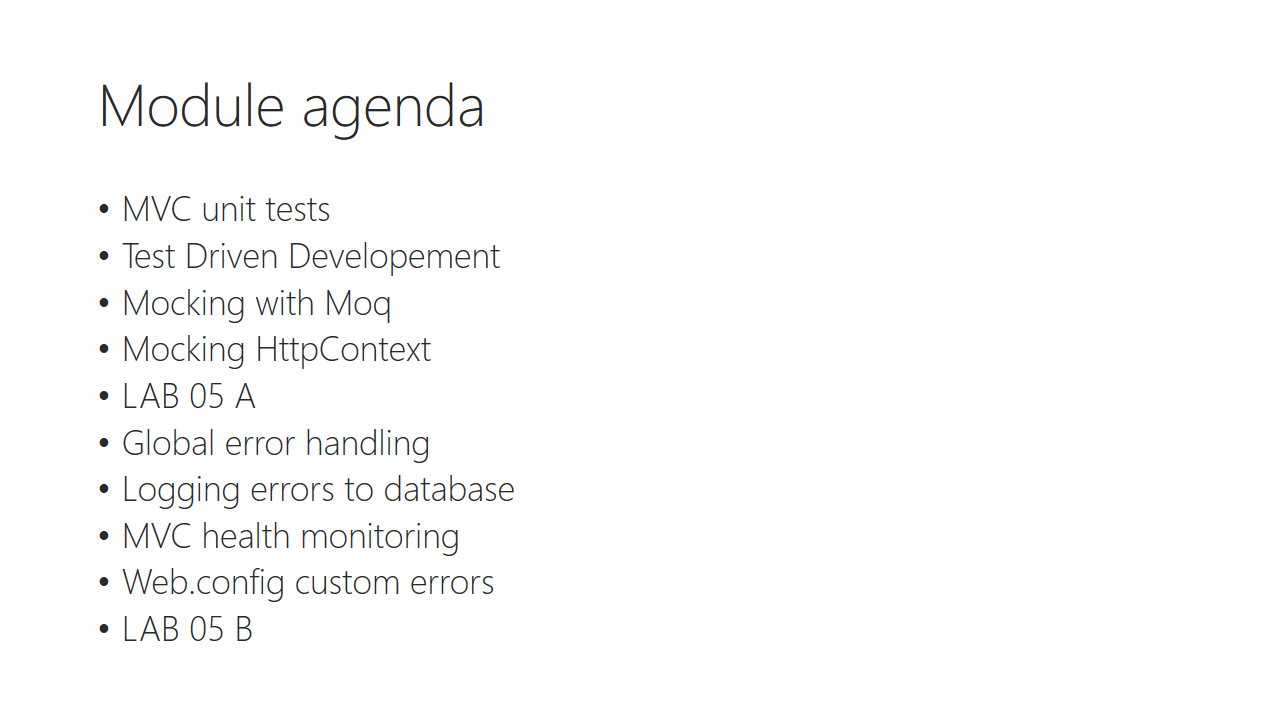
1. Navigate to **\_Layout.cshtml** file in **Views/Shared** folder.
2. Locate bundle script rendering command, duplicate it and fill with your global bundle. Make sure your bundle is rendered as the last one, or at least before jquery’s plugin. This is vital, as we use jQuery inside our code!
3. Build your project and start debugging.
4. Hit F12 on the newly opened browser tab, to show Developer tools and Esc key to display Console on the bottom. You should see *Search init* text there.
5. By adding additional line into your RegisterBundles method, we might also minify JavaScript files, that is compress it by removing comments, whitespaces and renaming long but meaningful variable names into shorter ones during release phase.
6. Add following code line as last in RegisterBundles method:

BundleTable.EnableOptimizations = true;

1. Run your solution and use browser’s developer tools to inspect javascript files.
2. Before finishing this lab, comment out optimizations, as we will need full JavaScript view for debugging purposes.

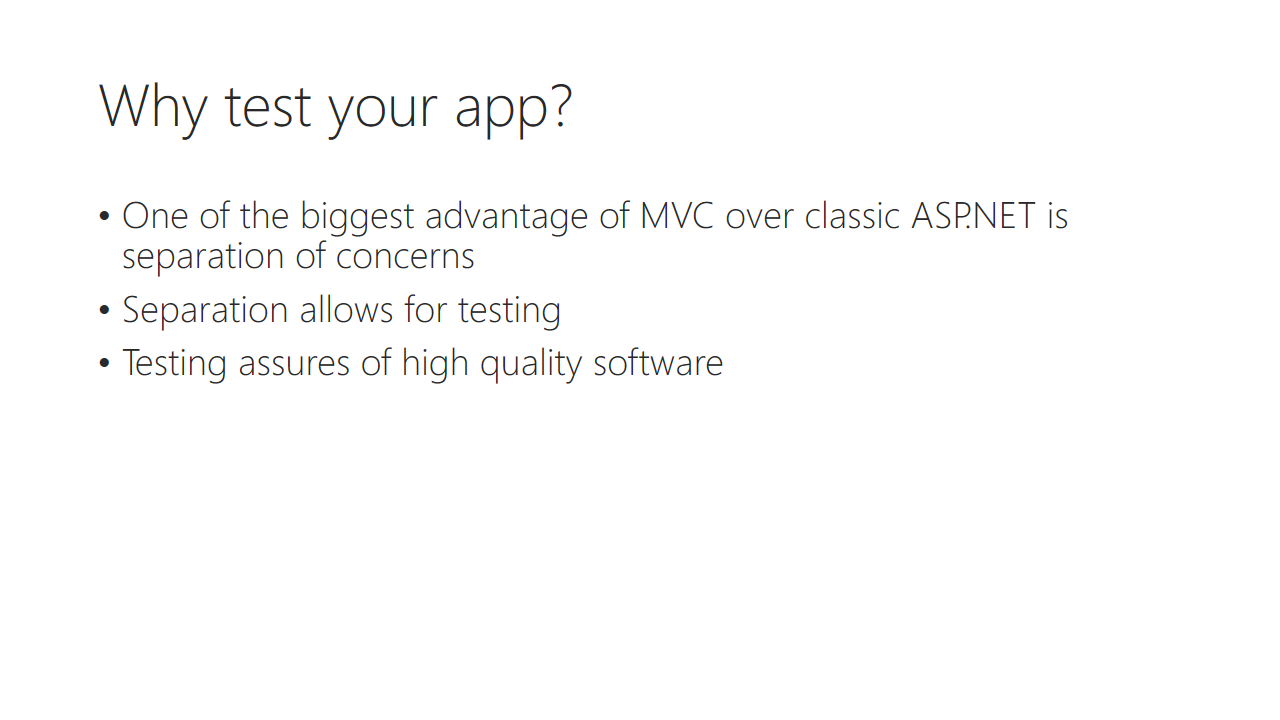
In this lab you have learned how to work with Views and razor syntax.

# Module 05 Testing and debugging



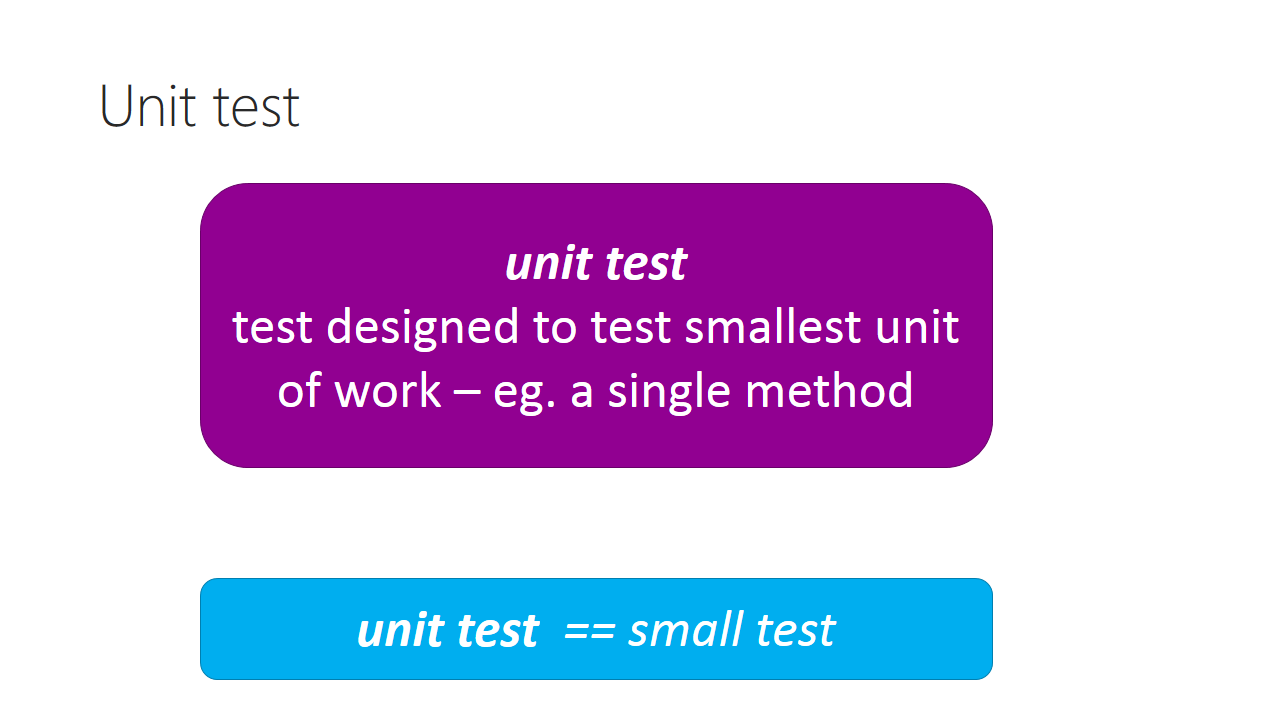
Trainer notes:

Student notes:



Trainer notes:

Student notes:



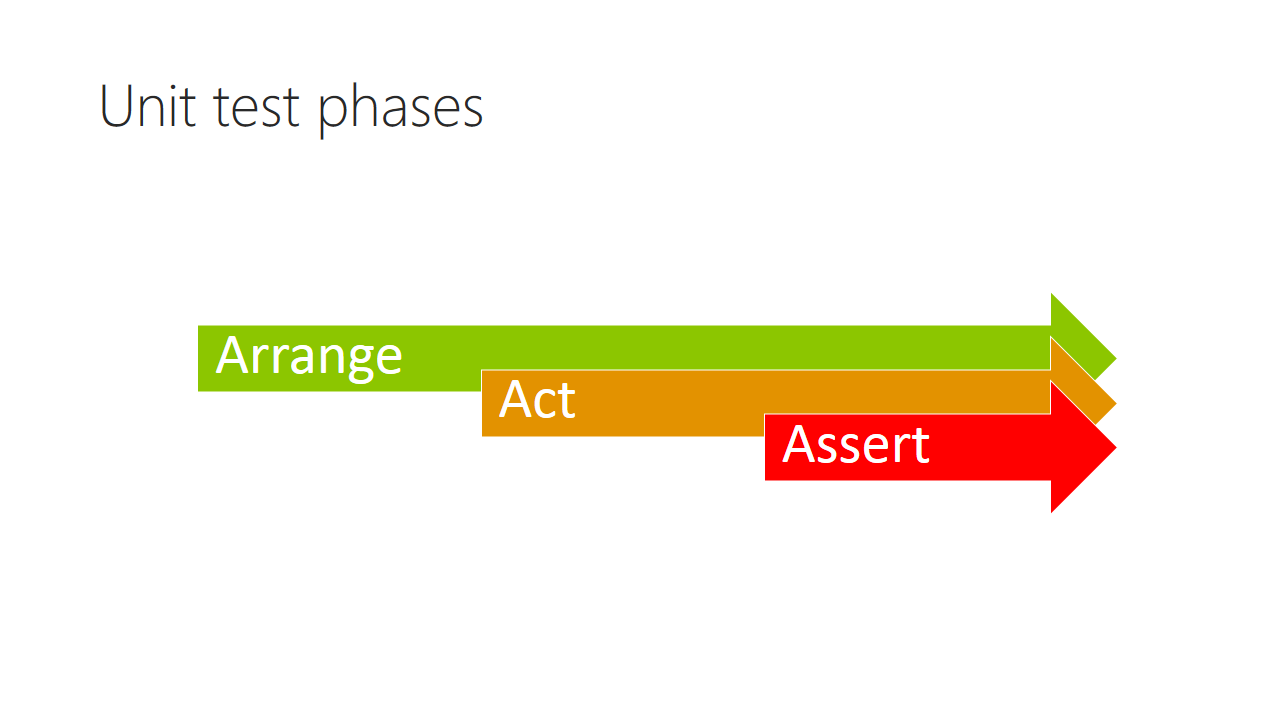
Trainer notes:

Student notes:



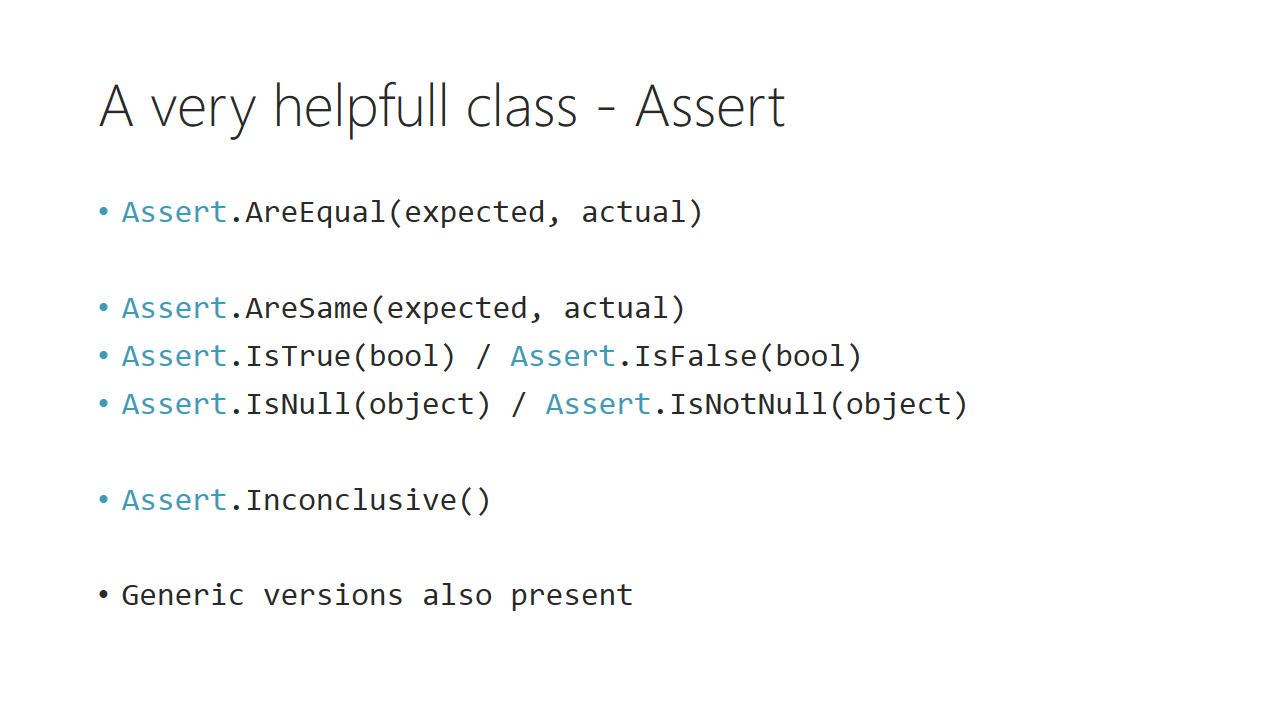
Trainer notes:

Student notes:



Trainer notes:

Student notes:



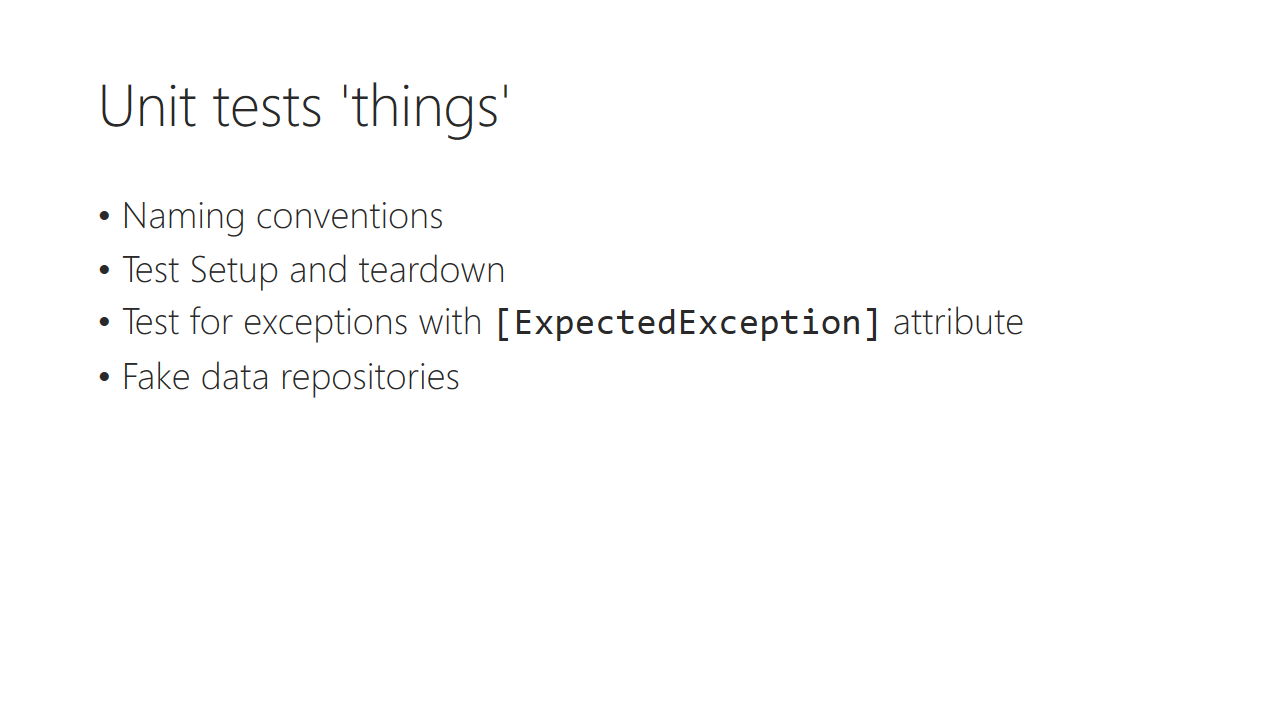
Trainer notes:

Student notes:

Demo: Unit test mvc template

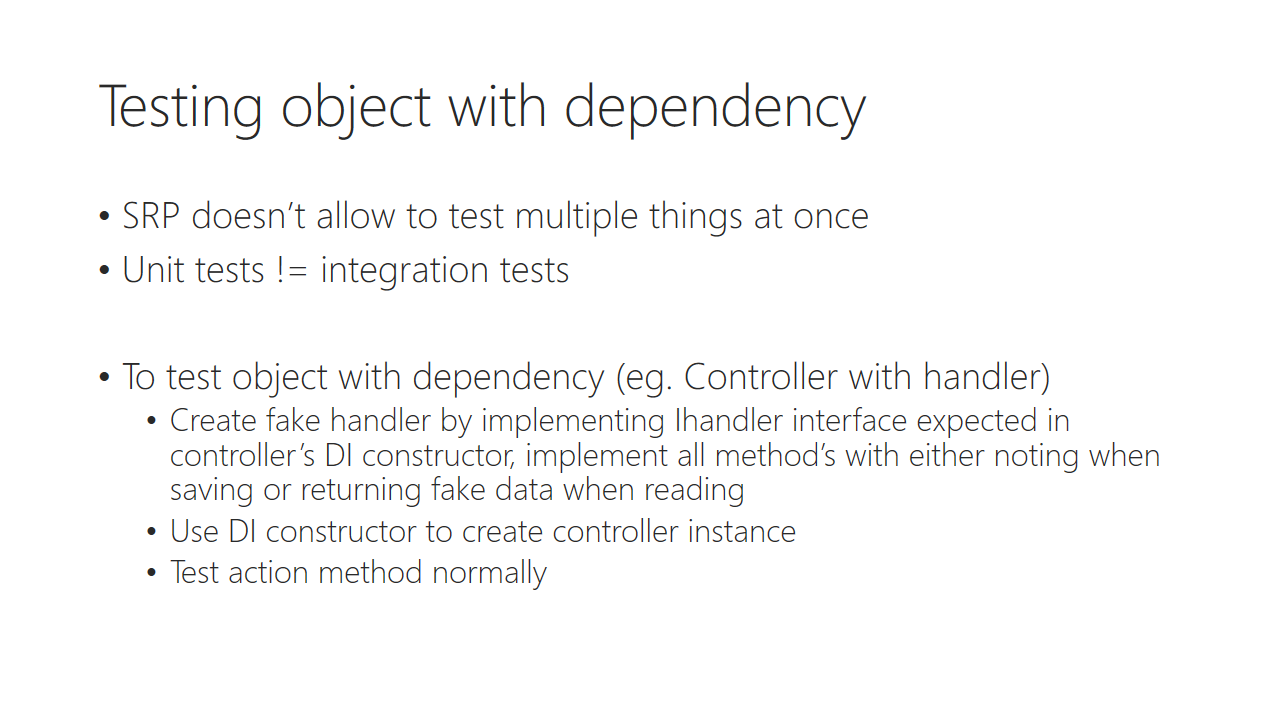
Trainer notes:

Student notes:



Trainer notes:

Student notes:



Trainer notes:

Student notes:

Demo: Testing objects with dependency

Trainer notes:

Student notes:



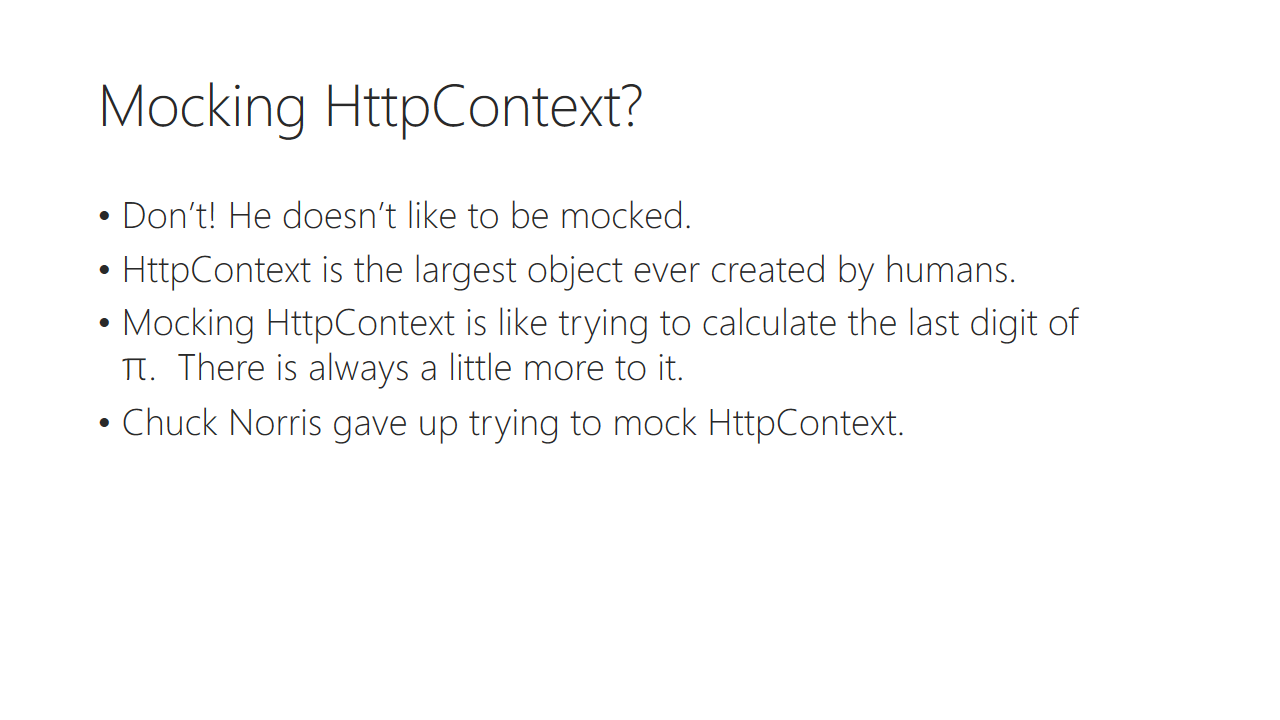
Trainer notes:

Student notes:

Demo: Mocking with Moq

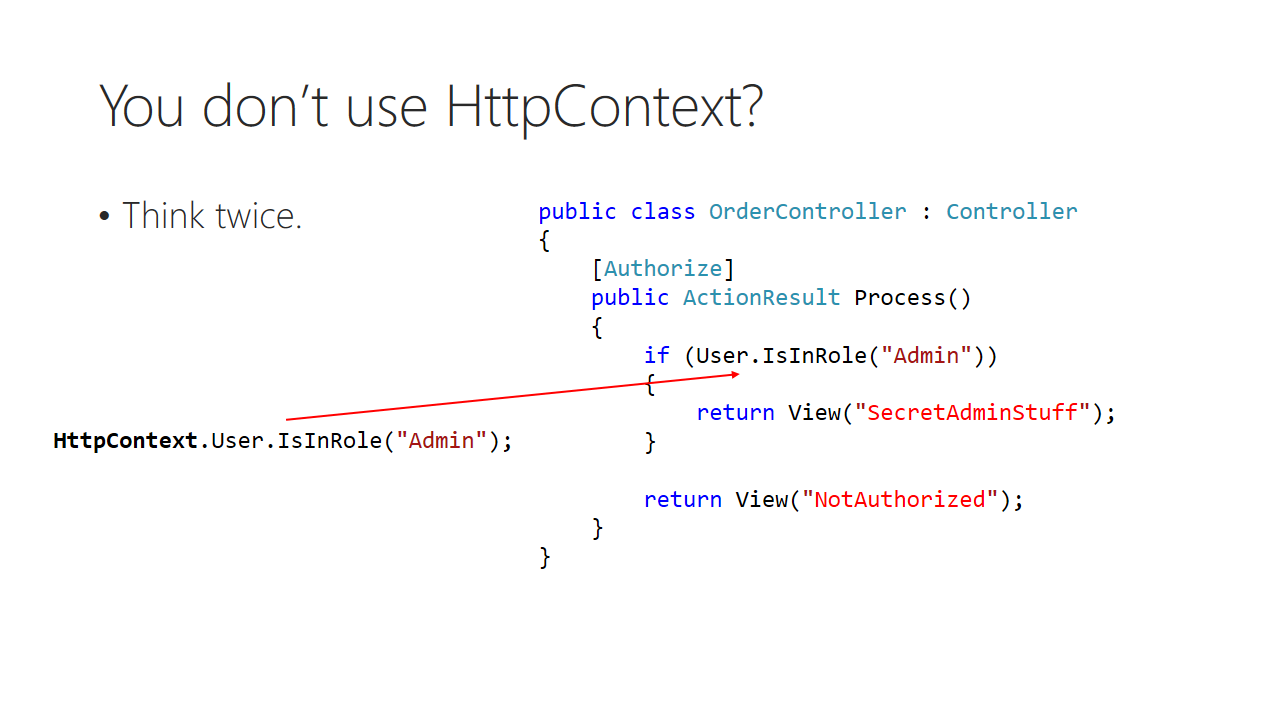
Trainer notes:

Student notes:



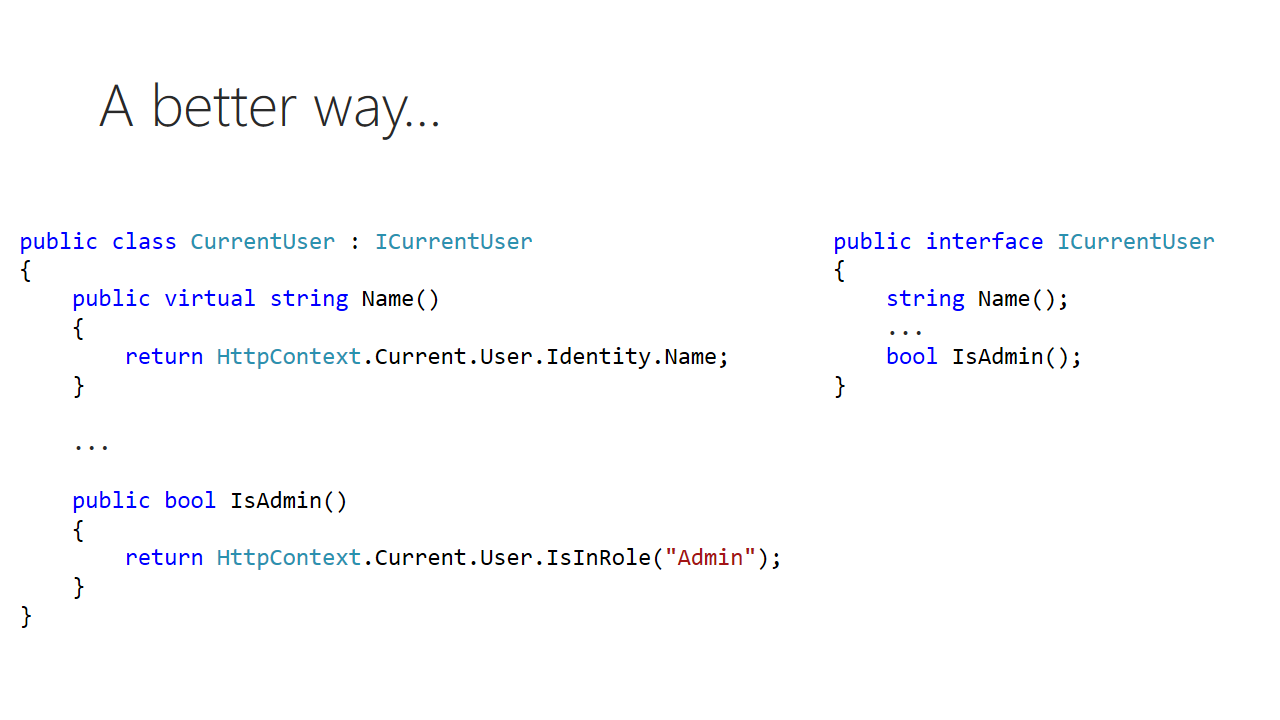
Trainer notes:

Student notes:



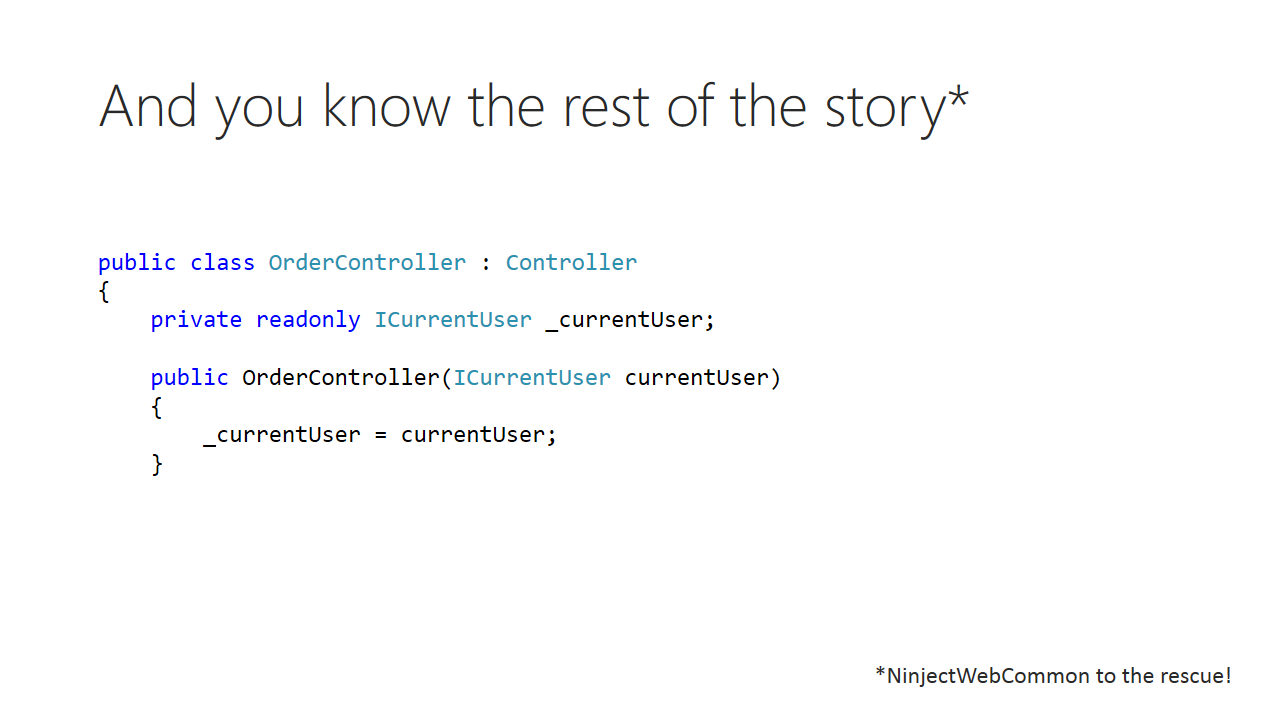
Trainer notes:

Student notes:



Trainer notes:

Student notes:



Trainer notes:

Student notes:

# Lab 05 A

## Content

In this lab you will learn how to test your controllers using unit testing. With Dependency Injection testing is easy, because – using Single Responsibility Principle – we can focus solely on particular method testing. To make things easier, we will manage our dependencies with Moq mocking framework.

## General steps

1. Unit testing home controller action methods
2. Unit testing with Moq framework
3. Refactoring test for code reuse
4. Mocking HttpContext object

## Task 1 – MVC Unit testing

1. Open **Komsky.Web.Tests** project. It has been added when you created your web application
2. Go to **Controllers/HomeControllerTest.cs** and analyse it’s content. Notice the three phases of tests – Arrange, Act, Assert
3. To run those tests, use Test => Run => All tests. A test explorer should appear, telling us that home controller test didn’t compile. We have removed default constructor, that is used in all of the tests.
4. Because the only constructor that is available is the DI one, we must provide some kind of object that implements IDataFacade interface. The only one existing in our project is DataFacade class itself. Let’s give a try and provide a new instance of it to HomeController constructor in each test method.

HomeController controller = new HomeController(new DataFacade());

1. Again, run all tests, using this key shortcut Ctrl + R, A . Tests passes fine. But is it a good idea to pass DataFacade object into HomeController? What if we test create method – do we want to actually insert new entity into database? Providing actual DataFacade object we brake most important rule of unit tests: ***test in isolation***
2. To fix this issue, let’s try implement IDataFacade interface in another object, and keep implementation methods empty. This would allow to test in isolation, right? A sample fake data façade object could look like this:

public class FakeDataFacade : IDataFacade

{

public void Dispose(){}

public void Commit(){}

public Task CommitAsync()

{

return new Task(() => { });

}

public ApplicationUserRepository ApplicationUsers { get; private set; }

public CustomerRepository Customers { get; private set; }

public ProductRepository Products { get; private set; }

public TicketRepository Tickets { get; private set; }

}

It quickly turns out that this is not that easy, as IDataFacade requires some real object. This gives us three options – to ignore them, to provide real instances of those, or to refactor IDataFace interface to use only interfaces without actual implementations. In real world the third option would have most sense, and because we have used pattern approach, this shouldn’t be this complex, right? Let’s fix that error of ours in a second!

