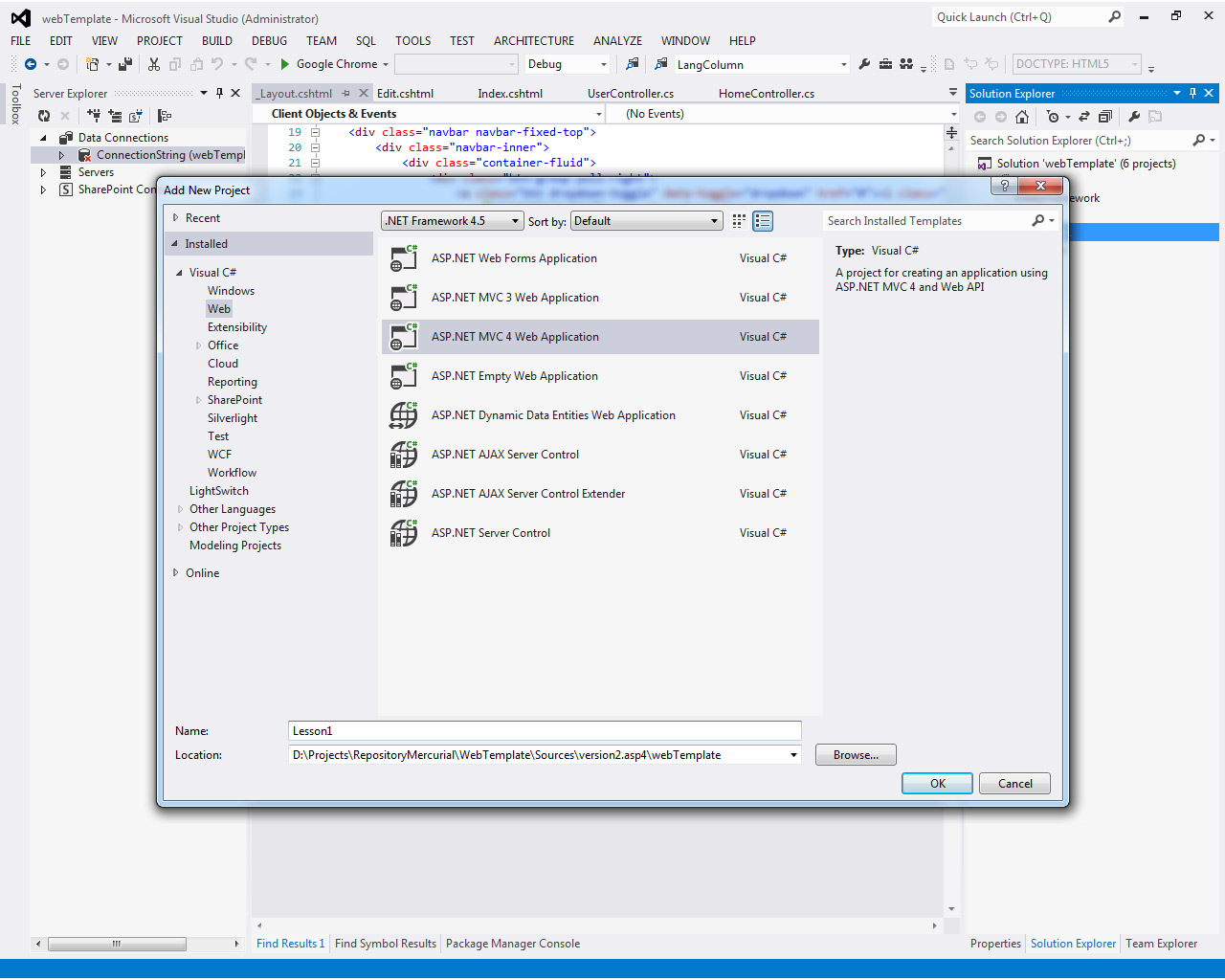
# ASP.NET MVC

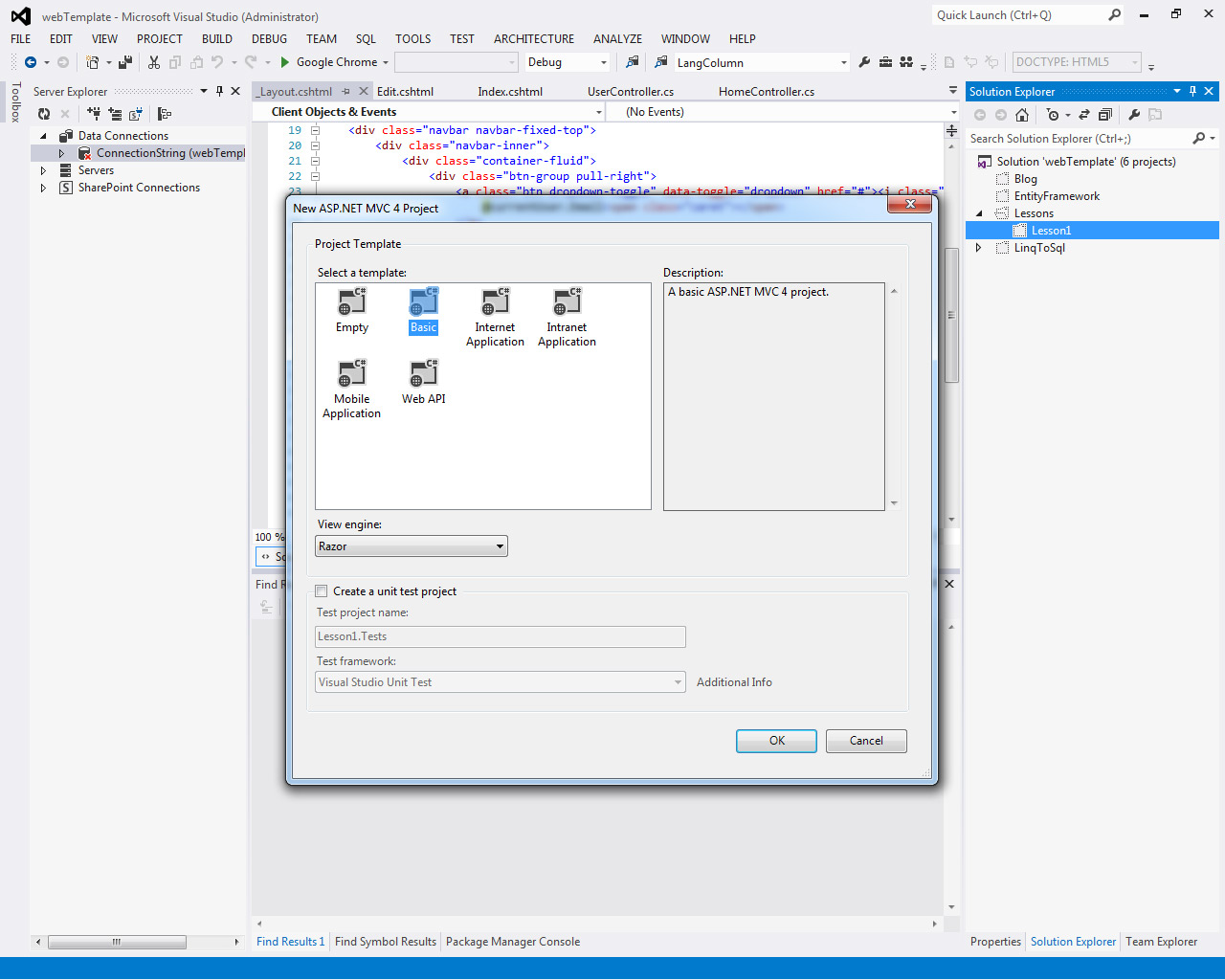
## Урок 1

**Цель урока:** Изучить Global.asax и поведение запуска веб-приложения, обработки веб-запроса. Изучение Nuget и Подключение протоколирования.

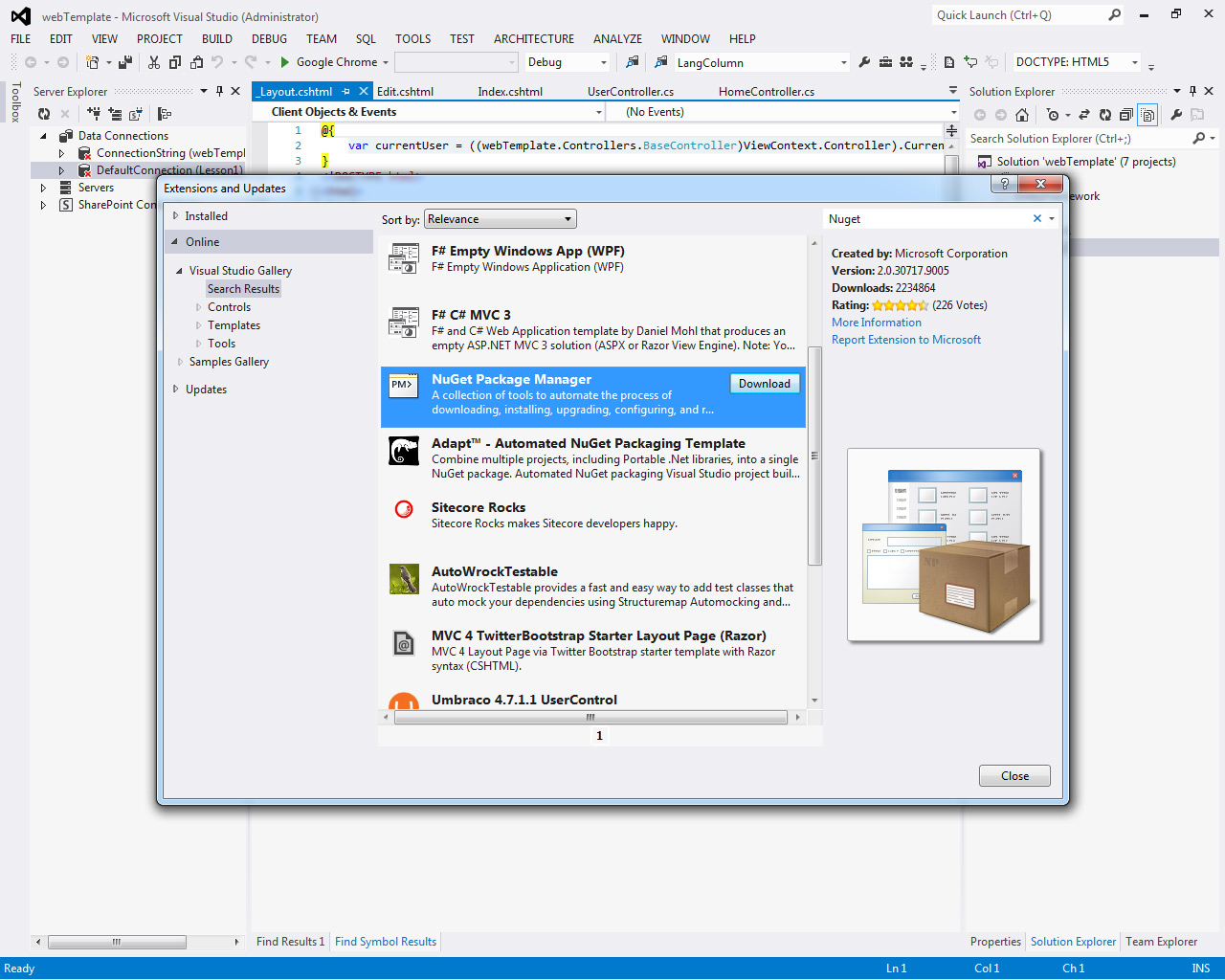
### Начало

Создадим приложение ASP.NET MVC 4 Web Application «Lesson1».

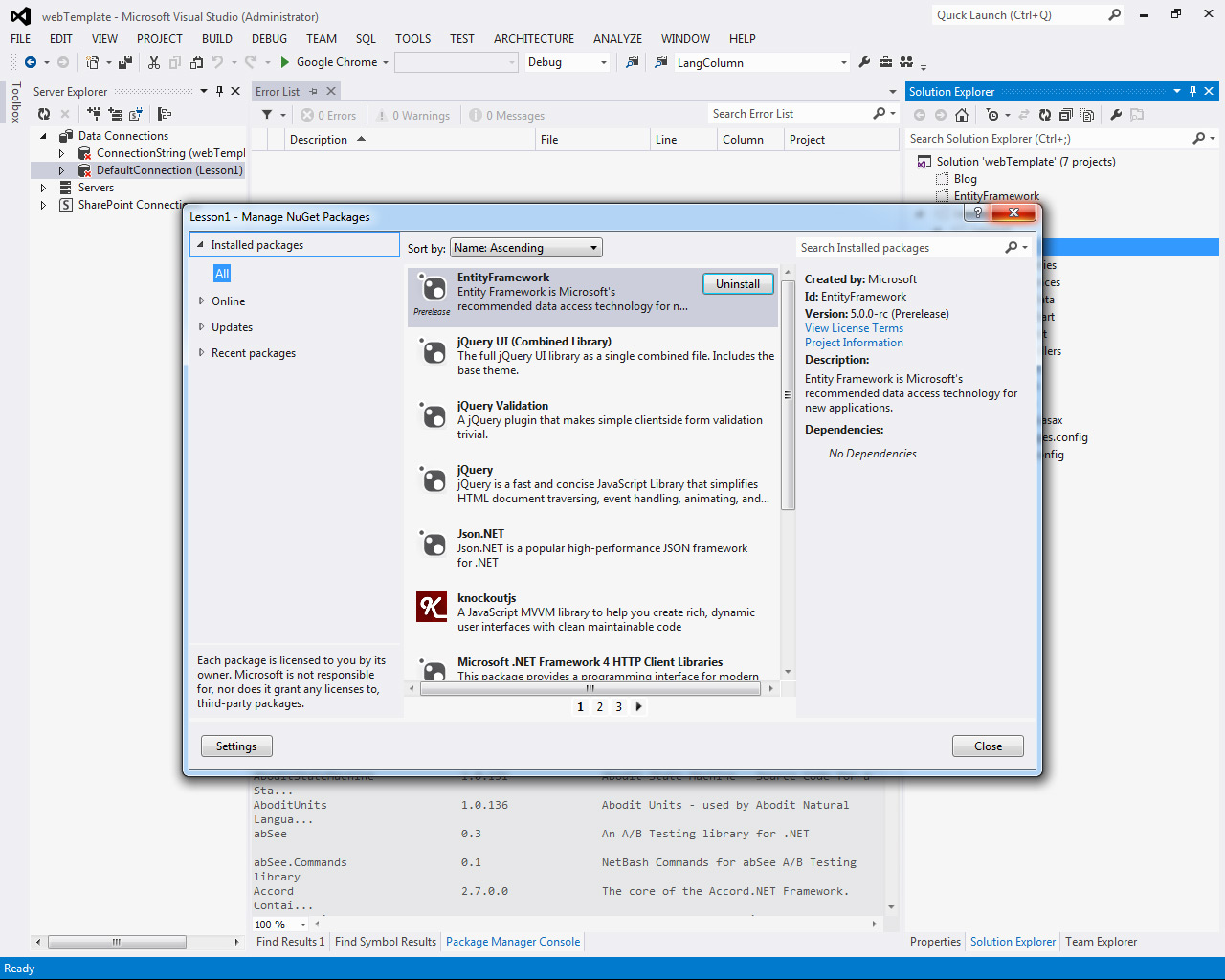




Не будем запускать приложение, а сразу установим (если до этого не сделали) NuGet расширение.



NuGet Package Manager – это расширение для Visual Studio, которое позволяет добавлять в существующие проекты модули, которые значительно упрощают работу. При создании «Basic» asp.net mvc 4 приложения в само приложение было добавлено изначально много модулей. Их список мы можем найти, кликнув в Manage NuGet Packages… в контекстном меню проекта

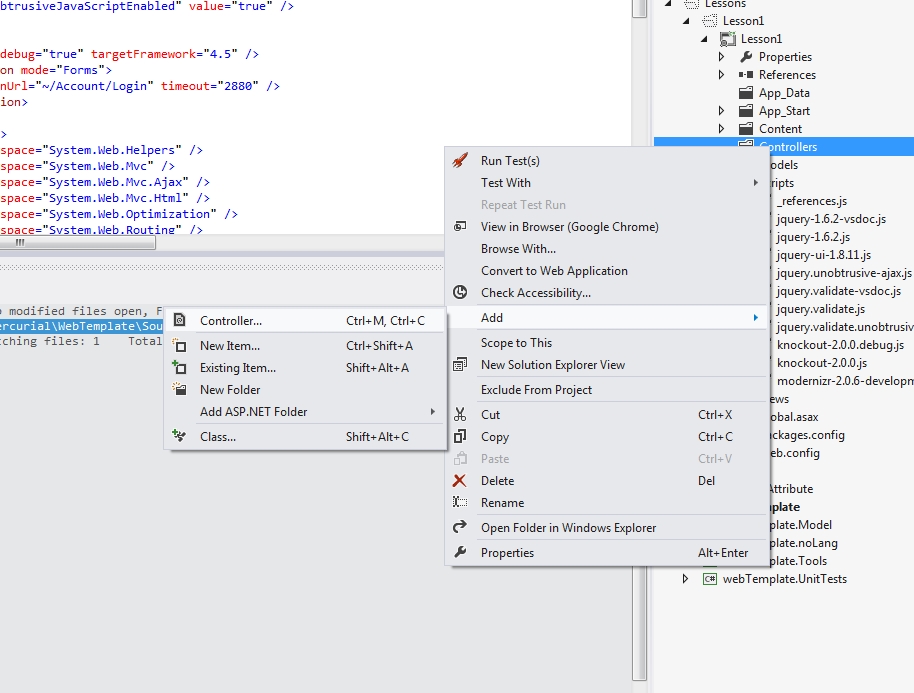


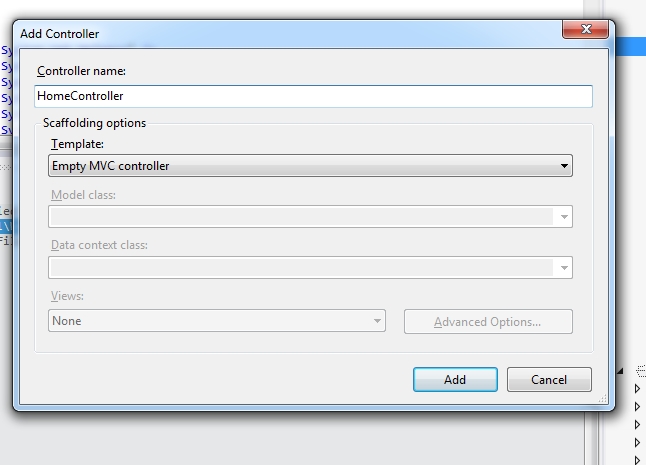
Из них по порядку:

* **Entity Framework** – обеспечивает работу с БД
* **jQuery (+ jQuery UI, jQuery Validation)** – популярный javascript framework (о нем потом еще пойдет речь).
* **Json.NET** – классы для работы с json-форматом данных.
* **knockoutjs** - не популярная javascript библиотека для работы с  Model View ViewModel архитектурой. (<http://knockoutjs.com/>)
* **Microsoft.Net Framework 4 Http Client Libraries** – программный интерфейс для работы с HttpContextом.
* **Microsoft ASP.NET MVC 4** – собственно классы паттерна проектирования MVC (то что есть основа всего этого)
* **Microsoft ASP.NET Razor 2** – view-движок. Есть еще ASPX и Spark – мы о них тоже будем говорить потом
* **Microsoft ASP.NET Universal Providers Core Libraries (Microsoft ASP.NET Universal Providers for LocalDB)**– предоставляет услуги для поддержки всех SQL Server 2005 (и более поздних) и для SQL Azure.
* **Microsoft ASP.NET Web API (Microsoft ASP.NET Web API Client Libraries, Microsoft ASP.NET Web API Core Libraries, Microsoft ASP.NET Web API Web Host)** – для создания REST приложений, работа с XML, JSON и всё такое
* **Microsoft ASP.NET Web Optimization Framework** – оптимизирует передачу данных, например минимизирует js-код
* **Microsoft ASP.NET WebPages 2** – набор классов для работы во View
* **Microsoft jQuery Unobtrusive Ajax (Microsoft jQuery Unobtrusive Validation)–** jQuery библиотека для поддержки ненавящевого ajax/валидации
* **Microsoft.Web.Infrastructure** – позволяет динамически регистрировать HTTP модули во время выполнения
* **Modernizr** – js-библиотека которая позволяет использовать html5 и css3 на старых браузерах
* **WebGrease** – позволяет минифицировать html, css, js.