1. Navigate to definition of IDataFacade, and change those four properties to expose interfaces instead of actual implementations (just add capital I on the beginning of each property type). The code should look like this:

public interface IDataFacade : IDisposable

{

void Commit();

Task CommitAsync();

IApplicationUserRepository ApplicationUsers { get; }

ICustomerRepository Customers { get; }

IProductRepository Products { get; }

ITicketRepository Tickets { get; }

}

1. Navigate to DataFacade class, and expose those interfaces instead of actual implementations. The same – add capital I on the type returned from property. Getter’s implementation should stay intact. You’re ready! That wasn’t that hard, was it? Now you see why sticking to patterns and interfaces is so important – refactoring your code is a glimpse of an eye!
2. Build and test your solution before continuing, to make sure the interfaces are doing their jobs (they really are!)
3. Head back to HomeControllerTest, as we need to finish our FakeDataFacade class. We will need four new classes to implement: FakeApplicationUserRepository, FakeCustomerRepository, FakeProductRepository and Fake TicketRepository.

*As implementing fake objects takes time and is very repeatable job, navigate to URL below to download full implementation of FakeDataFacade and other fake class in external project. Just download, extract and add this project to your solution under Test folder. Then, reference it in your Komsky.Web.Tests*

<https://github.com/komsky/training/blob/dev/Komsky/Komsky/Komsky.Fakes/Komsky.Fakes.zip?raw=true>

1. Take some time to review and analyse fake project. The implementation is really simple – wherever a object is required, some dummy data is created, and for void commands, fakes are doing nothing.
2. Head back to HomeControllerTest, and instead of DataFacade, provide FakeDataFacade object into DI constructor. Run tests and check if they have passed.

This is just simple showcase how you can unit test your application. This has also showed, that writing tests takes a lot of time – is there a way to focus more on testing rather that on writing fakes? Let’s see in second task – we will use Moq framework to speed things up.

## Task 2 – Unit testing using Moq mocking framework

1. Install *Moq* package from nuget for **Komsky.Web.Test** project
2. Add new class called TicketControllerTests to controllers folder. Decorate class definition with [TestClass] attribute (namespace Microsoft.VisualStudio.TestTools.UnitTesting).
3. Add new method called TicketIndexTest (parameterless, void return). Decorate it with [TestMethod]. Run all tests. Why your fresh TicketIndexTest passes flawlessly?
4. Add three lines of comments inside this mehod, to split awaiting job into three phases – arrange, act and assert
5. Just below arrange comment prepare instance of your TicketController class. What is required to run this constructor?
6. Because we don’t have FakeBaseHandler class, and we don’t have time to write it from scratch (can you imagine how long it took for me to write those DataFacade fakes?), we will use Moq to create those fakes on the fly.
7. Stop implementing this line, and create new, above current one.
8. Within the new line create new object of type Mock. This is a generic class, so you are required to provide a type. Provide IBaseHandlers<TicketDomain>. Code should look like this:

var ticketMock = new Mock<IBaseHandler<TicketDomain>>();

1. Repeat the line, but add productMock for product handler.
2. Go back to TicketController line, where constructor awaits for handlers interfaces. Provide ticketMock.Object and productsMock.Object parameters.
3. Seems, that this code is valid! Let’s write the rest of the test method to test it. Mimic HomeControllerTest class to implement act and assert part. The code should look like this:

[TestClass]

public class TicketControllerTests

{

[TestMethod]

public void TicketIndexTest()

{

// arrange

var ticketMock = new Mock<IBaseHandler<TicketDomain>>();

var productMock = new Mock<IBaseHandler<ProductDomain>>();

TicketController ticketController = new TicketController(ticketMock.Object, productMock.Object);

// act

ViewResult result = ticketController.Index() as ViewResult;

// asssert

Assert.IsNotNull(result);

}

}

1. Run your tests to see, if that code passes. Does it? Great! That was easy. You didn’t have to implement any of those stupid fakes. Let’s try with some more cumbersome method – Details.
2. Try implementing TicketDetails method by yourseft or ask your friend to help you. Just mimic the TicketIndexTest method
3. Unfortunatelly, this time it wasn’t that easy. Details method require parameter, and since you didn’t provide any I assume, you have called generated Details method from T4MVC template. That one doesn’t return ViewResult, so you’ve ended with exception and the test failed.
4. Provide some numeric value, and test Detail method again.
5. Again, we’ve ended with exception. This time code actually run, but mocking framework didn’t provide any value for model, so the factory have thrown null reference exception. But how can we fix it?
6. Put a breakpoint just above Details(1) call and start debugging your test.
7. When breakpoint hits, step into definition and try to find what method is returning null, but actually is expected to return data. It’s \_ticketHandler.GetById(id)!
8. We need to made out Mock return some data specifically for this one. Stop debugging and go back to TicketDetailsTest method.
9. Just after creating mocks, but before providing it to TicketController constructor add a line like below:

ticketMock.Setup(x => x.GetById(It.IsAny<Int32>()))

.Returns(Fakes.FakeTicketRepository.GetFakeTicket().CreateTicketDomain());

1. Run the test again. Does it still fail? No! We have successfully provided object expected in Details method without writing fakes!

*You might think now: “Hey, buddy, that’s cheating! I see what you did in this line, and you are actually using FakeTicketRepository. You call that ‘not writing fakes?”*

*Well, yes. If you take a closer look, you’ll see that I did in fact used existing FakceTicketReposotory, but only as a easy source of TicketDomain object. You could easily just put there fake TicketDomain object yourself.*

1. Using TicketDetailsTest as implementation source, mimic this behavior in TicketEditTest, where you test Edit(model) method. What has changed?

In this task you have learned basics of Moq mocking framework. Moq will create interface instance with fake method calls on the fly, and when you need to provide a definition for actual method, you use Mock.Setup method. Simple as that!

## Task 3 – Refactoring unit tests for code reuse

Take a general look at our test methods - it seems, that we have ended with code duplication, especially with ticketMock and productMock objects. We will refactor them into separate method called TestSetup. The advantage of test setup it that it’s being automatically called before every test, not needing to manually call it.

1. Extract ticketMock and productMock objects from TicketEditTest method into general, private fields. Use Refactor => Introduce field option.
2. Refactor ticketController the same way as previous mocks objects.
3. Create new void and parameterless method called TestSetup. Decorate it with [TestInitialize] attribute.
4. Move \_ticketMock and \_productMock field initialization into this method as well as every Setup() call on those objects. In fact, move the whole arrange comment content from TicketEditTest, so that only act and assets phases remains.
5. Refactor the rest of the methods in TicketControlerTests to use \_ticketController field.
6. Add new void and parameterless method called TestTeardown. Decorate it with [TestCleanup] attribute.
7. Inside this method set every field available in this class to null.
8. The whole refactored class could looks like this:

[TestClass]

public class TicketControllerTests

{

private Mock<IBaseHandler<TicketDomain>> \_ticketMock;

private Mock<IBaseHandler<ProductDomain>> \_productMock;

private TicketController \_ticketController;

[TestInitialize]

public void TestSetup()

{

\_ticketMock = new Mock<IBaseHandler<TicketDomain>>();

\_productMock = new Mock<IBaseHandler<ProductDomain>>();

\_ticketMock.Setup(x => x.GetById(It.IsAny<Int32>()))

.Returns(Fakes.FakeTicketRepository.GetFakeTicket().CreateTicketDomain());

\_ticketController = new TicketController(\_ticketMock.Object, \_productMock.Object);

}

[TestCleanup]

public void TestTeardown()

{

\_ticketMock = null;

\_productMock = null;

\_ticketController = null;

}

[TestMethod]

public void TicketIndexTest()

{

// act

ViewResult result = \_ticketController.Index() as ViewResult;

// asssert

Assert.IsNotNull(result);

}

[TestMethod]

public void TicketDetailsTest()

{

// act

ViewResult result = \_ticketController.Details(5) as ViewResult;

// asssert

Assert.IsNotNull(result);

}

[TestMethod]

public void TicketEditTest()

{

// act

var result = \_ticketController.Edit(new TicketViewModel());

// asssert

Assert.IsNotNull(result);

}

}

## Task 4 – Mocking Httpcontext

1. Let’s write one last test – the create method from TicketController. Go, and create TicketCreateTest.
2. As you have refactored repositories mocks, you only have to implement act and assert method. Call Create(model) method and check for any general errors. Does this test pass?
3. Debug and check what method causes our test to fail.
4. It seems that it’s User.Identity.GetUserId(). Actually this User object comes from HttpContext class and Current property. We could try to mock it, but as explained during lectures, it’s not very good idea. Let’s refactor TicketController class, and wrap HttpContext!
5. Go to Common folder and add **Komsky.Mvc** project.
   1. Important – don’t be tempted to name it with capital MVC, as it will cause T4MVC clash.
6. Create new ICurrentUser interface with following:
   1. String returning method GetUserId()
   2. String returning property Name
7. Implement this in class called CurrentUser. As methods body, return respectively HttpContext.Current.User.Identity.Name;and HttpContext.Current.User.Identity.GetUserId()
8. For the HttpContext class to be available, you’ll need System.Web reference.
9. For the GetUserId extension method to appear, you’ll need Identity OWIN package. Install it into MVC project with this command:

Install-Package Microsoft.AspNet.Identity.Owin

1. Build the solution and debug any errors.
2. Go to Komsky.Web and reference Komsky.Mvc project.
3. Introduct ICurrentUser field called \_currentUser, and create it the DI way in TicketController.
4. Bind ICurrentUser interface to CurrentUser class with Ninject
5. Replace all User calls with \_currentUser, especially this in Create method call in TicketController. You can easily find out where you have referenced User property so far, by deleting Microsoft.AspNet.Identity using directive and trying to build your project. Your code should look like this:

public virtual ActionResult Create(TicketViewModel model)

{

if (ModelState.IsValid)

{

**model.OwnerId = \_currentUser.GetUserId();**

\_ticketHandler.Add(model.CreateTicketDomain());

\_ticketHandler.Commit();

return RedirectToAction("Index");

}

return View(model);

}

1. Final step is to update out TicketControllerTest. Navigate to TestSetup method in TicketControllerTests.
2. Create new mock of ICurrentUser, setup GetUserId method and pass object to controller.

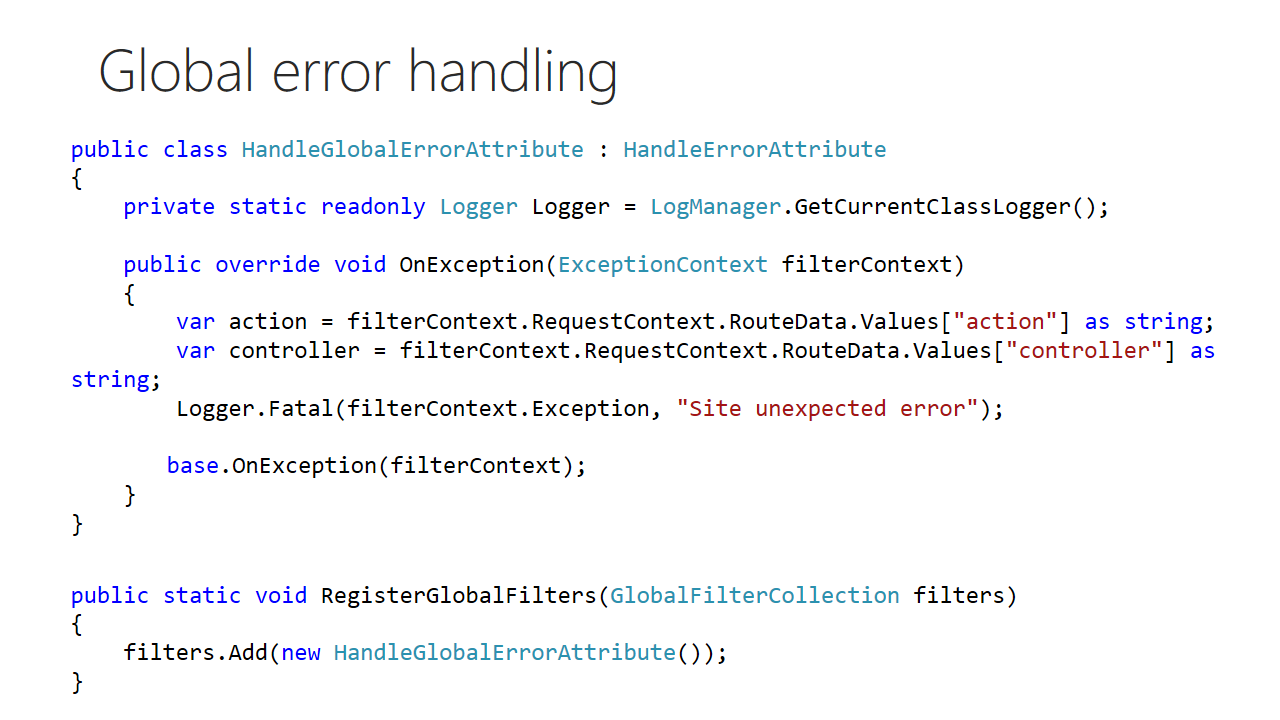
var currentUserMock = new Mock<ICurrentUser>();

currentUserMock.Setup(x => x.GetUserId()).Returns("FakeUserId");

\_ticketController = new TicketController(\_ticketMock.Object, \_productMock.Object, currentUserMock.Object);

1. Build, debug and test the solution, creating new ticket. Does it work?
2. Run all your tests. Does TicketCreateTest passes?

At this point we have thoroughly tested our controllers in isolation using Moq framework. Remember to test each method in controller as soon as possible.



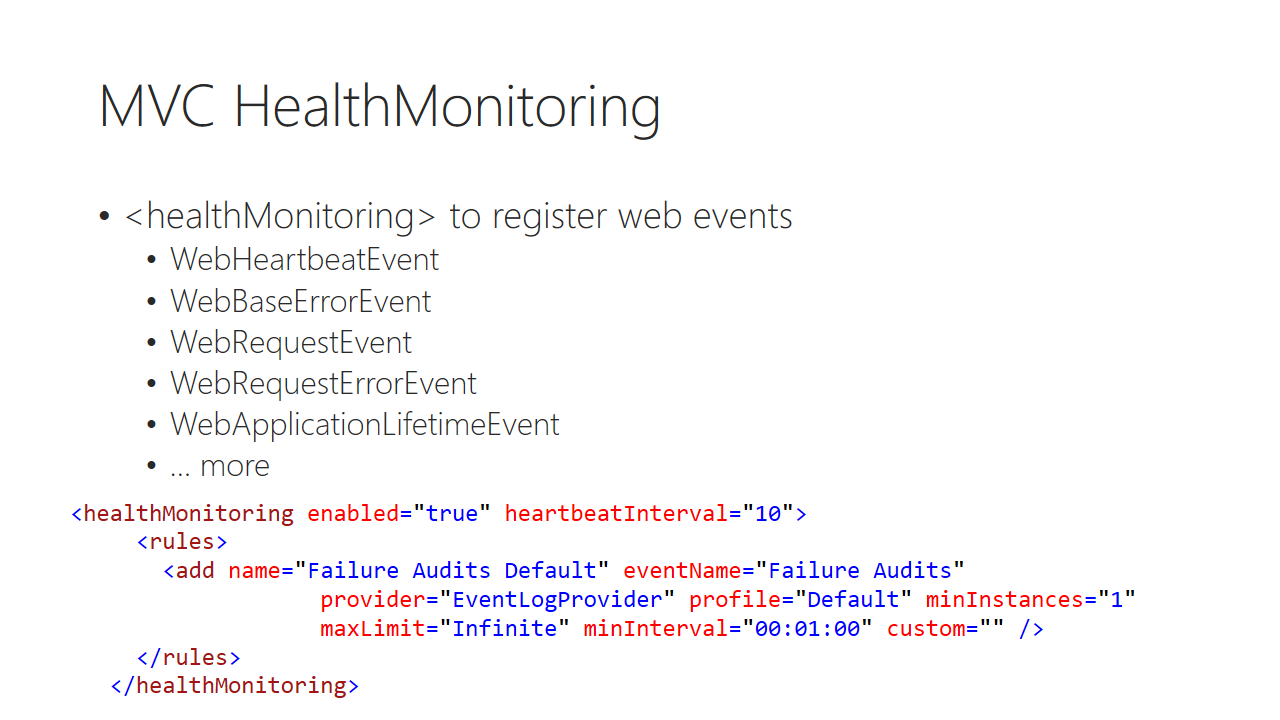
Trainer notes:

Student notes:



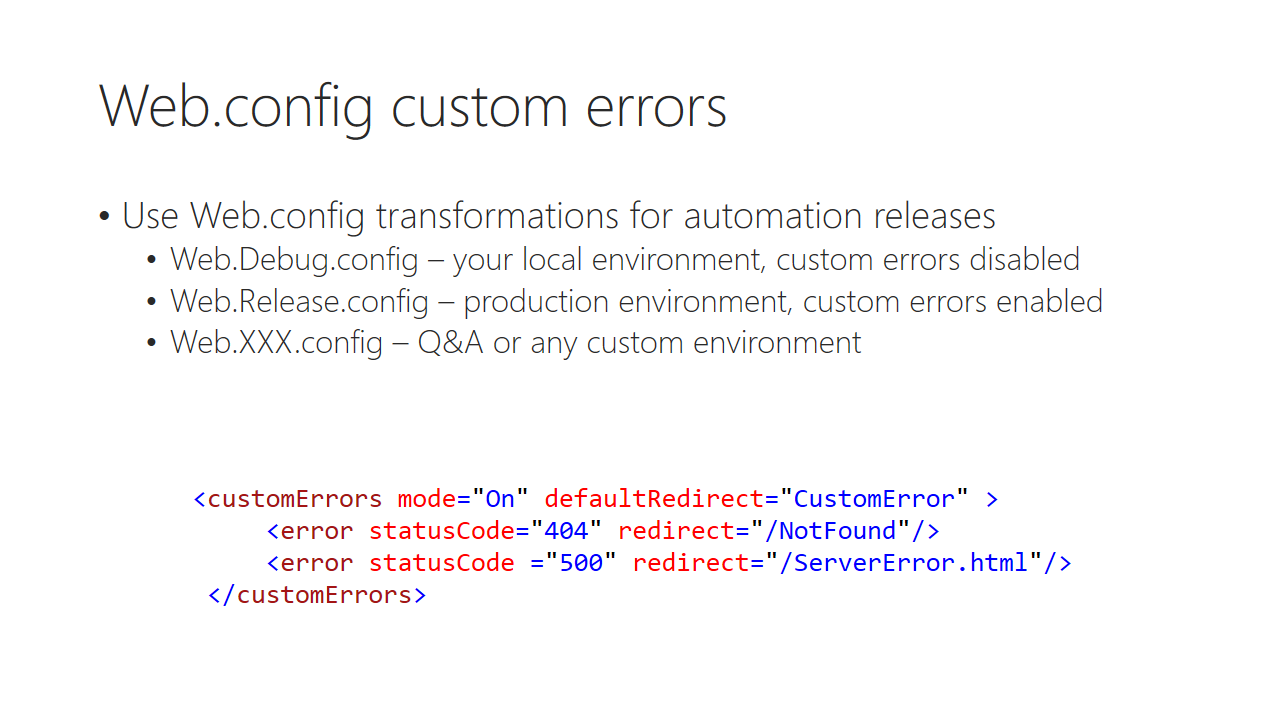
Trainer notes:

Student notes:



Trainer notes:

Student notes:



Trainer notes:

Student notes:

# Lab 05 B

## Content

In this lab you focus on handling global errors in the user friendly way, but still saving all the necessary data to debug our application.