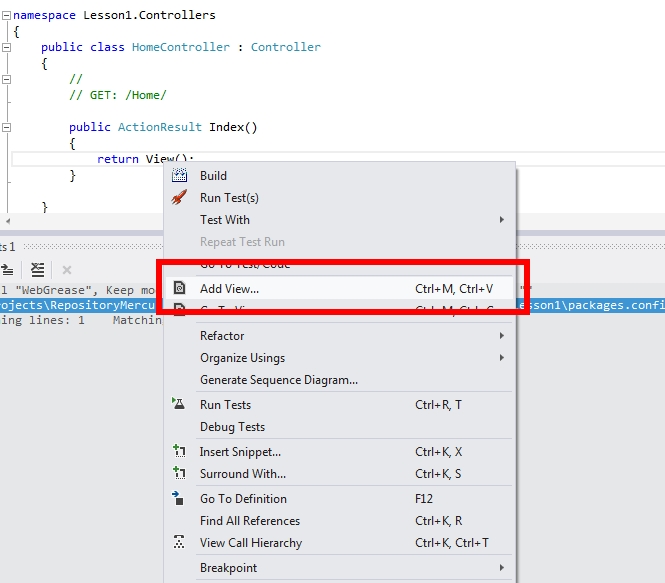
Ок, теперь когда мы так примерно прикинули из чего состоит наше приложение запустим его.

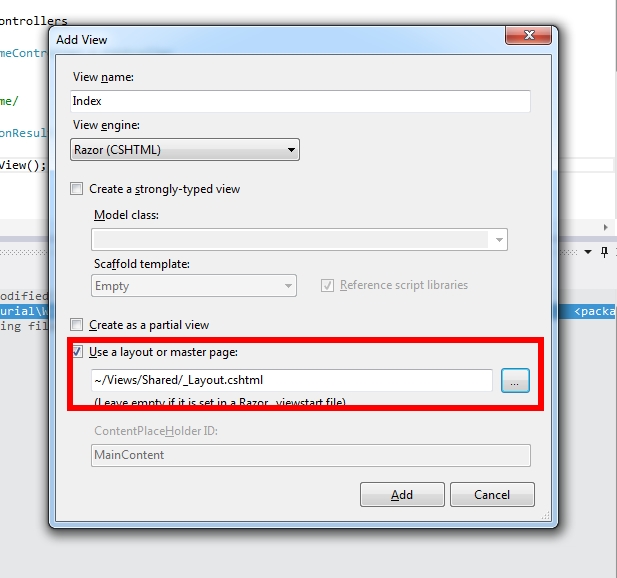
Для этого необходимо создать HomeController:





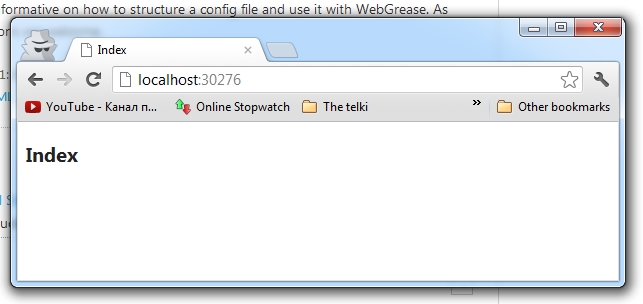
Для метода (действия) Index добавим View и выберем ~Views/Shared/\_Layout.cshtml в качестве layout (типа master page):





Собственно можно запускать.

Всё что мы увидим – это:



Почему контроллер надо было назвать именно Home, как это работает, более подробно мы будем изучать в следующих уроках.

### Global.asax

А счас мы обратим внимание на файл Global.asax:

public class MvcApplication : System.Web.HttpApplication

{

protected void Application\_Start()

{

AreaRegistration.RegisterAllAreas();

FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);

RouteConfig.RegisterRoutes(RouteTable.Routes);

BundleConfig.RegisterBundles(BundleTable.Bundles);

}

}

Класс MvcApplication наследует HttpApplication и содержит инструкции для инициализации приложения. Есть еще несколько событий, на которые можно добавить код. Рассмотрим их:

* **public void Init()** –приложение инициализируется или при первом вызове. Оно вызывается для всех экземпляров объекта HttpApplication.
* **public void Dispose()** – вызывается непосредственно перед уничтожением объекта HttpApplication. Это идеальное место для очистки ранее используемых ресурсов.
* **Application\_Error** – наступает, когда необработанное исключение случается в приложении.
* **Application\_Start** – наступает, когда первый экземпляр HttpApplication создается. Это позволяет создавать объекты доступные для всех экземпляров класса HttpApplication.
* **Application\_End** – наступает когда все созданные ранее экземпляры класса HttpApplication уничтожены. Это событие наступает только однажды в течении всего времени жизни приложения.
* **Application\_BeginRequest** – наступает когда приложение получает запрос. Первый раз это событие наступает для запроса страницы, когда пользователь вводит URL.
* **Application\_EndRequest** – Последнее событие которое наступает для запроса к приложению.
* **Application\_PreRequestHandlerExecute** – наступает прежде, чем ASP.NET запустит обработчик страницы или веб-службу.
* **Application\_PostRequestHandlerExecute** – наступает когда ASP.NET заканчивает обработку.
* **Applcation\_PreSendRequestHeaders** – наступает перед тем как ASP.NET посылает клиенту (браузеру) HTTP заголовки.
* **Application\_PreSendContent** – наступает перед тем как ASP.NET посылает клиенту (браузеру) HTTP содержимое.
* **Application\_AcquireRequestState** – наступает когда ASP.NET получает текущее состояние (состояние сессии) связанное с текущим запросом.
* **Application\_ReleaseRequestState** – наступает, когда ASP.NET завершает исполнение всех событий. В результате все модули сохраняют свои текущие состояния.
* **Application\_ResolveRequestCache** – наступает, когда ASP.NET выполняет запрос авторизации. Это позволяет модулям кеширования обработать запрос и обслужить из кэша, минуя обработчик выполнения.
* **Application\_UpdateRequestCache** – наступает, когда ASP.NET завершает выполнение обработчика, чтобы модули кеширования могли сохранить результат для использования в последующих ответах.
* **Application\_AuthenticateRequest** – наступает, когда модуль идентификации устанавливает личность текущего пользователя как действительные. На данный момент учетные данные пользователя уже проверены.
* **Application\_AuthorizeRequest** – наступает, когда модуль авторизации подтверждает, что пользователь может иметь доступ к ресурсам.
* **Session\_Start** – наступает, когда новый пользователь заходит на сайт.
* **Session\_End** – наступает, когда истекает время сессии пользователя, или он покидает сайт.

Ок, теперь чтобы воочию убедиться, что всё именно так и происходит, добавим протоколирование и сделаем это через добавление NLog модуля в NuGet

### Package Manager Console

В NuGet есть консоль для выполнения команд по установке\удалению\поиску модулей, и других вещей, типа скаффолдинга.

Для вывода всех установленных модулей пишем:

Get-Package

Для получения всех доступных к установке модулей:

Get-Package –ListAvailable

Для получения всех доступных модулей с названием NLog

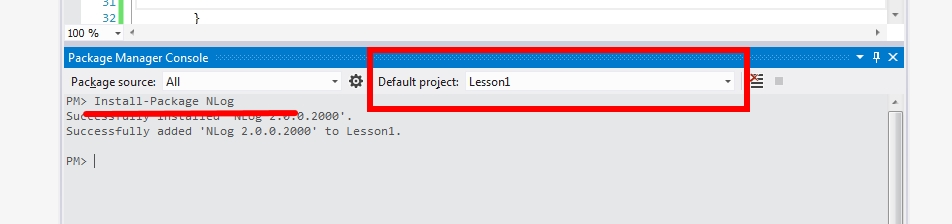
Get-Package -ListAvailable -Filter NLog

Или

Get-Package -ListAvailable | where {$\_.Id -match "NLog"} (это дольше)

Для установки модуля NLog необходимо вначале выбрать проект (если их в солюшене больше одного) и ввести команду:

Install-Package NLog



Файлы копируются в проект, добавляются ссылки на сборки и web.config может быть обновлен.

Для удаления из проекта модуля необходимо, чтобы он не был связан с другими модулями. Удаляем так:

Uninstall-Package NLog

### NLog

После установки пользуемся документацией на NLog (<http://nlog-project.org/wiki/Tutorial>) – и добавляем в web.config:

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

<targets>

<target name="logfile" xsi:type="File" fileName="C://file.txt" />

</targets>

<rules>

<logger name="\*" minlevel="Info" writeTo="logfile" />

</rules>

</nlog>

Мы ее потом исправим, а пока проверим как есть.

Добавим в код:

protected void Application\_Start()

{

**logger.Info("Application Start");**

AreaRegistration.RegisterAllAreas();

FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);

RouteConfig.RegisterRoutes(RouteTable.Routes);

BundleConfig.RegisterBundles(BundleTable.Bundles);

}

**public void Init()**

**{**

**logger.Info("Application Init");**

**}**

**public void Dispose()**

**{**

**logger.Info("Application Dispose");**

**}**

**protected void Application\_Error()**

**{**

**logger.Info("Application Error");**

**}**

**protected void Application\_End()**

**{**

**logger.Info("Application End");**

**}**

**...**

Запустим и откроем файл C://file.txt, завершим приложение (Stop). Мы увидим какие события наступили:

2012-09-18 19:18:11.5668|INFO|Lesson1.MvcApplication|Application Start

2012-09-18 19:18:13.7319|INFO|Lesson1.MvcApplication|Application Init

2012-09-18 19:18:14.2709|INFO|Lesson1.MvcApplication|Application Init

2012-09-18 19:18:14.2769|INFO|Lesson1.MvcApplication|Application BeginRequest

2012-09-18 19:18:14.3579|INFO|Lesson1.MvcApplication|Application AuthenticateRequest

2012-09-18 19:18:14.3579|INFO|Lesson1.MvcApplication|Application AuthorizeRequest

2012-09-18 19:18:14.3579|INFO|Lesson1.MvcApplication|Application ResolveRequestCache

2012-09-18 19:18:14.3989|INFO|Lesson1.MvcApplication|Session Start

2012-09-18 19:18:14.3989|INFO|Lesson1.MvcApplication|Application AcquireRequestState

2012-09-18 19:18:14.3989|INFO|Lesson1.MvcApplication|Application PreRequestHandlerExecute

2012-09-18 19:18:15.9580|INFO|Lesson1.MvcApplication|Application PreRequestHandlerExecute

2012-09-18 19:18:15.9580|INFO|Lesson1.MvcApplication|Application ReleaseRequestState

2012-09-18 19:18:15.9580|INFO|Lesson1.MvcApplication|Application UpdateRequestCache

2012-09-18 19:18:15.9580|INFO|Lesson1.MvcApplication|Application EndRequest

2012-09-18 19:18:15.9580|INFO|Lesson1.MvcApplication|Application PreSendRequestHeaders

2012-09-18 19:18:35.6061|INFO|Lesson1.MvcApplication|Session End

2012-09-18 19:18:38.0833|INFO|Lesson1.MvcApplication|Application Dispose

2012-09-18 19:18:38.0833|INFO|Lesson1.MvcApplication|Application End

2012-09-18 19:18:39.1383|INFO|Lesson1.MvcApplication|Application Dispose

В Application\_Start выполняется 4 строки регистрации Area (область), Filter (фильтров), Bundle (пучок), Route (маршрутов).

Подробности по инициализации Filter, Вundle и Route находятся в папке App\_Start.

### WebActivator

WebActivator – это модуль, который позволяет запустить код до самого первого старта App\_Start. Это может быть необходимо для того, чтобы, например, создать тестовую БД перед запуском.

Установим:

Install-Package WebActivator

Добавим класс в App\_Start папку:

[assembly: WebActivator.PreApplicationStartMethod(typeof(PreStartApp), "Start")]

namespace Lesson1.App\_Start

{

public static class PreStartApp

{

private static NLog.Logger logger = NLog.LogManager.GetCurrentClassLogger();

/// <summary>

/// Метод запускается один раз перед стартом приложения

/// </summary>

public static void Start()

{

logger.Info("Application PreStart");

}

}

}

В файле логов увидим что строка Application PreStart исполняется раньше Application Start:

2012-09-19 10:29:01.3950|INFO|Lesson1.App\_Start.PreStartApp|Application PreStart

2012-09-19 10:29:01.6290|INFO|Lesson1.MvcApplication|Application Start

Напоследок сделаем запись логов более осмысленными в файлы разложенные по разным категориям и пишущие в папку Contents а не в корень C:

Создадим 4 файла отдельно для trace (трассировки), debug (отладки), info (информации), error (ошибки). Определим место записи:  **/Contents/logs/[текущая дата]** Перепишем конфигурацию:

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

<variable name="logDirectory" value="${basedir}/Content/logs/${shortdate}" />

<targets>

<target name="fileLogTrace" xsi:type="File" fileName="${logDirectory}/trace.txt" />

<target name="fileLogDebug" xsi:type="File" fileName="${logDirectory}/debug.txt" />

<target name="fileLogInfo" xsi:type="File" fileName="${logDirectory}/info.txt" />

<target name="fileLogErrors" xsi:type="File" fileName="${logDirectory}/errors.txt" />

</targets>

<rules>

<logger name="\*" level="Trace" writeTo="fileLogTrace" />

<logger name="\*" level="Debug" writeTo="fileLogDebug" />

<logger name="\*" level="Info" writeTo="fileLogInfo" />

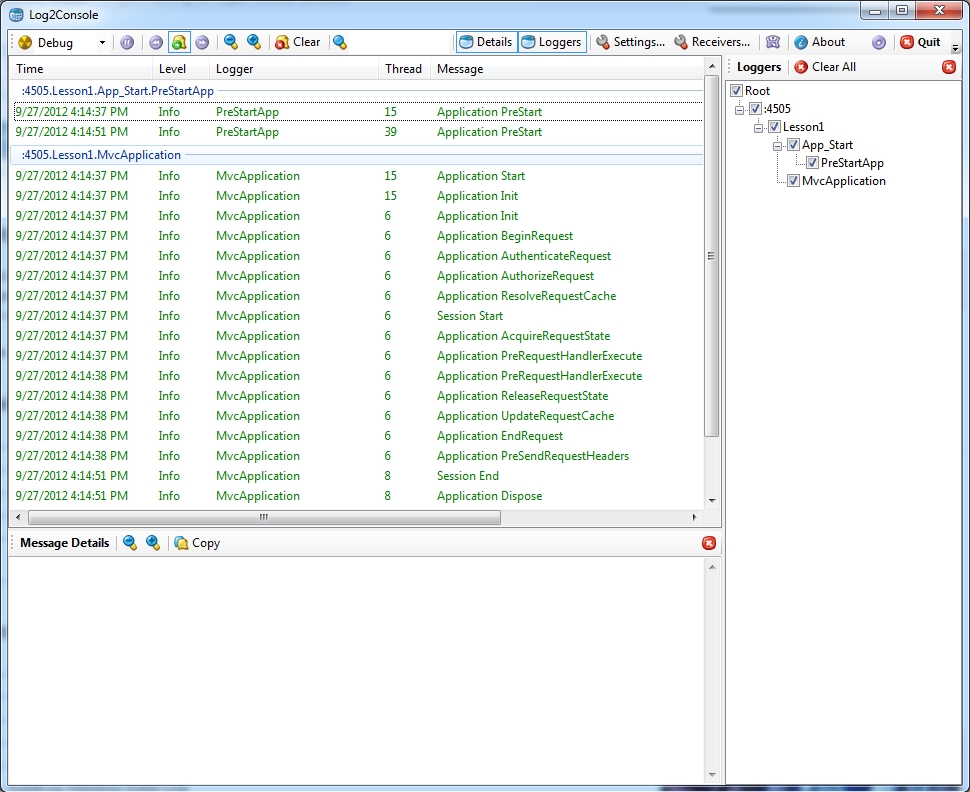
<logger name="\*" minlevel="Warn" writeTo="fileLogErrors" />

</rules>

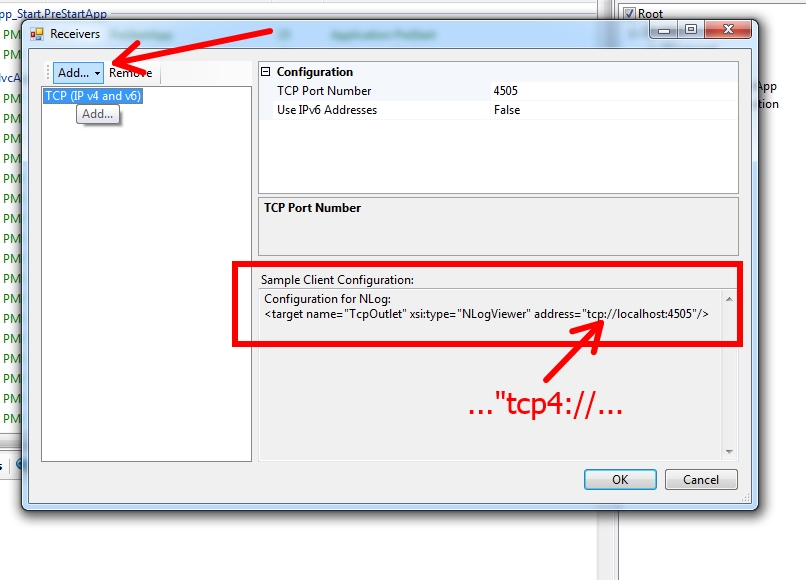
</nlog>

### Log2Console

Для NLog есть еще классная программа Log2Console, которая позволяет получать логи прямо в окно программы.



Запускаем программу и настраиваем приемщик:



В Web.config пишем:

<target name="TcpOutlet" xsi:type="NLogViewer" address="tcp4://localhost:4505"/>

Обращаю внимание, что писать надо **address=”tcp4://…”,** а не **address=”tcp://…”**

## Урок 2

Цель урока: Изучение DI (Dependency Injection). Пример на Ninject и Unity (Autofac, Winsor).