## General steps

1. Turning on custom errors
2. Handling unhandled exceptions globally
3. Saving errors into database

## Task 1 – Turning on custom errors

1. First, we need to simulate error. Navigate to Home controller and create new ActionMethod called Error. This method should return Index view, but before it does, throw new ApplicationException error without handling it (without try/catch). Code should look like this:

public virtual ActionResult Error()

{

throw new ApplicationException("Simulating fatal and unexpected error");

return View(MVC.Home.Views.Index);

}

1. Build your application and run without debugging. In URL navigate to /Home/Error action. What do you see? We call this ‘yellow screen of death’. This provides some useful info for developer, eg. Full stack trace, but we don’t want to provide this detailed and ugly message to actual client.
2. Close browser tab and go back to Visual Studio. Open **Web.config** file from **Komsky.Web** project.
3. Add following lines to <system.web> section:

<customErrors mode="On" defaultRedirect="~/Views/Shared/Error.cshtml">

<error statusCode="404" redirect="~/Views/Shared/404.cshtml"/>

</customErrors>

1. Go to shared folder and copy-paste on Error.cshtml file. Name newly created file 404.cshtml and customize it to your needs.
2. Build and run your application. Navigate to Home/Error page again. Now, you should see nice message without error details.
3. Remember to use custom errors entry on production server, but for now, disable it by setting mode attribute to off.

## Task 2 – Handling errors globally

1. Go to **Komsky.Mvc** project and create new class called HandleGlobalErrorAttribute, inheriting from HandleErrorAttribute class
   1. Reference System.Web.Mvc
2. Within this class override OnException method.
3. Inside, use following code to gain some info about where the exception was thrown:

var action = filterContext.RequestContext.RouteData.Values["action"] as string;

var controller = filterContext.RequestContext.RouteData.Values["controller"] as string;

1. You might want to save For now just leave it as is. We will get back here in next task, saving the details into browsable database table.
2. Go back to **Komsky.Web** project and open **FilterConfig.cs** file in **App\_Start** folder.
3. Change HandleErrorAttribute into HandleGlobalError attribute.
4. Test correct behaviour by navigating into definition of this class and setting breakpoint in OnException.
5. Run your app and navigate to Home/Error URL. Does our breakpoint hit? At least, it should ☺

## Task 3 – Saving error details into database

1. Install NLog logging framework from nuget to Komsky.Web project and to Komsky.Mvc project.
2. Add new xml file called **NLog.config** into **Komsky.Web** root folder.
3. Make sure that the properties of this file are set this way:
   1. Build action: Content
   2. Copy to Output Directory: Copy always
4. To make things simple, I provide here full content of this file. You can download it from git, it’s available here:

https://github.com/komsky/training/blob/dev/Komsky/Komsky/Komsky.Web/Nlog.config

<nlog xmlns="http://www.nlog-project.org/schemas/NLog.xsd" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" autoReload="true">

<targets>

<target type="Database" name="WebDatabase" connectionStringName="DefaultConnection">

<commandText>

insert into [Security].[SystemLog](DateTime,Level,Logger,Message,Exception) values(@time\_stamp, @level, @logger, @message,@exception);

</commandText>

<parameter name="@time\_stamp" layout="${date}"/>

<parameter name="@level" layout="${level}"/>

<parameter name="@logger" layout="${logger}"/>

<parameter name="@message" layout="${message}"/>

<parameter name="@exception" layout="${exception:format=tostring}"/>

</target>

</targets>

<rules>

<logger name="Komsky.\*" minlevel="Debug" appendTo="WebDatabase"/>

</rules>

</nlog>

1. To make use of this code, we need to create *SystemLog*. Open Data.Entities project and add following class:
   1. Name: SystemLog
   2. Properties:
      1. Id : int with [Key] attribute
      2. DateTime: DateTime
      3. Level : String
      4. Logger : String
      5. Message : String
      6. Exception : String
   3. Decorate class definition with [Table] attribute. Provide table name as: Security.SystemLog
2. Add this into DbContext DbSet<SystemLog> SystemLogs property.
3. Add migration and update database.
4. Now we can start logging data into database. Go to HandleGlobalErrorAttribute class.
5. Introduce private static Logger field with the code below into this class:

private static readonly Logger Logger = LogManager.GetCurrentClassLogger();

1. Add a line inside OnException method as follows:

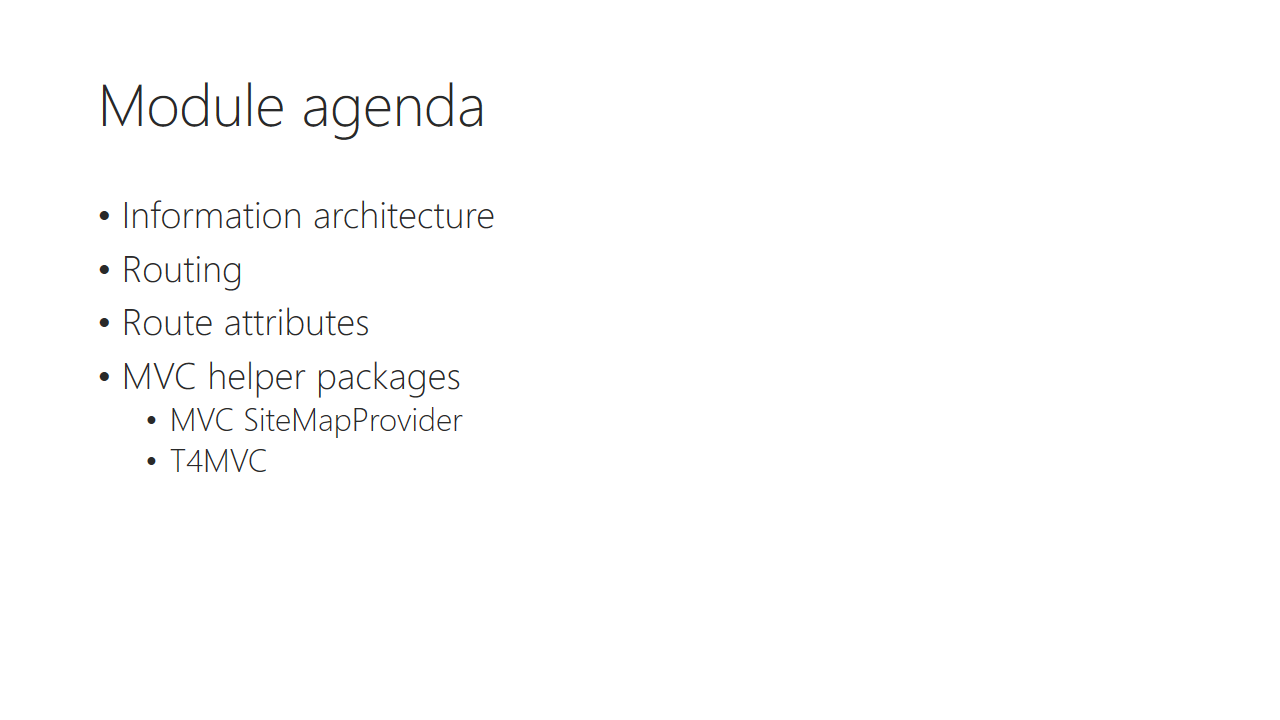
Logger.Fatal(filterContext.Exception, "Site unexpected error");

1. Build and debug your project. Then run it and navigate to Home/Error URL.
2. Click continue when Visual Studio notifies about your exception.
3. When you see yellow screen of death, stop debugging, open Server Explorer window and check content of *Security.SystemLog* table. It should contain your exception details.

*If you have ended with empty SecurityLog table, you might want to debug your solution. Navigate to this URL to get troubleshooting process details: https://github.com/NLog/NLog/wiki/Logging-Troubleshooting*

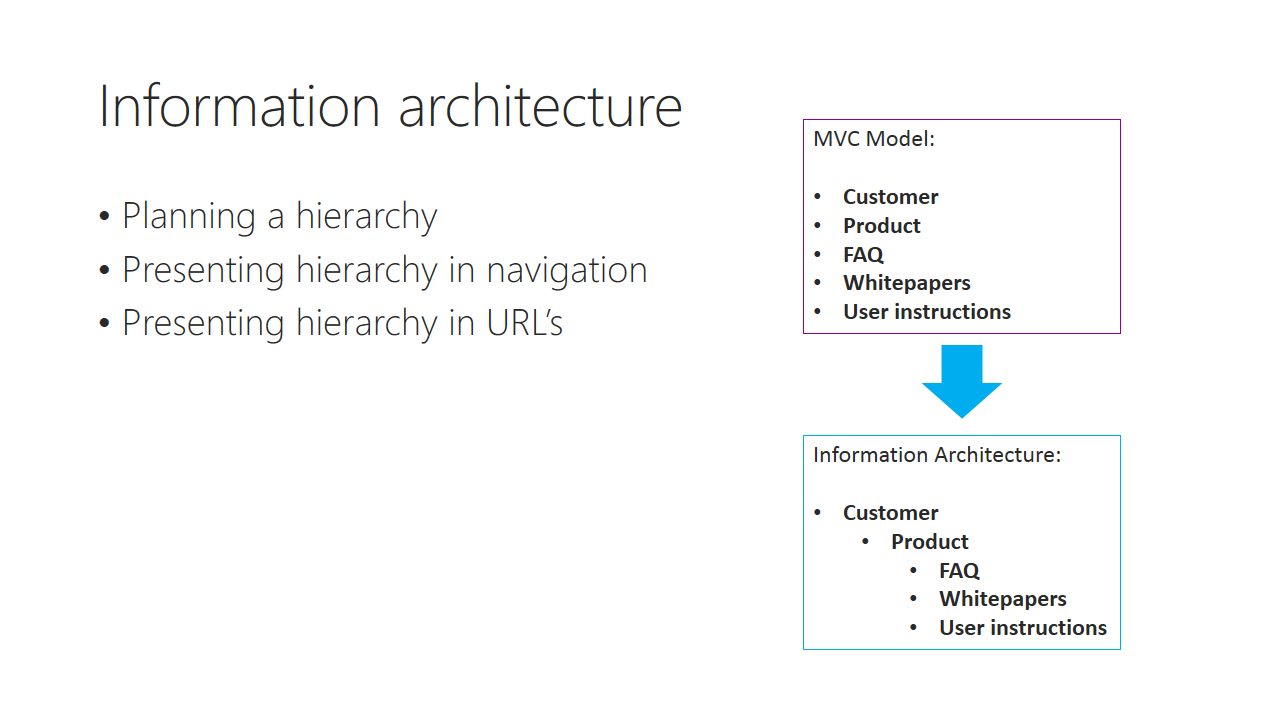
In this lab you have learned how to unit test your application, using Moq for mocking dependencies and how to save fatal errors details into database.

# Module 06 Routing



Trainer notes:

Student notes:



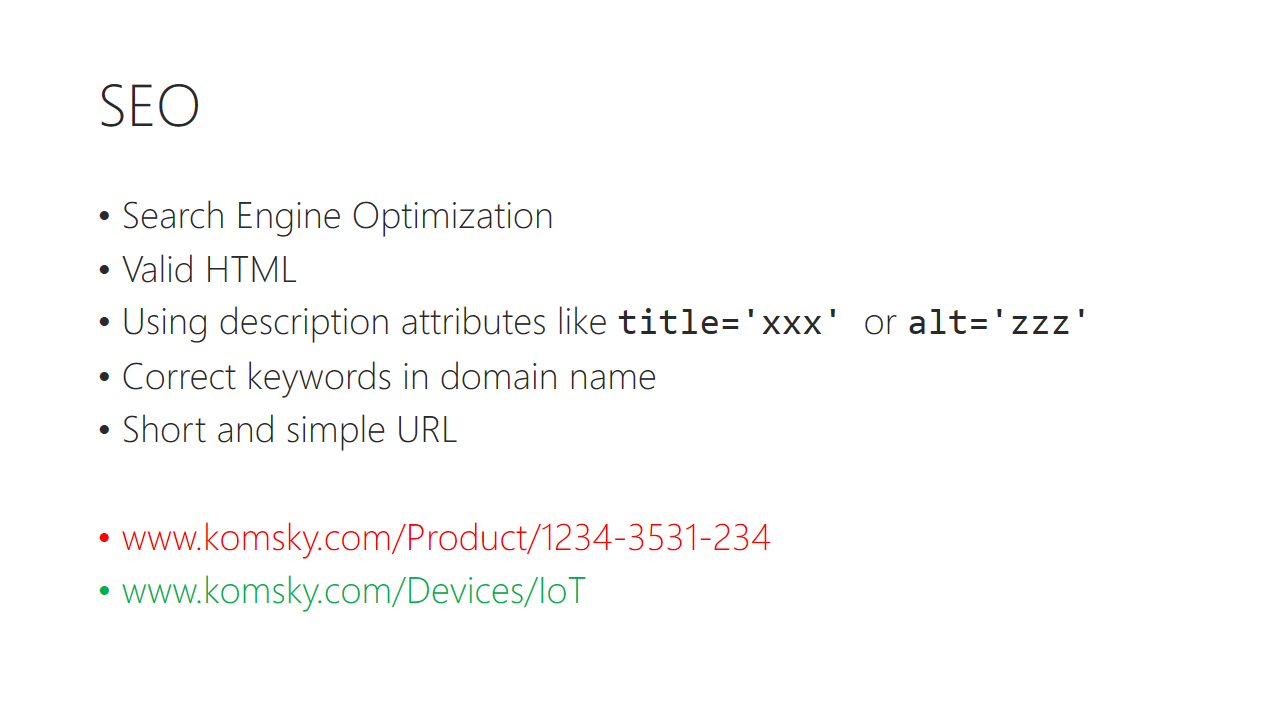
Trainer notes:

Student notes:

Discussion: Design an information architecture for photo sharing application

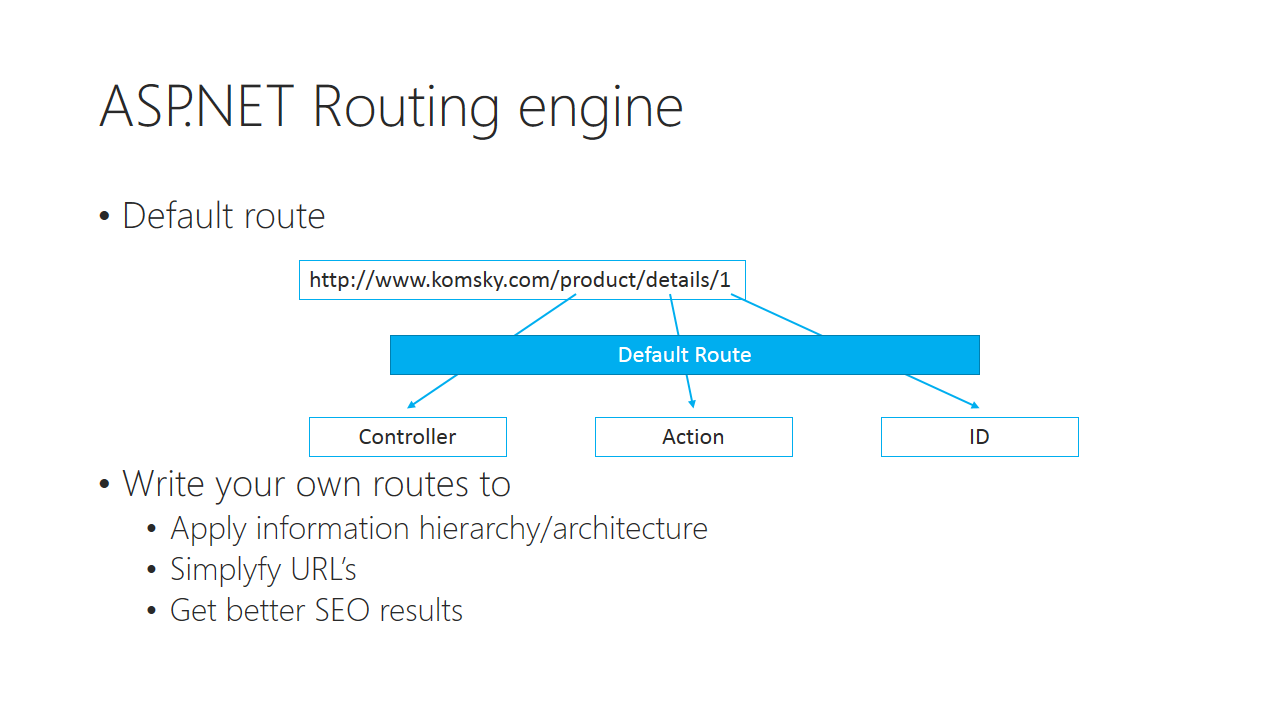
Trainer notes:

Student notes:



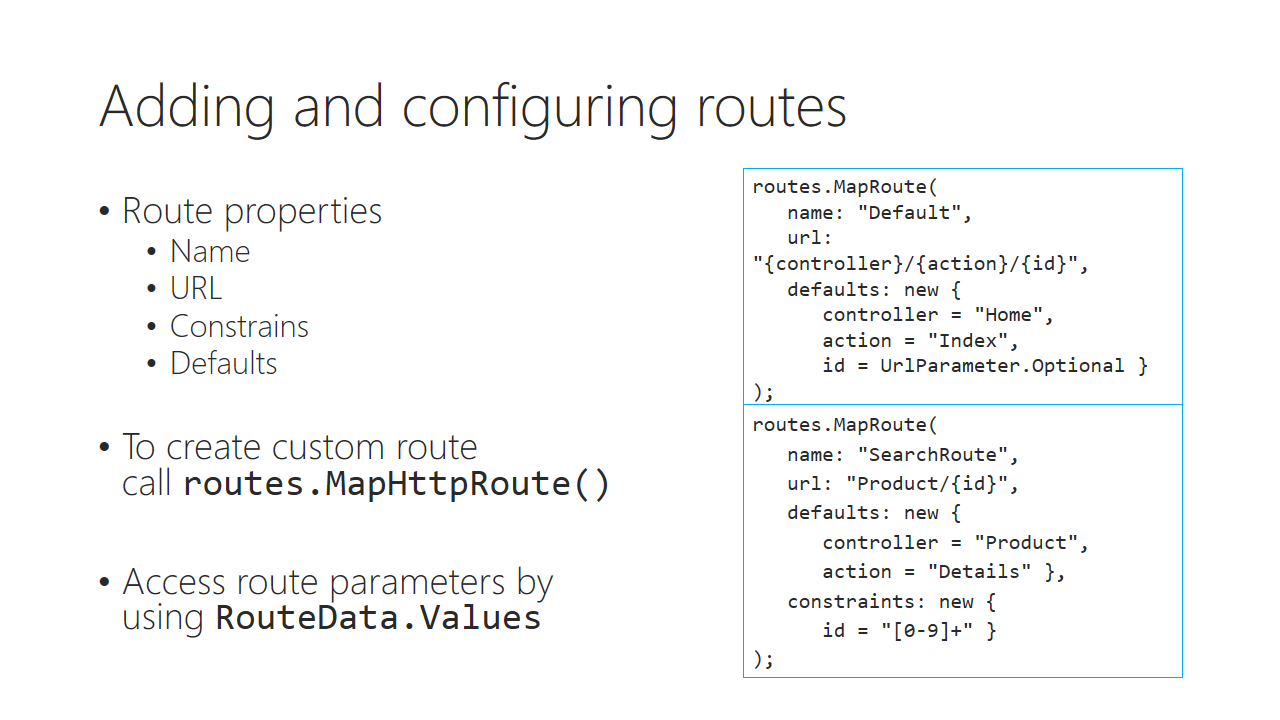
Trainer notes:

Student notes:



Trainer notes:

Student notes:



Trainer notes:

Student notes:



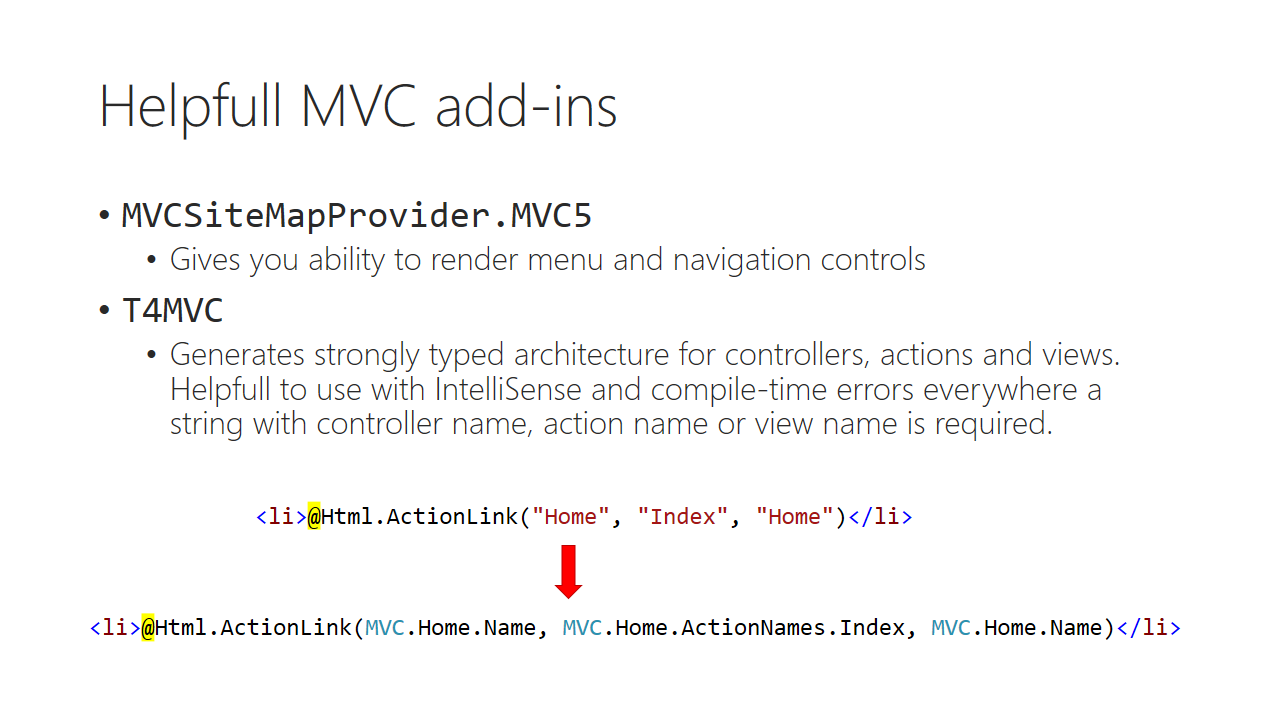
Trainer notes:

Student notes:

Demo: Custom search routing

Trainer notes:

Student notes:



Trainer notes:

Student notes:

Demo: T4MVC

Trainer notes:

Student notes:

# Lab 06

## Content

In this lab you will learn how to work with static routing as well as using routing attributes. The ultimate plan is to introduce search feature into our Ticketing app, as well as provide better SEO experience with urls

## General steps

1. Using static routing and implementing search feature
2. Using route attributes

## Task 1 – Using static routing

In this task we create new static route for searching purposes.

1. Open TicketController class in **Komsky.Web** project and add new method called Search. This method should accept one parameter called searchterm of type string.
2. Implement searching for tickets by introducing new method on \_ticketHandler called SearchTickets. The code could look like this:

public virtual ActionResult Search(string searchterm)

{

IEnumerable<TicketViewModel> foundTickets = \_ticketHandler.SearchTickets(searchterm).Select(TicketViewModelFactory.Create);

ViewBag.Title = "Search results";

if (foundTickets.Any())

{

ViewBag.Message = "Following tickets were matching your criteria:";

}

else

{

ViewBag.Message = "No tickets found";

}

return View(MVC.Ticket.Views.Index, foundTickets);

}

1. Update Index View to make use of ViewBag properties:

@{if (ViewBag.Title == null){ViewBag.Title = "Index";} }

<h2>ViewBag.Title</h2>

<span>@ViewBag.Message</span>

For implementing SearchTickets method, we will need to introduce some major changes into the system. So far we were using IBaseHandler interface for all the handlers, but Searching for tickets is ticket specific, so adding it to a global handler would make no sense.

1. Go to **Komsky.Services/Handlers** folder and add new interface called ITicketHandler and inherit from IBaseHandler<TicketDomain> interface
2. Add SearchTickets method with IEnumerable<TicketDomain> return type and string searchterm input parameter.

public interface ITicketHandler : IBaseHandler<TicketDomain>

{

IEnumerable<TicketDomain> SearchTickets(string searchterm);

}

1. Go to TicketHandler and add implementation of ITicketHandler interface
2. Implement this interface by adding SearchTickets method. It could look like this:

public IEnumerable<TicketDomain> SearchTickets(string searchterm)

{

return DataFacade.Tickets.GetAll()

.Where(x => x.Title.Contains(searchterm) ||   
 x.Description.Contains(searchterm))

.Select(TicketDomainFactory.Create);

}

1. Navigate back to TicketController and change field type for \_ticketHandler from IBaseHandler to ITicketHandler.
2. Update your constructor parameter type.
3. Bind ITicketHandler to TicketHandler in Ninject method.
4. Build and debug any errors, especially update your tests classes
5. Run your tests, to make sure everything works fine.
6. Run your app and test newly created method by navigating to /Tickets/Search/?searchterm=Superticket URL
7. Play with ticket names and descriptions to make sure searching works as expected
8. Go to RouteConfig class in Komsky.Web/App\_Start folder.
9. Duplicate the default route entry by copy-pasting it. Start modifying the upper one
   1. name: Search
   2. url: "Search/{searchterm"}
   3. defaults:
      1. controller: Ticket
      2. Action Search
      3. Id – change into searchterm = ""
10. Build and test newly created route by navigating to /Search/yoursearchterm url.
11. Go back to your solution and add following code below Create new link in **Index.cshtml** of Tickets view (don’t worry, we will make it beautiful later):

<div>

<input type="text" id="searchfield"/>

<button id="searchbutton">Search</button>

</div>

1. Go to global.js file and add implement searching feature:

GLOBAL.Search = GLOBAL.Search ||

{

Search: function() {

window.location.href = '/Search/' + $("#searchfield").val();

},

Init: function() {

$("#searchbutton").on("click", function() {

GLOBAL.Search.Search();

});

}

}

1. Build and test it.

## Task 2 – Using Route attributes

1. Go to Product controller and add new method called Tickets. This method should return tickets for particular product, so provide productId as input parameter.
2. Add instance of ITicketHandler and initialize it as usual.
3. Finish implementing Tickets method, using brand new TicketsForProduct TicketHandler method. The code could look like this:

public virtual ActionResult Tickets(int id)

{

IEnumerable<TicketViewModel> ticketsForProduct =

\_ticketHandler.TicketsForProduct(id).Select(TicketViewModelFactory.Create);

ViewBag.Title = "Product tickets";

if (ticketsForProduct.Any())

{

ViewBag.Message = "Following tickets were created for this product:";

}

else

{

ViewBag.Message = "No tickets were created for this product";

}

return View(MVC.Ticket.Views.Index, ticketsForProduct);

}

1. Add this method to ITicketHandler and implement it with TicketHandler class:

public IEnumerable<TicketDomain> TicketsForProduct(int productId)

{

return DataFacade.Tickets.GetAll()

.Where(x => x.ProductId == productId).Select(TicketDomainFactory.Create);

}

1. Head back to ProductController and decorate Tickets method with following attribute:

[Route("Product/{id}/Tickets")]

1. Open **Details.cshtml** view from Product, and add link to display product tickets as following:

<p>

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

@Html.ActionLink("See product tickets", MVC.Product.ActionNames.Tickets, new { id = Model.Id }) |

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

</p>

1. Update Global.asax file by adding following line before registering conventional routes:

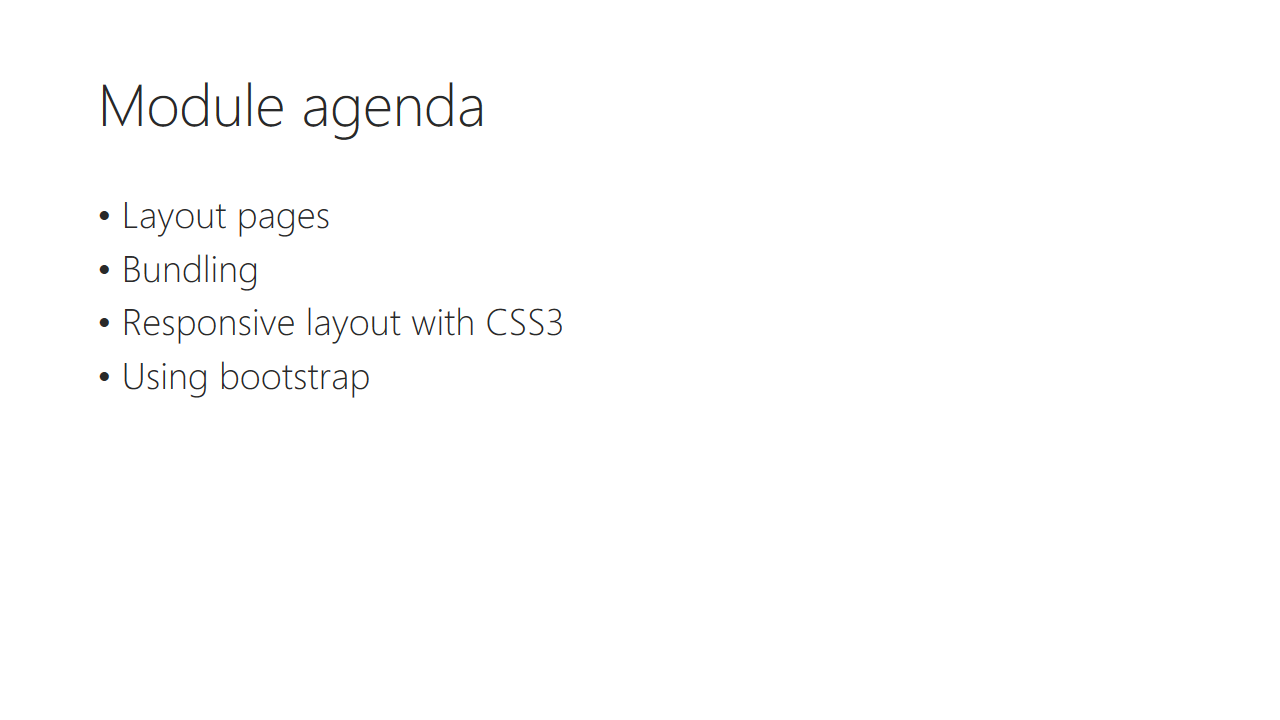
**RouteTable.Routes.MapMvcAttributeRoutes();**

RouteConfig.RegisterRoutes(RouteTable.Routes);

1. Run and test your solution by navigating to product details and checking for tickets for this product. Notice the URL address when displaying tickets for product.
2. When testing, notice how url links generated with Html.ActionLink have changed. This wouldn’t be possible with hardcoded anchors.

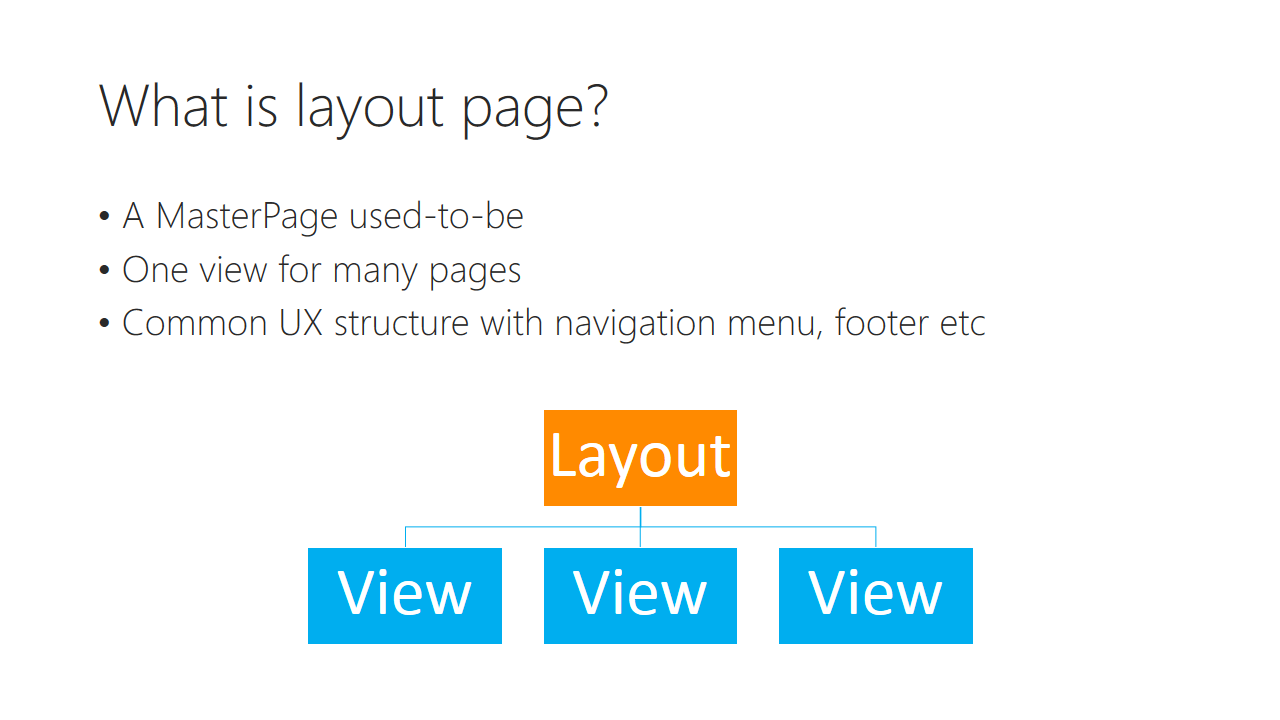
In this lab you have learned how to create and use custom routes.