Во многих случаях один и тот же экземпляр типа используется в вашем приложении. Распространенный способ для достижения этой цели является реализация шаблона Одиночка (Singleton).

Проблема в том, что вы в основном должны заблокировать ваш тип. Поступая так, ваш код в конечном итоге будет жестким и негибким. В частности, это будет очень трудно для модульного тестирования кода, который реализован таким образом.

DI контейнер позволяет использовать объекты проинициализированного класса без единого объявления создания объекта в коде.

Первая задача, которую решает DI – это передача за ответственность доступа к сервису от объекта, к «сервису-локатору», который выдает ссылку к сервису.

Например, до использования сервиса:

Опишем пару классов, интерфейс IWeapon с методом Kill, два класса реализации Bazuka и Sword, и класс Warrior, который пользуется оружием:

public interface IWeapon

{

void Kill();

}

public class Bazuka : IWeapon

{

public void Kill()

{

Console.WriteLine("BIG BADABUM!");

}

}

public class Sword : IWeapon

{

public void Kill()

{

Console.WriteLine("Chuk-chuck");

}

}

public class Warrior

{

readonly IWeapon Weapon;

public Warrior(IWeapon weapon)

{

this.Weapon = weapon;

}

public void Kill()

{

Weapon.Kill();

}

}

Используем это:

class Program

{

static void Main(string[] args)

{

Warrior warrior = new Warrior(new Bazuka());

warrior.Kill();

Console.ReadLine();

}

}

Читаем между строк. Создаем воина и даем ему базуку, он идет и убивает. В консоли получаем:

BIG BADABUM!

Заметим, что у нас нету проверки на null в строке

Weapon.Kill();

Что тут плохо? Воин не знает, есть ли у него оружие, и выдачей оружия занимается не отдельный модуль, а главная программа.

Суть DI – поручить выдачу оружия другому модулю.

Подключаем Ninject:

Install-Package Ninject

Создаем модуль, который занимается выдачей оружия:

public class WeaponNinjectModule : NinjectModule

{

public override void Load()

{

this.Bind<IWeapon>().To<Sword>();

}

}

Что буквально значит: «если попросят оружие – то выдайте мечи».

Создаем «сервис-локатор» и пользуемся оружием:

class Program

{

public static IKernel AppKernel;

static void Main(string[] args)

{

AppKernel = new StandardKernel(new WeaponNinjectModule());

var warrior = AppKernel.Get<Warrior>();

warrior.Kill();

Console.ReadLine();

}

}

Как видно объект warrior мы создаем не с помощью конструкции new, а через AppKernel.Get<>(). При создании AppKernel, мы передаем в качестве конструктора модуль, отвечающий за выдачу оружия (в данном случае это меч). Любой объект, который мы пытаемся получить через AppKernel.Get, будет (по мере возможности) проинициализирован, если существуют модули, которые знают, как это делать.

Другой момент применения, когда объект Warrior не берет с собой оружие каждый раз, а при не обнаружении оного обращается к сервису локатору и получает его:

public class OtherWarrior

{

private IWeapon \_weapon;

public IWeapon Weapon

{

get

{

if (\_weapon == null)

{

\_weapon = Program.AppKernel.Get<IWeapon>();

}

return \_weapon;

}

}

public void Kill()

{

Weapon.Kill();

}

}

Исполняем:

var otherWarrior = new OtherWarrior();

otherWarrior.Kill();

Наш воин получает оружие по прямым поставкам – супер!

Ок, в Ninject есть еще одна очень хорошая деталь. Если свойство (public property) помечено [Inject] – то при создании класса через AppKernel.Get<>() –поле инициализуется сервисом-локатором:

public class AnotherWarrior

{

[Inject]

public IWeapon Weapon { get; set; }

public void Kill()

{

Weapon.Kill();

}

}

var anotherWarrior = AppKernel.Get<AnotherWarrior>();

anotherWarrior.Kill();

### Unity

Абсолютно всё то же:

* Установка

Install-Package Unity

* Инициализация сервиса локатора (Container)

Container = new UnityContainer();

* Регистрация типа

Container.RegisterType(typeof(IWeapon), typeof(Bazuka));

* Получение объекта и использование:

var warrior = Container.Resolve<Warrior>();

warrior.Kill();

* Кроме того у Unity есть класс-одиночка (Singleton) ServiceLocator, который регистрирует контейнер и позволяет получить доступ к сервисам из любого места.

var serviceProvider = new UnityServiceLocator(Container);

ServiceLocator.SetLocatorProvider(() => serviceProvider);

* Хитрый OtherWarrior теперь так получает оружие:

public class OtherWarrior

{

private IWeapon \_weapon;

public IWeapon Weapon

{

get

{

if (\_weapon == null)

{

\_weapon = ServiceLocator.Current.GetInstance<IWeapon>();

}

return \_weapon;

}

}

public void Kill()

{

Weapon.Kill();

}

}

### Autofac

Такая же собственно всё и происходит:

* Установка

Install-Package Autofac

* Инициализация строителя сервиса-локатора (ContainerBuilder) – нет-нет, это еще не сам контейнер, это как модули

var builder = new ContainerBuilder();

* Регистрация типов, надо зарегистрировать всех, потому что самостоятельная попытка вызова конструктора без параметров неизвестных типов тут не учитывается

builder.RegisterType<Bazuka>();

builder.RegisterType<Warrior>();

builder.Register<IWeapon>(x => x.Resolve<Bazuka>());

* Создание сервиса локатора (Container)

var container = builder.Build();

* Получение объекта и использование:

var warrior = container.Resolve<Warrior>();

warrior.Kill();

### Castle Windsor

* Установка

Install-Package Castle.Windsor

* Инициализация сервиса-локатора

var container = new WindsorContainer();

* Регистрация типов, надо зарегистрировать всех, потому что самостоятельная попытка вызова конструктора без параметров неизвестных типов тут не учитывается

container.Register(Component.For<IWeapon>().ImplementedBy<Bazuka>(),

Component.For<Warrior>().ImplementedBy<Warrior>());

* Получение объекта и использование:

var warrior = container.Resolve<Warrior>();

warrior.Kill();

### Маленький подытог

На самом деле реализации Dependency Injection не сильно, но всё же отличаются. Некоторые поддерживают инициализацию в Web.config (App.config) файлах. Некоторые задают правила для инициализации, как мы сейчас посмотрим на расширении Ninject для asp.net mvc – это касается инициализации сервиса-локатора как генератора общих объектов, так и отдельно для каждой сессии.

### Объекты областей (Ninject)

В Ninject можно задать несколько способов инициализации получения объекта из класса. Это требуется для того, что если мы работаем в различных контекстах (например, в разных потоках (Thread)), то объекты должны быть использованы разные. Тем самым поддерживается масштабируемость и гибкость приложения

|  |  |  |
| --- | --- | --- |
| Область | Метод связывания | Объяснение |
| Временный | .InTransientScope() | Объект класса будет создаваться по каждой просьбе. (метод по умолчанию) |
| Одиночка | .InSingletonScope() | Объект класса будет создан однажды и возвращаться будет он по каждому запросу. |
| Поток | .InThreadScope() | Один объект на поток |
| Запрос | .InRequestScope() | Один объект будет создан на каждый web-запрос |

Для нашего web приложения мы будем использовать InRequestScope() для работы с репозиторием (БД).

### Lifetime Manager в Unity

В Unity для задачи правил инициализации используется реализация абстрактного класса LifetimeManager.

Происходит это таким образом:

\_container.RegisterType<DbContext, SavecashTravelContext>(new PerRequestLifetimeManager());

Где PerRequestLifetimeManager – это реализация LifetimeManager:

public class PerRequestLifetimeManager : LifetimeManager

{

/// <summary>

/// Key to store data

/// </summary>

private readonly string \_key = String.Format("SingletonPerRequest{0}", Guid.NewGuid());

/// <summary>

/// Retrieve a value from the backing store associated with this Lifetime policy.

/// </summary>

/// <returns>

/// the object desired, or null if no such object is currently stored.

/// </returns>

public override object GetValue()

{

if (HttpContext.Current != null && HttpContext.Current.Items.Contains(\_key))

return HttpContext.Current.Items[\_key];

return null;

}

/// <summary>

/// Stores the given value into backing store for retrieval later.

/// </summary>

/// <param name="newValue">The object being stored.</param>

public override void SetValue(object newValue)

{

if (HttpContext.Current != null)

HttpContext.Current.Items[\_key] = newValue;

}

/// <summary>

/// Remove the given object from backing store.

/// </summary>

public override void RemoveValue()

{

if (HttpContext.Current != null && HttpContext.Current.Items.Contains(\_key))

HttpContext.Current.Items.Remove(\_key);

}

}

Суть заключается в том, что все объект сохраняется в HttpContext.Current.Items[\_key] и выдается только, если находится в том же контексте HttpContext.Current, иначе создается новый объект. Если HttpContext.Current в области кода не существует (используем такой LifetimeManager в консольном приложении или в отдельном потоке) – то данный контейнер не будет работать.

### Использование Ninject в asp.net mvc

Устанавливаем Ninject в среду asp.net mvc. Отдельно создаем свой проект LessonProject, создадим там HomeController с методом и view Index.

public class HomeController : Controller

{

public ActionResult Index()

{

return View();

}

}

И

@{

ViewBag.Title = "LessonProject";

Layout = "~/Views/Shared/\_Layout.cshtml";

}

<h2>LessonProject</h2>

Запустим – работает.

Примечание: В дальнейшем мы будем переносить этот проект в последующие уроки. Так что он так называется (по общему).

Теперь установим модуль Ninject и Ninject.MVC3 для этого проекта.

Install-Package Ninject.MVC3

Добавляется класс в папку App\_Start:

[assembly: WebActivator.PreApplicationStartMethod(typeof(LessonProject.App\_Start.NinjectWebCommon), "Start")]

[assembly: WebActivator.ApplicationShutdownMethodAttribute(typeof(LessonProject.App\_Start.NinjectWebCommon), "Stop")]

namespace LessonProject.App\_Start

{

using System;

using System.Web;

using Microsoft.Web.Infrastructure.DynamicModuleHelper;

using Ninject;

using Ninject.Web.Common;

public static class NinjectWebCommon

{

private static readonly Bootstrapper bootstrapper = new Bootstrapper();

/// <summary>

/// Starts the application

/// </summary>

public static void Start()

{

DynamicModuleUtility.RegisterModule(typeof(OnePerRequestHttpModule));

DynamicModuleUtility.RegisterModule(typeof(NinjectHttpModule));

bootstrapper.Initialize(CreateKernel);

}

/// <summary>

/// Stops the application.

/// </summary>

public static void Stop()

{

bootstrapper.ShutDown();

}

/// <summary>

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

/// </summary>

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

private static IKernel CreateKernel()

{

var kernel = new StandardKernel();

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

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

RegisterServices(kernel);

return kernel;

}

/// <summary>

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

/// </summary>

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

private static void RegisterServices(IKernel kernel)

{

}

}

}

В RegisterServices мы добавим инициализации своих сервисов. Для начала добавим шутливый IWeapon, а в дальнейшем еще будем возвращаться к этому методу для регистрации других сервисов:

public interface IWeapon

{

string Kill();

}

…

public class Bazuka : IWeapon

{

public string Kill()

{

return "BIG BADABUM!";

}

}

…

private static void RegisterServices(IKernel kernel)

{

kernel.Bind<IWeapon>().To<Bazuka>();

}

В контроллере используем атрибут [Inject]:

public class HomeController : Controller

{

[Inject]

public IWeapon weapon { get; set; }

public ActionResult Index()

{

return View(weapon);

}

}

Изменяем View:

@model LessonProject.Models.IWeapon

@{

ViewBag.Title = "LessonProject";

Layout = "~/Views/Shared/\_Layout.cshtml";

}

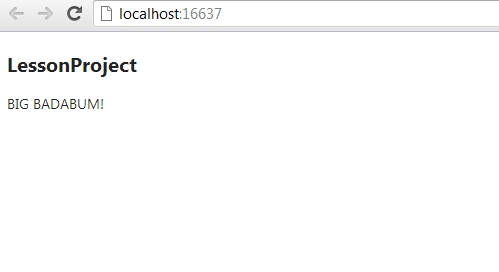
<h2>LessonProject</h2>

<p>

@Model.Kill()

</p>

На выходе получаем:



Ninject пользуясь WebActivator регистрирует свои модули OnePerRequestHttpModule и NinjectHttpModule, а также создает StandartKernel, далее мы можем проинициализировать свои сервисы.

### DependencyResolver

В asp.net mvc 3 появился класс DependencyResolver, который оборачивает DI-контейнеры (но только если сами DI-контейнеры об этом беспокоятся) и его можно использовать как сервис-локатор. Перепишем HomeController:

public class HomeController : Controller

{

private IWeapon weapon { get; set; }

public HomeController()

{

weapon = DependencyResolver.Current.GetService<IWeapon>();

}

public ActionResult Index()

{

return View(weapon);

}

}

Результат тот же.

### Итог:

Использование DI-контейнеров в современных приложениях необходимо, чтобы избавиться от сильной связности кода, и чтобы легко получать доступ из любой части кода к сервису. Также это необходимо для легкости написания Unit-тестов.

## Урок 3. Работа с БД

Цель урока: Изучить основные принципы работы с базой данных. Краткое описание реляционной модели баз данных. Работа с базой данных (создание таблиц, связей в VS 2012 [не забыть показать где стоит галочка]). Команды INSERT, UPDATE, DELETE, SELECT. Использование LinqToSql и Linq. Создание репозитария, IRepository, SqlRepository.

### Что такое БД

Реляционная база данных — [база данных](http://ru.wikipedia.org/wiki/%D0%91%D0%B0%D0%B7%D0%B0_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85), основанная на [реляционной модели данных](http://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D0%BB%D1%8F%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D0%BC%D0%BE%D0%B4%D0%B5%D0%BB%D1%8C_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85). Реляционность – это отношения (связи) от англ. relation.

#### Таблицы

Это таблица:

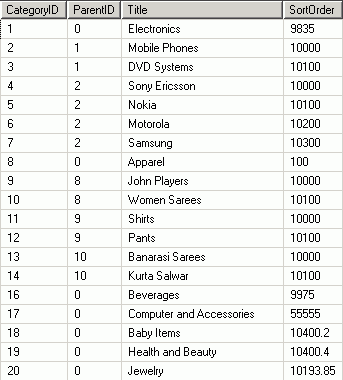


Таблица состоит из столбцов и строк. Столбцы имеют свойства – имя, тип данных.

Таблицы обладать следующими свойствами:

* у таблицы есть имя
* нет двух одинаковых строк
* столбцы имеют разные наименования (нет двух одинаковых столбцов)
* порядок строк в таблице произвольный (т.е. не надо учитывать порядок строк, если не задана сортировка)

Структуру таблицы можно записать в таком виде:

* Имя столбца
* Тип данных для этого столбца

#### Связи

Между таблицами существуют связи (relation). Для установки связи необходимо иметь следующее:

* Первичный ключ – это набор столбцов (атрибутов) таблицы, которые однозначно определяют уникальность строки. Обычно это одно поле, называется ID. Оно является автоикрементным, т.е. при попытке добавления записи там автоматически вставляется 1, 2, 3, 4… n+1, где n – это значение последнего добавленного ID.
* Внешний ключ – это набор столбцов (атрибутов) таблицы, которые однозначно определяют уникальность строки в другой таблице. Опять же это обычно одно поле, названное [Имя таблицы]ID. Но не является автоинкрементным.
* Прописана связь между первичным ключом и внешним ключом.

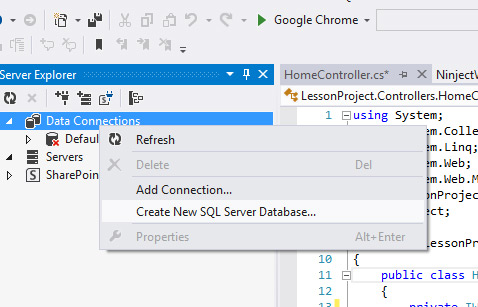
Связи бывают 3 типов:

* Один-к-одному. Т.е. одной строки в таблице соответствует одна строка в другой таблице. Это редко используется, но используется. Например, в одной таблице данные о пользователе, а в другой таблице дополнительные данные о том же пользователе. Такой вариант необходим, чтобы манипулировать по необходимости меньшим количеством данных.
* Один-ко-многим. Одной строке в таблице A соответствует одна или несколько строк в таблице B. Но одной строке в таблице B соответствует только одна строка в таблице A. В этом случае в таблице B существует внешний ключ, который однозначно определяет запись в таблице A.
* Многие-ко-многим. Одной строке в таблице А соответствует одна или несколько строк в таблице В, что истинно и в обратном случае. В данном случае создается дополнительная таблица со своим первичным ключом, и двумя внешними ключами к таблице A и B.

Сейчас разберемся, как это делать.

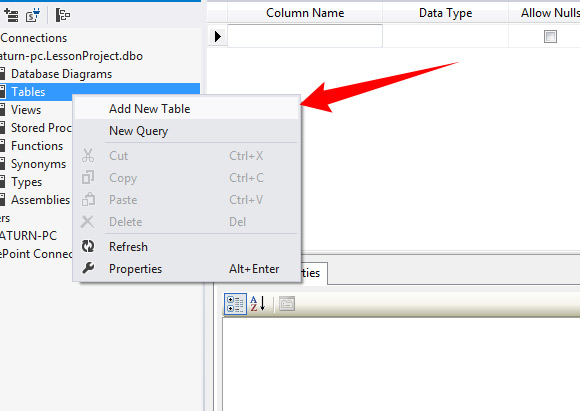
### Создание простой схемы в БД

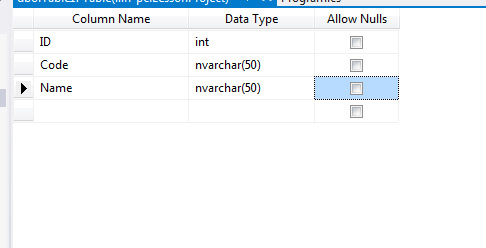
Создадим БД в VS 2012:



Назовем LessonProject, и добавим 3 таблицы Role User и UserRole.