# Module 07 Designing And Styling Interface



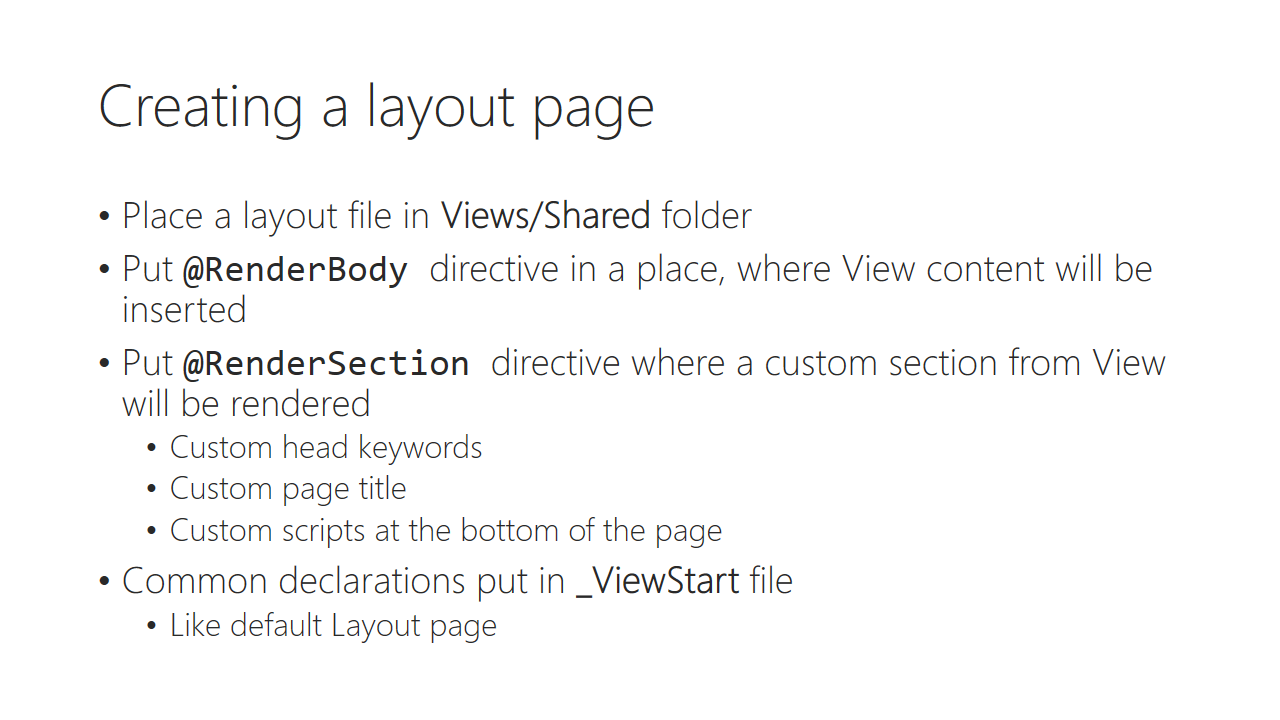
Trainer notes:

Student notes:



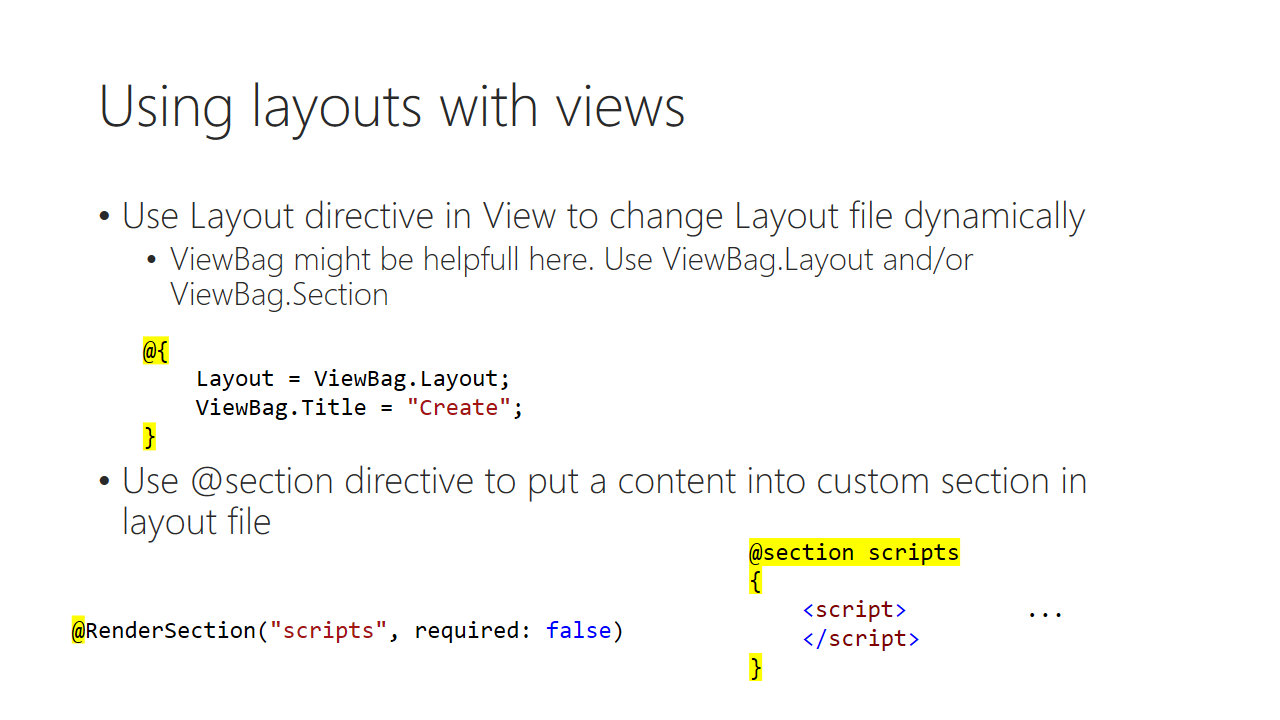
Trainer notes:

Student notes:



Trainer notes:

Student notes:



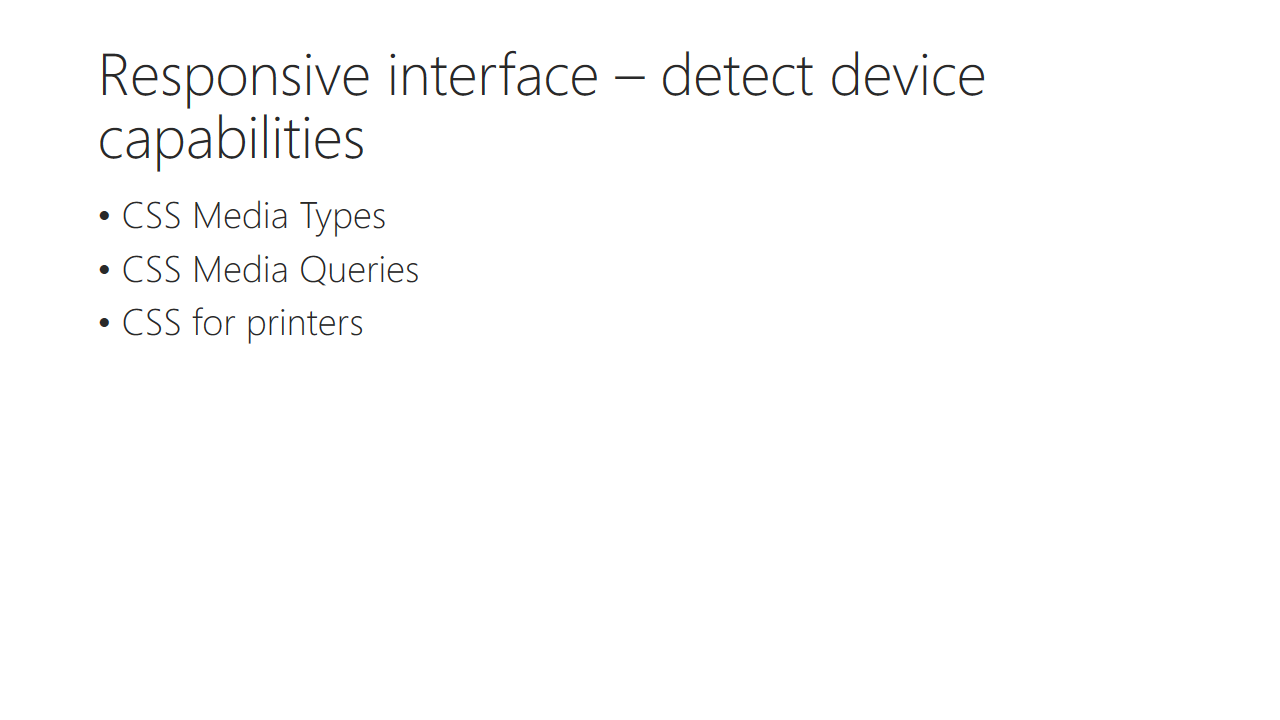
Trainer notes:

Student notes:



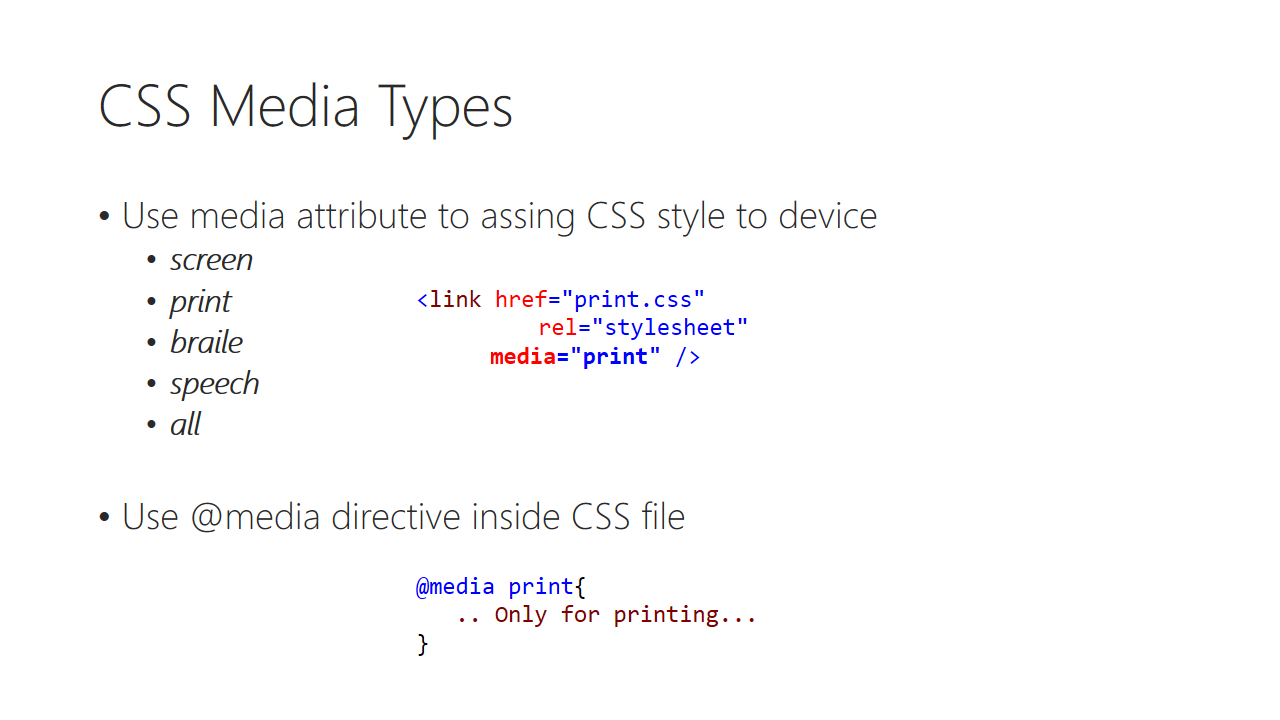
Trainer notes:

Student notes:



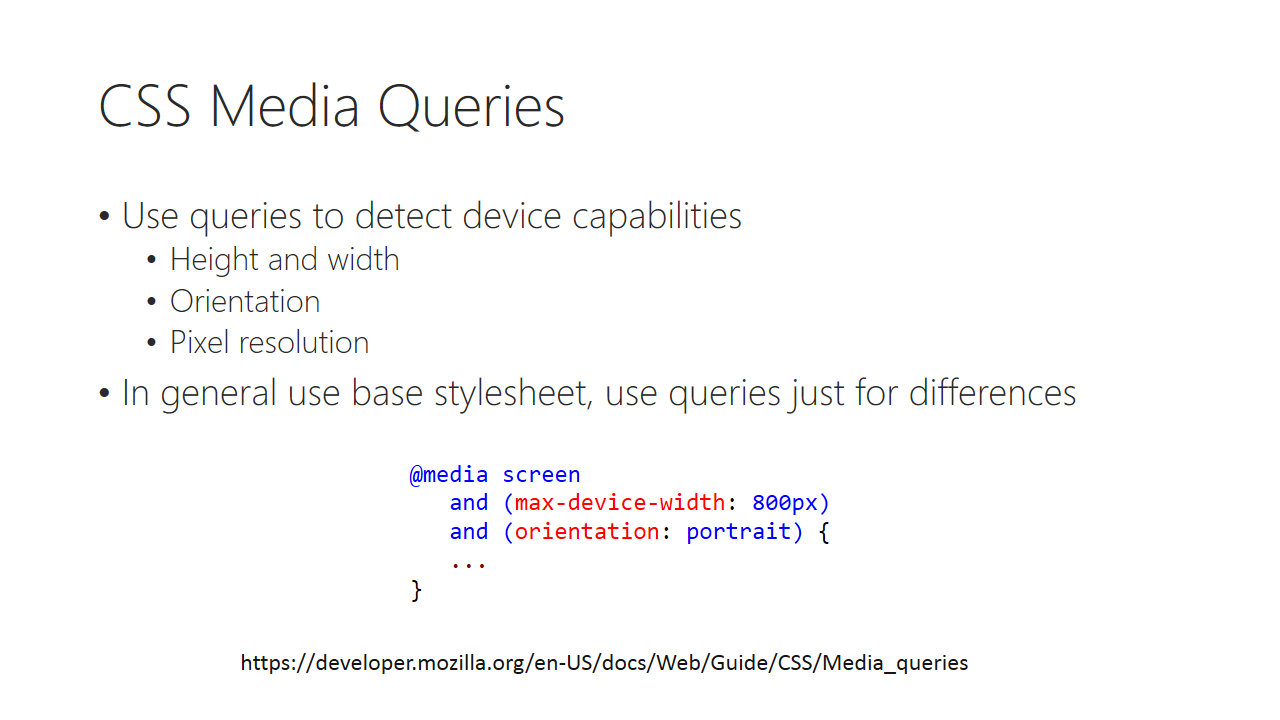
Trainer notes:

Student notes:



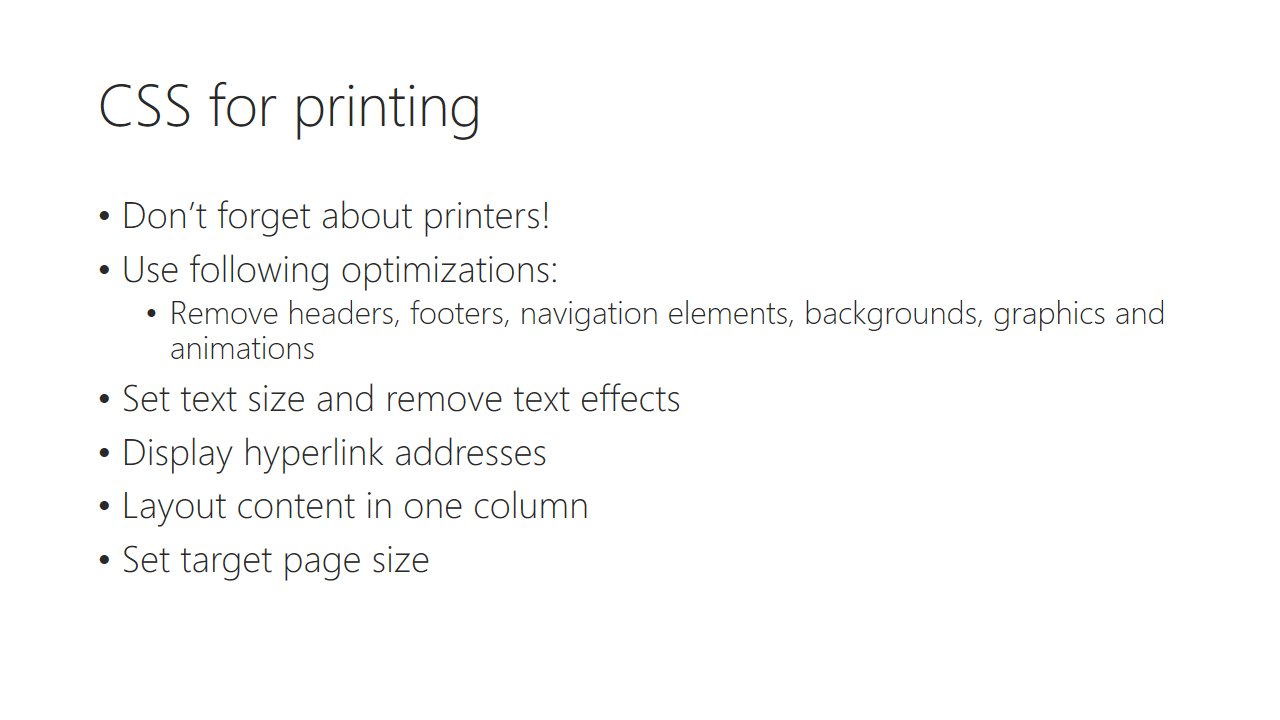
Trainer notes:

Student notes:



Trainer notes:

Student notes:



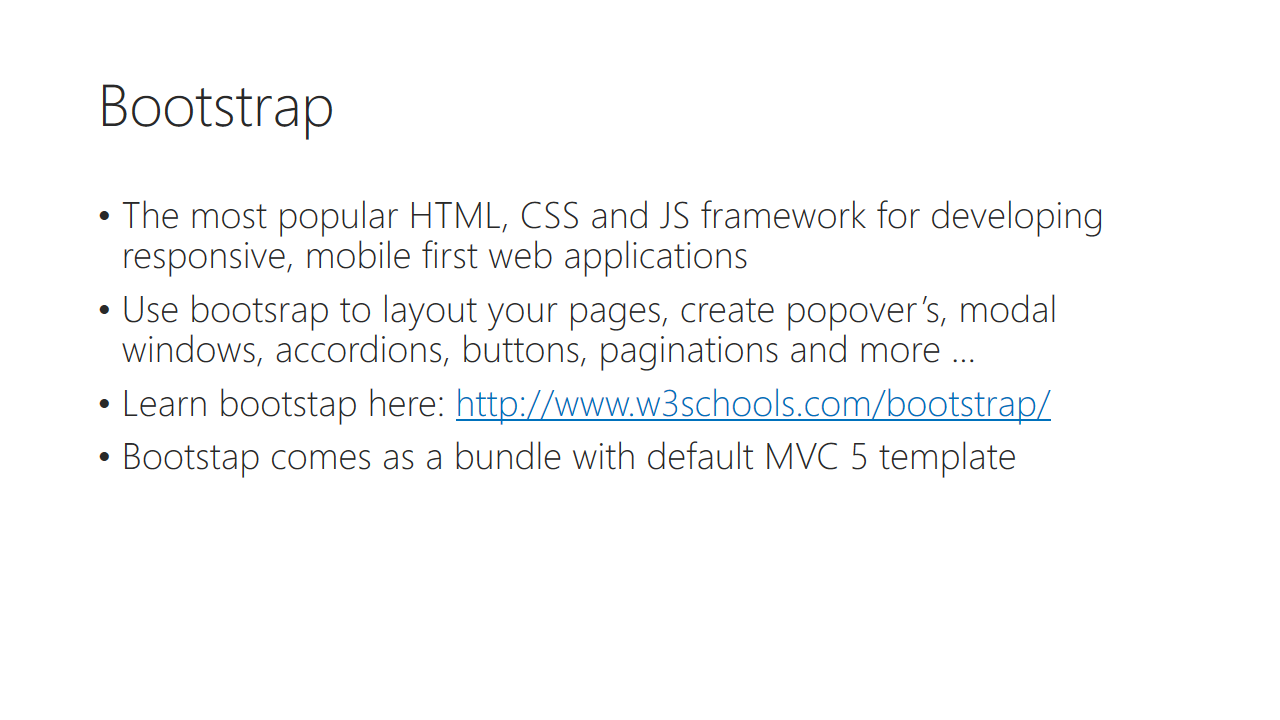
Trainer notes:

Student notes:

Demo: Who cares about printable pages?

Trainer notes:

Student notes:



Trainer notes:

Student notes:



Trainer notes:

Student notes:

Demo: using bootstrap

Trainer notes:

Student notes:

# Lab 07

## Content

In this lab we take closer look at bootstrap capabilities, and integrate beautiful responsive template, that adjust to device used to display our web application

## General steps

1. Using classes to change display of elements
2. Implementing bootstrap template

## Task 1 – Using bootstrap classes

Remember that ugly Search field, we are using in Index view of Ticket controller? Let’s change it for better user experience, by using built-in bootstrap.

1. Update bootstrap to newest version by running following command in **Komsky.Web** project:

PM> install-package Bootstrap

1. Yes, overwrite those files! All of them ☺
2. Update jQuery as well with similar command.
3. Install one more very useful package – FontAwesome.
4. To make use of FontAwesome, we need to reference it in bundles config. Modify bundle css definition in RegisterBundles method:

bundles.Add(new StyleBundle("~/Content/css").Include(

"~/Content/bootstrap.css",

"~/Content/site.css",

**"~/Content/font-awesome.css"**));

1. Open **Index.cshtml** file in **Views/Ticket** folder.
2. Run website by hitting F5 to see how it looks currently. As you can see, search field and button is positioned directly below messange and create new link. We would like to position it on the right, since there’s plenty of empty space – at least on big screens!

*Notice that when editing only cshtml files you don’t have to stop debugging. Use this fact to speed up development process and after modifying view file just refresh browser window.*

1. First, we need to create mockup layout. Below search div, add another with following class definitions:

<div class="row">

<div class="col-lg-7"></div>

<div class="col-lg-5"></div>

</div>

1. Move to the first column div message and create new action link. Move your search div content into second column, deleting the div itself. Effect should resemble this:

<h2>@ViewBag.Title</h2>

<div class="row">

<div class="col-lg-7">

<span>@ViewBag.Message</span>

<p>

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

</p>

</div>

<div class="col-lg-5">

<input type="text" id="searchfield" />

<button id="searchbutton">Search</button>

</div>

</div>

1. Add following classes to your searchfield: form-control pull-left
2. Add following classes to your button: btn btn-default pull-right

<input type="text" id="searchfield" class="form-control pull-left" />

<button id="searchbutton" class="btn btn-default pull-right">Search</button>

1. Add placeholder to your searchfiled: Search for tickets…

<input type="text" id="searchfield" class="form-control pull-left" placeholder="Search for tickets..."/>

1. Add magnifying glass icon into search button, by adding following code after Search text command: <i class="fa fa-search"></i>
2. Open **\_Ticket.cshtml** file, and center Edit and Delete links by adding text-center class to wrapping div
3. Test your solution by resizing browser window. How does your search field behaves with smaller widths?

In this task I provided a glimpse of how bootstrap works by using html classes instead of css styles directly. With bootstrap it’s very easy to lay out your elements. We will continue to use bootstrap in next task, and also in next module, by implementing modal windows.

## Task 2 – Integrating bootstrap theme

In this task we will integrate very nice bootstrap theme to make our application even nicer. At first, take a look on this theme here:

<http://ironsummitmedia.github.io/startbootstrap-creative/>

During this lab use sources provided by links on this page, but would this page becomes unavailable, you can download this theme from Materials/Resources folder on this project’s github.

1. Download current version of this theme here:   
   http://startbootstrap.com/template-overviews/creative/
2. Unpack it to some location on your computer and run **index.html**
3. Take some time to analyse html content of **index.html** as this is the heart of this template. What are its main parts? What files it’s referencing? I especially like main navigation header that blends into background photo when page is positioned on top and section with photo-links, that blends into orange when mouse-hovered.
4. Let’s start with implementing actual HTML into **\_Layout.cshtml** file. Go ahead and duplicate it. A copy will be used for a backup of main theme as well as reference of previous features, that we need to merge. Name the old one **\_OldLayout.cshtml**
5. Replace all content of current **\_Layout.cshtml** with theme’s index.html. Run your application. What happens? We need to place RenderBody somewhere.
6. Collapse all of sections inside body tag, and cut out header and every section tag, keeping nav tag intact.
7. Navigate to **Views/Home/Index.cshtml** and replace its content with your clipboard content (CTRL+A, CTRL+V). Close this file tab, and head back to **\_Layout.cshtml**.
8. Call @RenderBody() in place where you just cut content.
9. Add <footer> tag just below RenderBody. Render footer section there, as well as copyright information with AppSettings application name.
10. Again, build and run your project to see the effect.
11. All right, we can see … this ugly content of our page. Is this really theme we are implementing?
12. Open Developer Tools and tackle every error showed on console. These are described below.
13. First error is referencing bootstrap. Replace jQuery script call on the bottom of the page with our old jQuery bundle call.

Change this:

<!-- jQuery -->

<script src="js/jquery.js"></script>

Into this:

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

1. Repeat above step, but with bootstrap.
2. Replace core bootstap css with old bootstrap css bundle on top of the page:

Replace this:

<!-- Bootstrap Core CSS -->

<link rel="stylesheet" href="css/bootstrap.min.css" type="text/css">

Into this:

@Styles.Render("~/Content/css")

1. Run your app to boost up your morale ☺ We still have a lot to fix, but it’s getting better, right?
2. OK, seems like previously integrated font awesome works fine. Happily remove stylesheet reference to font-awesome css file, but keep fonts from googleapis
3. Seems like next two lines of code are custom css files. Copy these files into **Content/Theme** folder (create it) and integrate into css bundle.

bundles.Add(new StyleBundle("~/Content/css").Include(

"~/Content/bootstrap.css",

"~/Content/site.css",

"~/Content/font-awesome.css",

"~/Content/Theme/animate.min.css",

"~/Content/Theme/creative.css"));

1. Remove /Content/site.css entry from css bundle, as well as **/Content/site.css** file itself – we won’t need it any longer.
2. Repeat similar steps for custom javascript files, that is: jquery easing, fittext, wow and creative.js files
   1. Seems like jQuery.Easing is actually available via nuget. You can choose if to use nuget of provided file from template.
3. Add old references, required either by MVC or by us:

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

@RenderSection("scripts", required: false)

1. Create folder **Content/Images**. Put there images from theme.
2. Update images references in **Index.cshtml** file of home controller.
3. Run your application. Seems that last resource missing is **header.jpg** file, but it’s not being referenced directly in **Index.html**, so it must be referenced somewhere else. Locate that reference by using search option in visual studio– search through current project rather that current file (CTRL + F – Project scope). Update this last reference as well.
4. Go back to **\_Layout.cshtml** and replace static navigation links with those old ones.
5. Add missing \_LoginPartial reference below <ul> navigation list.
6. Add Modernizr Script bundle on top of the page
7. Update Title tag.
   1. Provide ViewBag.Title in Index method.
   2. Delete UserDetails if section as we no longer require it
8. Build run your page. Use Browser Developer tools to check for any issues with scripts, styles, etc.
9. Front page should be working fine by now, but subpages like Ticketing system might have issues with navigation menu. This is caused by header element that is required by bootstrap template. Let’s refactor layout, so if header is provided by view, it will be used, but if not, a default header element will be present.
10. Go to **Views/Home/Index.html** and following code on top of the page:

@{

ViewBag.HeaderPresent = true;

}

1. Go to **Views/Shared/\_Layout.cshtml** and add following code just before calling @RenderBody:

@if (ViewBag.HeaderPresent == null)

{

<header class="header-small">

</header>

}

1. Go to **Content/Theme/creative.css** and add following code at the bottom of the file:

.header-small {

min-height: 50px;

}

1. If everything works fine, spend some time playing with content of main page, eg. Put your own content, remove unnecessary elements.

We have integrated bootstrap theme into our project. As you can see, still a lot can be done here, but let’s leave this for front-end developers, so we can focus on back-end. In next lab, we will still use bootstrap features to provide AJAX modal forms.

# Lab 08 A

## Content

In this lab we implement Bootstrap AJAX modals. Basic bootstrap modals are using hidden page portion to display it on top of visible space. But what if we have more than too much of them, and their content is different? We want a single modal hidden page portion page, but with dynamic content loaded on demand from MVC action method.

## General steps

1. Refactoring Ticket details partial view as bootstrap modal
2. Dynamically loading modals using ActionMethod and jQuery

## Task 1 – Refactoring Ticket details partial view as bootstrap modal

1. Open <http://www.w3schools.com/bootstrap/bootstrap_modal.asp> in browser and analyse structure of bootstrap modals
   1. Look especially on modal content div hierarchy
   2. See how modal works by using ‘Try it Yourself’ button
   3. Don’t close this tab, we will need it later as a reference and source code
2. In Visual Studio, open **\_Ticket.cshtml** partial view, and wrap its content with modal div as follows:
   1. Remember to keep model directive on top of the page.
   2. Remove edit and delete links div.
   3. Move model’s title from h4 tag to modal title h4 tag. Remove old h4 tag as well as hr tag.
   4. Remove general ticket’s outer div section, so that <dl> tag is direct child of <div class=’modal-body’>.
   5. The code should looks like that:

@model Komsky.Web.Models.TicketViewModel

<div class="modal-content">

<div class="modal-header">

<button type="button" class="close" data-dismiss="modal">&times;</button>

<h4 class="modal-title">@Html.DisplayFor(model => model.Title)</h4>

</div>

<div class="modal-body">

<dl class="dl-horizontal">

<dt> @Html.DisplayNameFor(model => model.Description) </dt>

<dd>@Html.DisplayFor(model => model.Description)</dd>

<dt>@Html.DisplayNameFor(model => model.AgentReply)</dt>

<dd>@Html.DisplayFor(model => model.AgentReply)</dd>

<dt>Owner's name</dt>

<dd>@Html.DisplayFor(model => model.Owner.UserName)</dd>

<dt>OTHER DISPLAY NAME FORs</dt>

<dd>OTHER DISPLAY FORs</dd>

</dl>

</div>

<div class="modal-footer">

<button type="button" class="btn btn-default" data-dismiss="modal">Close</button>

</div>

</div>

1. Open **Views/Home/Index.cshtml**, locate section tag with id=”portfolio” and cut it into clipboard
2. Open **Views/Ticket/Index.cshtml** and paste the content just above foreach loop (delete the loop)
3. Collapse bootstrap’s divs representing project, and delete them all except one.
4. Cut into clipboard last project’s div and paste the clipboard content inside the foreach loop, replacing its whole content
5. Move the loop into the bootstrap wrapper, where you just cut the last project.
6. Replace <img> tag with following code (use **\_Ticket.cshtml’s** content to speed up your work:

<dl class="dl-horizontal">

<dt> @Html.DisplayNameFor(model => model.Title) </dt>

<dd>@Html.DisplayFor(model => item.Title)</dd>

<dt> @Html.DisplayNameFor(model => model.Description) </dt>

<dd>@Html.DisplayFor(model => item.Description)</dd>

<dt> @Html.DisplayNameFor(model => model.TicketPriority) </dt>

<dd>@Html.DisplayFor(model => item.TicketPriority)</dd>

</dl>

1. Replace the ‘Category’ text with @item.Title and ‘Project’ with @item.Description
2. Open **Content/Theme/creative.css**, locate the .portfolio-box class selector and add following style definition: min-height: 100px;
3. Save and test your solution, see the effect of your work.

## Task 2 – Dynamically loading modals using ActionMethod and jQuery

1. Go back to Visual Studio and **Index.cshtml** file. Add two data-\* attributes to portfolio-box anchor:

<a href="#" class="portfolio-box" **data-toggle="modal" data-target="#ticketDetailsModal"**  **data-id="@item.Id">**

1. Go back to w3schools page with bootstrap’s modal description. Copy HTML below <!—Modal --> comment and paste it on the bottom of **Index.cshtml**
2. Update the id of pasted code into: ticketDetailsModal

<div **id="ticketDetailsModal"** class="modal fade" role="dialog">

1. Save and test your modal. For now it just displays dummy data, but we are on good track of putting here our dynamic partial modal.
2. Open **TicketController.cs** file and add new action method called TicketModal. This method should return PartialView, and read one parameter called ticketId of type int.
   1. You might want to create a new region in your controller called AJAX
3. Inside the method, use ticketId to find and create model and pass it to PartialView:
   1. For clarity, we don’t use any validation here, but in reality you should validate if model with such id exists and display correct information.

#region AJAX

[HttpGet]

public virtual PartialViewResult TicketModal(int id)

{

var model = \_ticketHandler.GetById(id).CreateTicketViewModel();

return PartialView(MVC.Ticket.Views.\_Ticket, model);

}

#endregion

1. Open Scripts.js and add following code on bottom of it
   1. This is a generic code that allows for multiple types of modal windows anywhere on your website. It’s composed of three parts:
      1. List of modals windows
      2. Initialization function, that is run once
      3. ModalSetup function that is run for every modal window in modal window list. It searches for that modal, and attaches on show event. When the modal is shown, it’s content is loaded dynamically by calling $.ajax() method:

GLOBAL.Modals = GLOBAL.Modals ||

{

ModalList: [{ ModalSelector: "#ticketDetailsModal", ModalActionMethod: "/Ticket/TicketModal" }], // You can add more modals here

Init: function () {

GLOBAL.Modals.ModalList.forEach(GLOBAL.Modals.ModalSetup);

},

ModalSetup: function (modal, index, array) {

$(modal.ModalSelector).on('show.bs.modal', function (event) {

var button = $(event.relatedTarget); // Button that triggered the modal

var id = button.data('id');

var modalWindow = $(this);

var ajaxReplace = modalWindow.find('.modal-dialog');

ajaxReplace.text(''); // Clear any older data

$.ajax({ // Call MVC for Partial view

url: modal.ModalActionMethod,

type: 'GET',

data: { id:id },

success: function (partialView) {

ajaxReplace.html(partialView); // Replace modals content

},

error: function() {

console.log('modal ajax error');

}

});

});

}

}

1. Run and test your solution. Use Browser Developer tools to debug any errors.
2. If you have finished before group spend the rest of the time for ‘polishing’ your solution for display quirks of yours. Can you implement displaying TicketPriority with colors, eg. Highest is red, and Lowest is green? Maybe add an FontAwesome icons? Implement product background image from database?

At this point we have working generic dynamic bootstrap modals ready for action. If you need a modal on any other page, just create it as partial view, add action method to serve it, create html container and add it to ModalList JavaScript array.

# Lab 08 B

## Content

In this lab we are going to learn how to use [OutputCache] attribute as well as MemoryCache object for time-consuming methods

Output cache, object cache

## General steps

1. Implementing Output cache
2. Implementing Memory cache - optional

## Task 1 – Output cache

1. Navigate to **HomeController.cs** and find About method
2. Add following line to it

Thread.Sleep(10000);

ViewBag.ServerTime = DateTime.Now.ToLongTimeString();

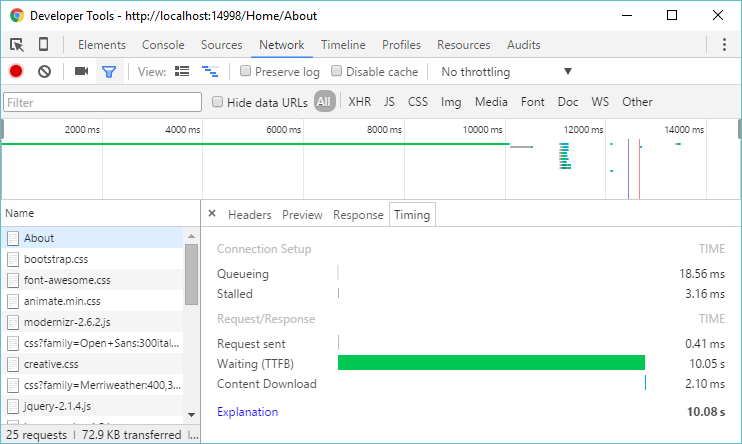
1. Open **About.cshtml** view from home controller and display content of ViewBag.ServerTime variable just below the message:

<h3>@ViewBag.Message</h3>

**<h3>Server time: @ViewBag.ServerTime</h3>**

<p>Use this area to provide additional information.</p>

1. Run the solution. Hit F12 to open Development Tools to open Network panel. When Network panel is opened navigate to /Home/About address. Take a note of waiting time for About request:



1. Go back to Visual Studio. Describe About method with [OutputCache] attribute. Set Duration to 30 (it means 30 seconds):

**[OutputCache(Duration=30)]**

public virtual ActionResult About()

1. Repeat the step 4. You should get the same result. Keep dev tools opened.
2. Refresh the page and again note the time. This time it’s milliseconds! But the server time didn’t refresh and shows invalid time (old one)
3. Hit F5 to make sure that the page is responsive but server time is not refreshed.
4. Wait for 30 seconds and again refresh page. What is the response time and server time on page?
5. Stay here in Visual studio, because we continue from this point in task 2.