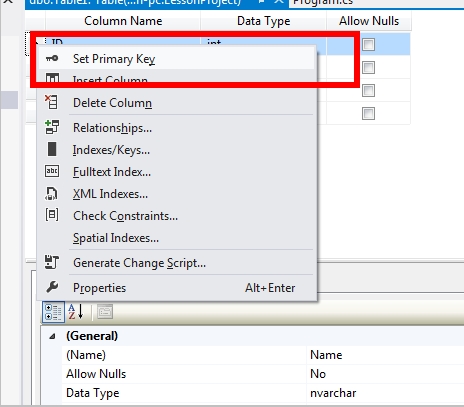
Создадим таблицу Role:



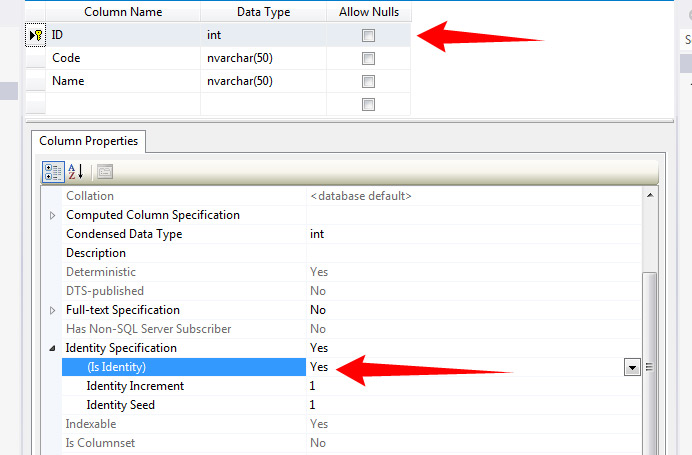


Для строковых значений используем тип nvarchar(n), где n – максимальная длина строки, обычно используется от 50 до 500. Для больших текстовых строк используется nvarchar(MAX).

Устанавливаем первичный ключ:



Задаем для ID автоинкремент:



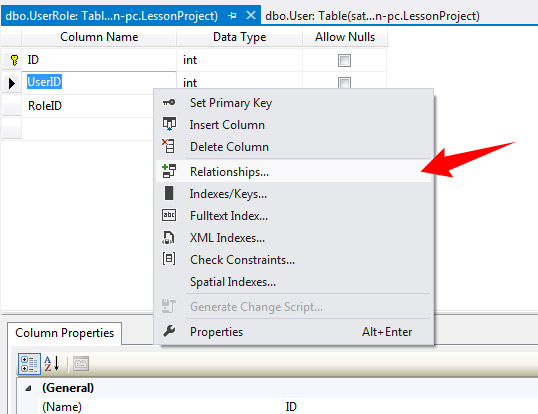
Подобным образом создаем таблицу User:

|  |  |
| --- | --- |
| **Поле** | **Тип поля** |
| ID | int |
| Email | nvarchar(150) |
| Password | nvarchar(50) |
| AddedDate | datetime |
| ActivatedDate | Datetime (null) |
| ActivatedLink | nvarchar(50) |
| LastVisitDate | Datetime (null) |
| AvatarPath | nvarchar(150) (null) |

Создаем таблицу UserRole:

|  |  |
| --- | --- |
| **Поле** | **Тип поля** |
| ID | int |
| UserID | int |
| RoleID | int |

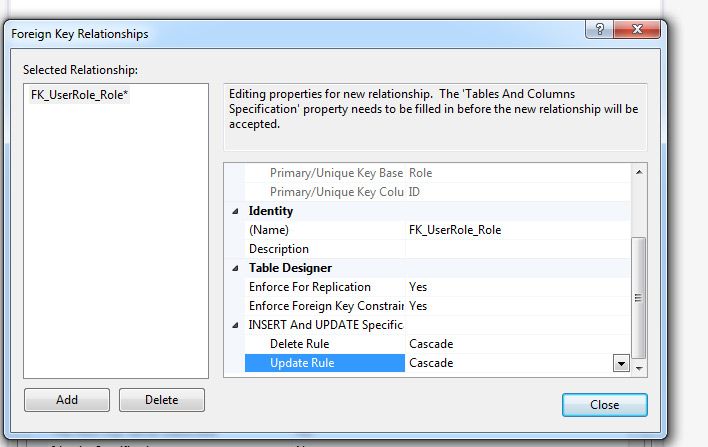
Добавим связи:



Добавляем новую связь, нажав Add, добавление связей происходит в таблице где находятся внешние ключи. Раскрываем Tables and Columns вкладку и выставляем таблицу с первичным ключом, и выбираем внешний ключ в текущей таблице UserRole.



В свойствах INSERT And UPDATE Specification выставляем On Update/On Delete свойства Cascade:



Это необходимо для того, что при изменении/удалении столбца из таблицы Role все связанные с этой строкой строки таблицы UserRole должны быть изменены или удалены. Тут касается больше удаления, так как первичный ключ в таблице обычно никогда не изменяют.

Аналогичную связь мы устанавливаем с таблицей User.

Таким образом, таблицы Role и User имеют отношения многие ко многим через таблиц UserRole. Т.е. у одного пользователя может быть больше одной роли, и одна и та же роль может быть у нескольких пользователей.

### Использование команд SELECT , INSERT, UPDATE, DELETE.

В реляционных базах данных используется язык запросов SQL.

Есть 4 основные команды для манипулирования данными, SELECT, INSERT, UPDATE, DELETE

SELECT – выбрать данные и таблицы(таблиц).

Пример:

SELECT \* FROM User

INSERT - Добавление строк в таблицу

Пример:

INSERT INTO Role (Code, Name)  
VALUES (“admin”, “Администратор”)

UPDATE – изменение значений в таблице

Пример:

UPDATE User  
SET Password=”password1”   
WHERE ID=1

DELETE – удаление строк из таблицы

Пример:

DELETE FROM User  
WHERE ID =1

*Примечание: Подробнее можно изучить SQL по ссылкам:*

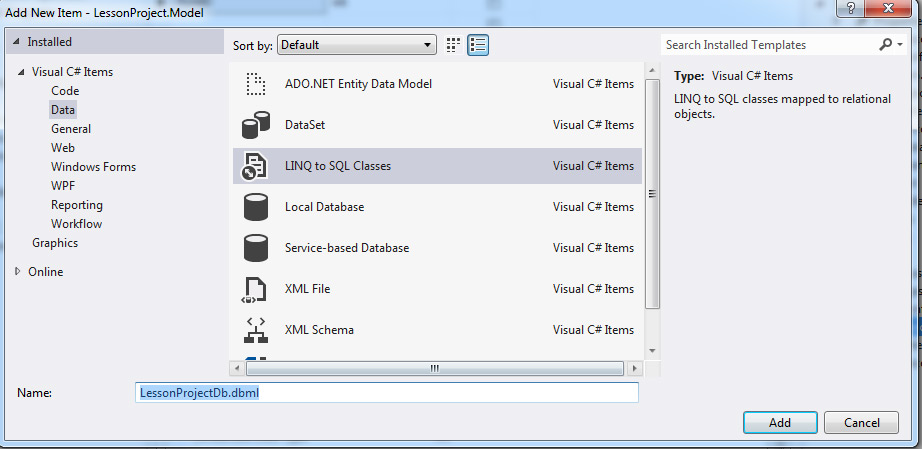
<http://www.w3schools.com/sql/>

<http://codingcraft.ru/sql_queries.php>

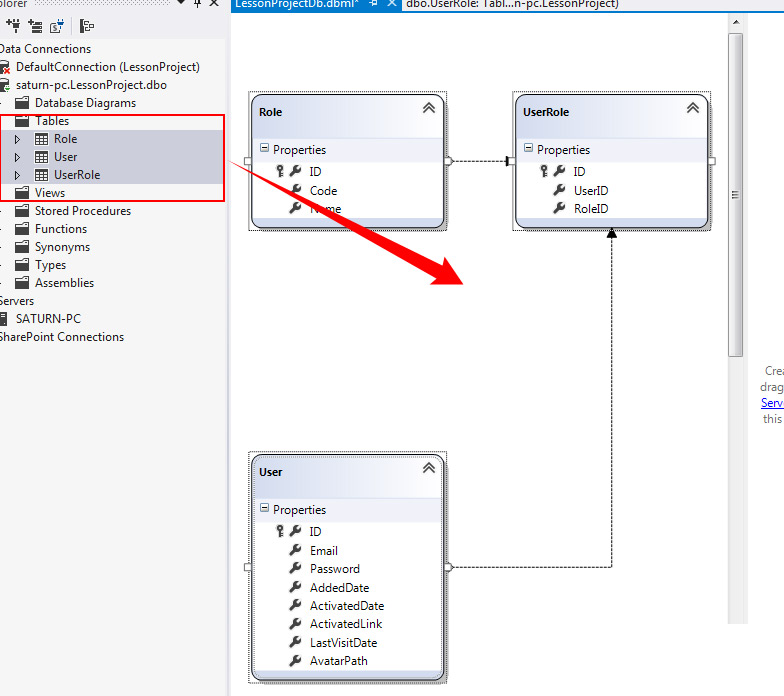
### LinqToSQL и Linq.

Создаем проект LessonProject.Model для работы с БД типа ClassLibrary.