## Task 2 – Memory cache - Optional

1. Copy into clipboard two lines you just created – Thread.Sleep and ServerTime
2. Delete the first line with Thread.Sleep
3. In second line, replace DateTime.Now with GetServerTime(). This method does not exists, so create it.
4. When method has been created, paste into it content from your clipboard. The whole code might looks like this:

public virtual ActionResult About()

{

ViewBag.Message = "Your application description page.";

ViewBag.ServerTime = GetSErverTime();

return View();

}

private dynamic GetSErverTime()

{

Thread.Sleep(10000);

string serverTime = DateTime.Now.ToLongTimeString();

return serverTime;

}

1. Add new **Cache.cs** class with following content [Snippet:MemoryCache]

using System.Runtime.Caching;

public interface IMemoryCacheService

{

MemoryCache MemoryCache

{

get;

set;

}

}

public class MemoryCacheService : IMemoryCacheService

{

public MemoryCacheService()

{

MemoryCache = new MemoryCache("ServerTime");

}

public MemoryCache MemoryCache

{

get;

set;

}

}

1. Add following binding to Ninject:

kernel.Bind<IMemoryCacheService>().To<MemoryCacheService>().InSingletonScope();

1. InSingletonScope() is important so Ninject knows to create only one copy and then reuse it every time it is asked forA sd
2. Update your HomeController to use updated field and DI constructor:

private readonly IDataFacade \_dataFacade;

**private readonly IMemoryCacheService \_memoryCache;**

public HomeController(IDataFacade dataFacade, **IMemoryCacheService memoryCache**)

{

\_dataFacade = dataFacade;

**\_memoryCache = memoryCache;**

}

1. Replace GetServerTime method body with following:

private dynamic GetSErverTime()

{

if (!\_memoryCache.MemoryCache.Contains("ServerTime"))

{

Thread.Sleep(10000); //this is just for simulating long-running job

\_memoryCache.MemoryCache.Add(new System.Runtime.Caching.CacheItem("ServerTime", DateTime.Now.ToLongTimeString()), new System.Runtime.Caching.CacheItemPolicy());

}

String serverTime = \_memoryCache.MemoryCache.Get("ServerTime").ToString();

return serverTime;

}

1. Build and debug your application. Run it.
2. Open Developer Tools in browser, open Network tab and navigate to /Home/About
3. Note the first requests timing. Refresh the page
4. Note the second request timing. Is the server time cached?

In this LAB you have learned how to use caching to speed up access to your website.

# Lab 09 A

## Content

In this lab we will learn how to secure our website with authentication and authorization. Because a lot of complex code is going to be introduced, I strongly advice to import and use snippets for this purpose.

## General steps

1. Creating new User controller for admins user management
2. Securing User controller with attributes for admin-only access
3. Seeding first admin user
4. Refactoring tickets for user creation only and agent reply only

## Task 1 – User Controller

1. Add new model class called UserCreationViewModel, as follows   
   [Snippet: UserCreationViewModel]:

[Validator(typeof(UserCreationValidator))]

public class UserCreationViewModel

{

public String Email { get; set; }

public String Password { get; set; }

[Display(Name = "Repeat password")]

public String RepeatPassword { get; set; }

public Roles Role { get; set; }

}

public class UserCreationValidator : AbstractValidator<UserCreationViewModel>

{

private readonly PasswordValidator \_passwordValidator;

public UserCreationValidator()

{

\_passwordValidator = new PasswordValidator();

\_passwordValidator.RequireNonLetterOrDigit = true;

\_passwordValidator.RequireLowercase = true;

\_passwordValidator.RequireUppercase = true;

\_passwordValidator.RequiredLength = 6;

RuleFor(x => x.Password).Must(PasswordCustomLogic).WithMessage("Passwords must be at least 6 characters with minimum two uppercase letters, two digits and two non-character makrs.");

RuleFor(x => x.RepeatPassword).Equal(x => x.Password).WithMessage("Password must match!");

RuleFor(x => x.Email).NotNull().EmailAddress();

}

private bool PasswordCustomLogic(string password)

{

if (String.IsNullOrEmpty(password))

{

return true; //password can't be empty

}

//using built'in password validation

var validationResult = \_passwordValidator.ValidateAsync(password).Result;

return validationResult.Succeeded;

}

}

1. Add new empty controller called UserManagementController
2. Introduce new field and property of type ApplicationUserManager. Initialize those as following [Snippet: UserManagementManager]:

private ApplicationUserManager \_userManager;

public ApplicationUserManager UserManager

{

get

{

return \_userManager ?? HttpContext.GetOwinContext().GetUserManager<ApplicationUserManager>();

}

private set

{

\_userManager = value;

}

}

public UserManagementController(ApplicationUserManager userManager)

{

UserManager = userManager;

}

1. Because ActionMethods using UserManager are so freakin complicated, there’s no point in rewriting two pages of C# code. Just use following snippet to fill the controller with methods: [Snippet:UserManagerControllerActionMethods]
2. Navigate to Views/UserManager and add following views using snippets:
   1. **\_CreationConfirmation.cshtml**
      1. Empty partial view
      2. [Snippet:ViewCreationConfirmation]
   2. **Create.cshtml**
      1. Template: Create
      2. Model class: UserCreationViewModel *(make sure it’s not UserCreationValidation!)*
      3. After scaffolding view, replace EditorFor’s in Password and Repeat password to PassfordFor Html.Helper:

Change this:

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

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

To this:

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

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

* 1. **Edit.cshtml**
     1. Template: Edit
     2. Model class: UserCreationViewModel
     3. After scaffolding view use snippet: [Snippet:ViewsUserEditMessages] just below the <h2>Edit</h2>:

@if (!String.IsNullOrEmpty(ViewBag.Error))

{

<div class="alert alert-danger alert-white text-center">@ViewBag.Error</div>

}

@if (!String.IsNullOrEmpty(ViewBag.Message))

{

<div class="alert alert-info text-center">@ViewBag.Message</div>

}

* 1. **Index.csthml**
     1. Template: List
     2. Model class: UserCreationViewModel
     3. After scaffolding remove link *Details* on the bottom
     4. Update Edit and Delete links, by uncommenting id and providing Email there:

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

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

1. Decorate UserManagementController with following attribute:

[Authorize(Roles=”Admin”)]

1. Open **Configuration.cs** file from Migrations folder in Data Access Layer and replace Seed method with following snippet: [Snippet: DbContextSeed]

protected override void Seed(Komsky.Data.DataAccess.ApplicationDbContext context)

{

var userManager = new UserManager<ApplicationUser>(new UserStore<ApplicationUser>(context));

var roleManager = new RoleManager<IdentityRole>(new RoleStore<IdentityRole>(context));

//seed roles

var rolesNames = new List<string> { "Admins", "Agents", "Customers" };

foreach (var roleName in rolesNames)

{

if (!roleManager.RoleExists(roleName))

{

var roleresult = roleManager.Create(new IdentityRole(roleName));

}

}

//seed user

string userName = "admin@komsky.com";

string password = "Pa$$w0rd";

var user = new ApplicationUser();

user.UserName = userName;

user.Email = userName;

var adminResult = userManager.Create(user, password);

if (adminResult.Succeeded)

{

var result = userManager.AddToRole(user.Id, rolesNames[0]);

}

base.Seed(context);

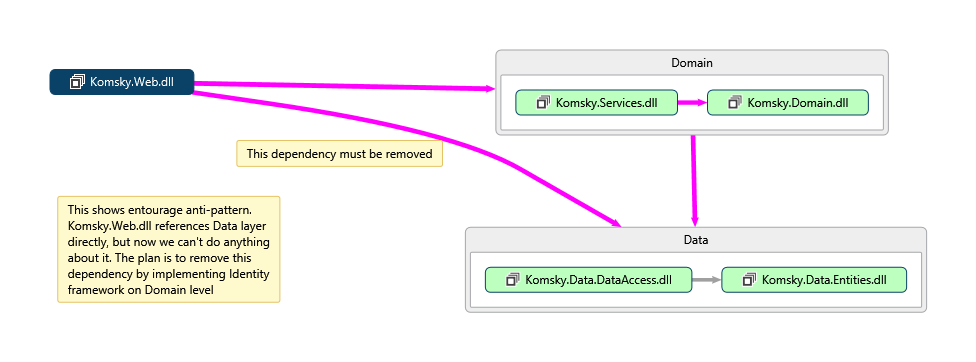
}

1. Update links in **\_Layout.cshtml** by adding User Management
2. Run T4 transformations. Build and debug your application, but don’t run it yet.
3. Run Management Studio, log in to (LocalDb)\v11.0 and delete your database
4. Go back to Visual Studio and run update-database command from Package Manager Console to seed new admin user on fresh database
5. Now run your application, log in as [admin@komsky.com](mailto:admin@komsky.com) / Pa$$w0rd
6. Test adding new users with different roles
7. Log out and log in as different user (not admin). Can you add new users?

# Lab 09 B - Optional

## Content

On Module 01 I’ve explained some design principles by using patterns and removing dependencies. But we didn’t removed dependencies correctly, because it required knowledge about Identity framework. In this Lab we implement Identity on service layer and remove dependency on Data layer, so it can be easily replaced with NHibernate or any ORM.



## General steps

1. Removing dependency on Data layer
2. Implementing Identity
3. Removing references to truly use Stairway pattern

## Task 1 – Removing dependency on Data Layer

If you have Visual Studio 2013 Enterprise or above, you can use Code Map feature to see dependencies visually. Use points from 1 to 2 to see them. If you don’t have Enterprise or Ultimate edition, start on point 3.

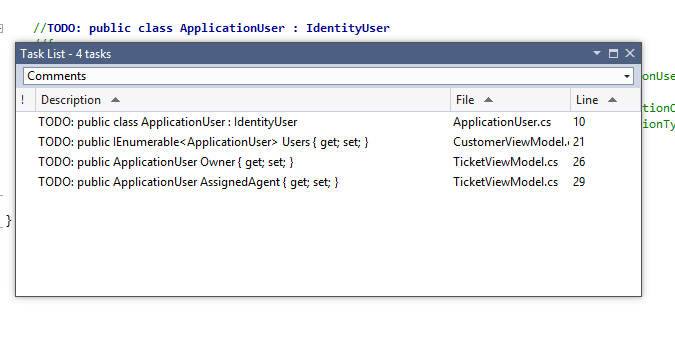
1. Right click on **Komsky.Web** project and choose *Show on Code Map*
2. Repeat this step with **Komsky.Services** and **Komsky.Data.DataAccess** to see dependencies on diagram.
3. Go to **Komsky.Web/References** and remove following references:
   1. Komsky.Data
   2. Komsky.Data.DataAccess
   3. Komsky.Data.Entities
4. Build and debug your project. What build errors have Visual Studio found? Let’s try to fix some of them:
   1. First error might be using Data Façade class for HomeController. Delete usings and objects from NinjectWebCommon and HomeController, as we don’t use it anymore.
   2. Briefly walk through the rest of the errors and delete all the usings that point to any Komsky.Data namespaces.

That’s it. We have successfully removed dependency on **Komsky.Data** layer. Or did we? Now the solution won’t build. So actually, we have just removed references, not dependencies! To fully remove dependency, we need to replace ApplicationUser class with another one – a new one, from domain level.

## Task 2 – Implementing Identity on domain level

In this task we focus on implementing only most basic required fields for user and role, to make solution work. For the simplicity, I keep Identity implementation in Domain and Services projects, but you could create new project for this purpose.

1. Open **Komsky.Data.Entities** project, and **ApplicationUser.cs** file inside it
2. Temporarily comment out ApplicationUser class, adding *TODO:* declaration for easier locating in the future.
   1. This will allow to track every ApplicationUser references, because a build error will be thrown.
   2. Adding *TODO:* declaration on the beginning of the comment allows us to track it with TaskList window, when switched into comments mode.



1. Open **Komsky.Domain** project
2. And add new class called ApplicationUserDomain into **Models/Identity** folder.
   1. Make this class public, inherit from IUser<string> generic interface. Implement missing members from interface.
3. Add new class named ApplicationRoleDomain into **Models/Identity** folder.
   1. Make this class public, inherit from IRole<string> generic interface. Implement missing members from interface.
4. Delete following references from **Komsky.Domain** project:
   1. EntityFramework
   2. EntityFramework.SqlServer
   3. Microsoft.AspNet.Identity.EntityFramework
   4. Every Komsky.Data reference
   5. Every Microsoft.Owin and Owin
5. Open **packages.config** file and manually delete following entries:
   1. EntityFramework
   2. Microsoft.AspNet.Identity.EntityFramework
   3. Every Microsoft.Owin and Owin
6. Build and debug your **Komsky.Domain** project (right click on project and build just this single project). Replace any instances of ApplicationUser to ApplicationDomainUser to fix errors.
7. If you have fixed all the errors and **Komsky.Domain** builds without any errors and warning, head to **Komsky.Services**
8. Again, build your project to see what errors we need to tackle. One of the major ones are Factories – they reference both data layer and domain layer.
9. Remove comment on ApplicationUser class in **Komsky.Data.Entities**, so it becomes available again.
10. Add two new factories for converting between ApplicationUser and ApplicationUserDomain. The factories should internally use following code:

public static class ApplicationUserDomainFactory

{

public static ApplicationUserDomain Create(ApplicationUser applicationUser)

{

return new ApplicationUserDomain

{

UserName = applicationUser.UserName,

Id = applicationUser.Id

};

}

public static ApplicationUserDomain CreateApplicationUserDomain(this ApplicationUser applicationUser)

{

return Create(applicationUser);

}

}

1. Update Ticket factories to use appropriate ApplicationUserDomain back and forth.
2. Build your **Komsky.Services** project and debug any errors. Project should build fine by now
3. Again, Data layer doesn’t need to reference EntityFramework. Remove every reference that has EntityFramework in its name.
4. Update packages.config by removing unnecessary EntityFramework packages.
5. Build again your project. Seems that we have broken something!

*And now comes the hardest part – ApplicationUserManager and ApplicationSignInManager classes! As our target is to remove dependencies on EntityFramework, we can’t rely on built-in classes like IdentityUser, UserStore etc. but rather on interfaces. And interfaces require implementing. If you feel that you are running out of time don’t hesitate to use finished solution, or better parts of it you don’t understand.*

1. Open **IdentityConfig.cs** file and replace every ApplicationUser with ApplicationUserDomain class. Use CTRL+F replace feature on current document. Exactly 17 occurrences should be found and replaced.
2. Add new folder to Komsky.Services called Identity, and move there IdentityConfig.cs
3. Add new class called UserStore and inherit from IUserStore<ApplicationUserDomain> generic interface. Implement missing members
4. We need to implement UserStore in similar manner as any handler, that is use DataFacade class to manage users.
   1. Add IDataFacade private field and initialize it the DI way
   2. Dispose \_dataFacade in Dispose method
5. Implement CreateAsync method by converting UserDomain into user while adding it into datafacade’s ApplicationUsers repository. Then return datafacade’s CommitAsync result.

public Task CreateAsync(ApplicationUserDomain user)

{

\_dataFacade.ApplicationUsers.Add(user.CreateApplicationUser());

return \_dataFacade.CommitAsync();

}

*Ouch! What is going on? Why createApplicationUser is underlined with red indicating error? Visual Studio wants us to add EntityFramework reference. Again, you have experienced Entourage pattern.*

1. Navigate to ApplicationUser definition and replace inheritance from IdentityUser with IUser<string>. Implement missing Properties and comment out GenerateIdentityAsync method.
2. Head back to UserStore definition and implement UpdateAsync and DeleteAsync the same way as CreateAsync.
3. FindByIdAsync method is more difficult, as we don’t have similar method on our repository. Navigate to IApplicationUserRepository and add two method signatures:
   1. Task<ApplicationUser> FindByIdAsync(string id);
   2. Task<ApplicationUser> FindByNameAsync(string name);
4. Implement those in ApplicationUserRepository class as follows:

public Task<ApplicationUser> FindByIdAsync(string id)

{

return Task.FromResult(GetById(id));

}

public Task<ApplicationUser> FindByNameAsync(string name)

{

return Task.FromResult(GetAll().First(x=>x.UserName == name));

}

1. As we are still in ApplicationUserRepository, seems that ApplicationUser is missing Email property. Navigate to definition and add public Email property of type String.
2. Head back to UserStore in **Services** project and finish implementing FindByIdAsync and FindByNameAsync. Use methods from repository you just created and convert to ApplicationUserDomain in ContinueWith method.

public Task<ApplicationUserDomain> FindByIdAsync(string userId)

{

return \_dataFacade.ApplicationUsers

.FindByIdAsync(userId)

.ContinueWith(x=>x.Result.CreateApplicationUserDomain());

}

public Task<ApplicationUserDomain> FindByNameAsync(string userName)

{

return \_dataFacade.ApplicationUsers

.FindByNameAsync(userName)

.ContinueWith(x => x.Result.CreateApplicationUserDomain());

}

1. Initialize ApplicationUserDomainManager instance with our new UserStore class. Also Initialize UserStore with DataFacade instance.

This is not perfect testable DI solution, but must do for now.

1. Add helper from ApplicationUser …
2. Replace inheritance on DbContext, remove throwIv
3. Replace ManageControllers ApplicationUserManager with ApplicationUserDomainManager
4. Replace AccountControllers ApplicationUserManager with ApplicationUserDomainManager
5. Add email to ApplicationUserDomain
6. Update factories
7. Update TicketViewModel
8. Startup.Auth.cs
9. Add PasswordHash, PhoneNumber to ApplicationUser and Domain
10. Update Tests
11. Sdf
12. Sdf
13. Sd
14. Fsd
15. Fsd
16. F
17. Sdf
18. Sdf
19. Sdf

---------------------------------- for future generations ☺

1. We don’t have time to write another factory, and you know from previous modules, that this work can be easily avoided by using one of ready solutions available on the internet. We are going to use QueryableExtensions, as this solution is easy, and allows for application scalability.
2. Add new Class Library project to **Common** folder called **Komsky.Utils**.
3. Download **QueryableExtensions.cs** file from location below or access it from local training materials folder. Add it to your **Komsky.Utils** project

github.com/komsky/training/blob/dev/Komsky/Komsky/Komsky.Utils/QueryableExtensions.cs

1. Reference **Komsky.Utils** in **Komsky.Services**