Добавляем LINQ to SQL Classes тип, называем LessonProejctDb.dbml



Открываем объект, выделяем все таблицы и мышкой переносим на холст:



Собственно, с помощью таких простых действий мы получаем:

* классы готовые к использованию в работе с БД
* визуальное отображение таблиц и связей

Добавим несколько данных в таблицу Role и User:

|  |  |  |
| --- | --- | --- |
| 1 | admin | Админ |
| 2 | customer | Пользователь |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | chernikov@gmail.com | 123456 | 1/1/2012 12:00:00 AM | NULL | 123456 | NULL | NULL |
| 3 | chernikov2@gmail.com | 123456 | 1/1/2012 12:00:00 AM | NULL | 123456 | NULL | NULL |

И UserRole

|  |  |  |
| --- | --- | --- |
| **ID** | **UserID** | **RoleID** |
| 1 | 1 | 1 |
| 2 | 1 | 2 |
| 3 | 3 | 2 |

Создадим консольный проект Lesson3 и поключим LessonProject.Model. Добавим сборку System.Configuration и System.Data.Linq. Проинициализируем context и выведем данные о ролях:

class Program

{

static void Main(string[] args)

{

var context = new LessonProjectDbDataContext(ConfigurationManager.ConnectionStrings["ConnectionString"].ConnectionString);

var roles = context.Roles.ToList();

foreach (var role in roles)

{

Console.WriteLine("{0} {1} {2}", role.ID, role.Code, role.Name);

}

Console.ReadLine();

}

}

Для добавления строки в Role делаем так:

var newRole = new Role

{

Code = "manager",

Name = "Менеджер"

};

context.Roles.InsertOnSubmit(newRole);

context.Roles.Context.SubmitChanges();

Для удаления строки в Role делаем так:

var role = context.Roles.Where(p => p.Name == "Менеджер").FirstOrDefault();

if (role != null)

{

context.Roles.DeleteOnSubmit(role);

context.Roles.Context.SubmitChanges();

}

Для изменения данных делаем так:

var role = context.Roles.Where(p => p.Name == "Менеджер").FirstOrDefault();

if (role != null)

{

role.Name = "Манагер";

context.Roles.Context.SubmitChanges();

}

Для манипуляции данных используется язык запросов Linq. Мы рассмотрим только некоторые основные функции Linq. Linq применяется для типов реализующий интерфейс IQueryable<>

**.Where()** – основная функция фильтрации. Возвращает тип IQueryable. Условие внутри должно возвращать булево значение (bool).

var roles = context.Roles.Where(p => p.Name == "Менеджер")

**.FirstOrDefault()** - .First(), .Single(), .SingleOrDefault() – получают первую или единственную запись. Если записи нет, то FirstOrDefault() или SingleOrDefault() возвращают null (на самом деле значение по умолчанию этого типа).

var roles = context.Roles.Where(p => p.Name == "Менеджер").FirstOrDefault() – получаем первую (или не получаем) роль названную «Менеджер».

**.Take()** – выбирает N первых записей

var roles = context.Roles.Where(p => p.Name == "Менеджер").Take(4) – выберет 4 первые записи

**.Skip()** – пропускает выбор N первых записей

var roles = context.Roles.Where(p => p.Name == "Менеджер"). Skip(2).Take(3) – пропустит первые 2 и выберет 3 следующие записи

**.OrderBy()** – сортирует по возрастанию. А также OrderByDescending(), ThenBy(), ThenByDescending(). Лябда выражение должно возвращать int по которому и будет происходить сортировка.

var roles = context.Roles.Where(p => p.Name == "Менеджер").OrderBy(p => p.ID) – сортирует по порядку

**.Count()** – получает количество записей

var rolesCount = context.Roles.Where(p => p.Name == "Менеджер").Count() – количество записей

**.Any()** – существует одна или больше записей по данному условию

var rolesExist = context.Roles.Where(p => p.Name == "Менеджер").Any() – есть ли запись такая

**.Select()** – возвращает IQueryable заданного типа, может быть даже динамический:

var otherRole = context.Roles.Where(p => p.Name == "Менеджер").Select(p => new {  
ID = p.ID, Kod = p.Code}) – получаем динамический тип сформированный на основе Role.

**.SelectMany()** – возвращает объединение всех IQueryable типов внутри выборки:

var otherRole = context.Roles.Where(p => p.Name == "Менеджер").SelectMany(p => p.UserRoles) – получаем все UserRole из роли названной «Менеджер»

**.Distinct()** – удаляет дубликаты

var managers = context.Roles.Where(p => p.Name == "Менеджер").SelectMany(p => p.UserRoles).Select(p => p.User).Distinct() – все пользователи с ролью названной «Менеджер»

*Примечание: First(), FirstOrDefault(), Single(), SingleOrDefault(), Any(), Count() – могут применять параметр соответствующий Where() тем самым можно сокращать запись:*

*var roles = context.Roles.FirstOrDefault(p => p.Name == "Менеджер")*

*Примечание, больше примеров и вариантов использования linq можете найти:*<http://code.msdn.microsoft.com/101-LINQ-Samples-3fb9811b>

### Создание репозитория IRepository , SqlRepository.

Собственно с БД мы уже можем работать, только теперь надо отделить модель данных от конкретной реализации, т.е. наши контроллеры про context и System.Data.Linq вообще не должны ничего знать.

Для этого создадим интерфейс IRepository где будет дан доступ ко данным, а также выведены методы для создания, изменения, удаления этих данных.

public interface IRepository

{

IQueryable<Role> Roles { get; }

bool CreateRole(Role instance);

bool UpdateRole(Role instance);

bool RemoveRole(int idRole);

…

}

И реализацию назовем SqlRepository. Так как мы с данным контекстом SqlRepository не хотим особо связывать, то добавим Ninject модуль в проект LessonProject.Model :

Install-Package Ninject

Получим класс SqlRepository:

public class SqlRepository : IRepository

{

[Inject]

public LessonProjectDbDataContext Db { get; set; }

public IQueryable<Role> Roles

{

get { throw new NotImplementedException(); }

}

public bool CreateRole(Role instance)

{

throw new NotImplementedException();

}

public bool UpdateRole(Role instance)

{

throw new NotImplementedException();

}

public bool RemoveRole(int idRole)

{

throw new NotImplementedException();

}

}

Прежде, чем реализовать доступ ко всем таблицам, создание, удаление и изменение подумаем, что файл этот будет выглядеть весьма монструозно, и тяжело будет физически управлять таким кодом. Так что сделаем отдельную папку SqlRepository и SqlRepository класс сделаем partial, а в папке создадим реализации разбитые по каждой таблице. Назовем файл Role и реализуем:

public partial class SqlRepository

{

public IQueryable<Role> Roles

{

get

{

return Db.Roles;

}

}

public bool CreateRole(Role instance)

{

if (instance.ID == 0)

{

Db.Roles.InsertOnSubmit(instance);

Db.Roles.Context.SubmitChanges();

return true;

}

return false;

}

public bool RemoveRole(int idRole)

{

Role instance = Db.Roles.FirstOrDefault(p => p.ID == idRole);

if (instance != null)

{

Db.Roles.DeleteOnSubmit(instance);

Db.Roles.Context.SubmitChanges();

return true;

}

return false;

}

}

Ок. Небольшой проект содержит от 10 до 40 таблиц, большой проект от 40, и собственно всё это хотелось бы хоть как-то автоматизировать. Создадим несколько сниппетов, для IRepository, для SqlRepository. Сниппеты – это уже готовые шаблоны кода, которые вызываются с помощью intelliSence, и позволяют быстро создавать код.

### Сниппеты

Для IRepository таблиц, назовем table.snippet:

<CodeSnippets

xmlns="http://schemas.microsoft.com/VisualStudio/2005/CodeSnippet">

<CodeSnippet Format="1.0.0" >

<Header>

<Title>

Table

</Title>

<Shortcut>Table</Shortcut>

</Header>

<Snippet>

<Declarations>

<Literal>

<ID>Table</ID>

<ToolTip>Table name for create.</ToolTip>

<Default>Table</Default>

</Literal>

</Declarations>

<Code Language="CSharp">

<![CDATA[

#region $Table$

IQueryable<$Table$> $Table$s { get; }

bool Create$Table$($Table$ instance);

bool Update$Table$($Table$ instance);

bool Remove$Table$(int id$Table$);

#endregion

]]>

</Code>

</Snippet>

</CodeSnippet>

</CodeSnippets>

Для SqlRepository создадим сниппет sqlTable.snippet:

<CodeSnippets

xmlns="http://schemas.microsoft.com/VisualStudio/2005/CodeSnippet">

<CodeSnippet Format="1.0.0" >

<Header>

<Title>

Sql repository

</Title>

<Shortcut>sqltable</Shortcut>

</Header>

<Snippet>

<Declarations>

<Literal>

<ID>Table</ID>

<ToolTip>Table name for create.</ToolTip>

<Default>Table</Default>

</Literal>

</Declarations>

<Code Language="CSharp">

<![CDATA[

public IQueryable<$Table$> $Table$s

{

get

{

return Db.$Table$s;

}

}

public bool Create$Table$($Table$ instance)

{

if (instance.ID == 0)

{

Db.$Table$s.InsertOnSubmit(instance);

Db.$Table$s.Context.SubmitChanges();

return true;

}

return false;

}

public bool Update$Table$($Table$ instance)

{

$Table$ cache = Db.$Table$s.Where(p => p.ID == instance.ID).FirstOrDefault();

if (cache != null)

{

//TODO : Update fields for $Table$

Db.$Table$s.Context.SubmitChanges();

return true;

}

return false;

}

public bool Remove$Table$(int id$Table$)

{

$Table$ instance = Db.$Table$s.Where(p => p.ID == id$Table$).FirstOrDefault();

if (instance != null)

{

Db.$Table$s.DeleteOnSubmit(instance);

Db.$Table$s.Context.SubmitChanges();

return true;

}

return false;

}

]]>

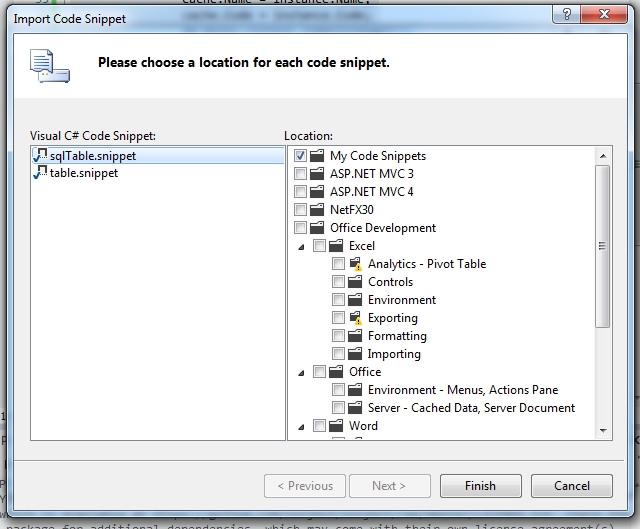
</Code>

</Snippet>

</CodeSnippet>

</CodeSnippets>

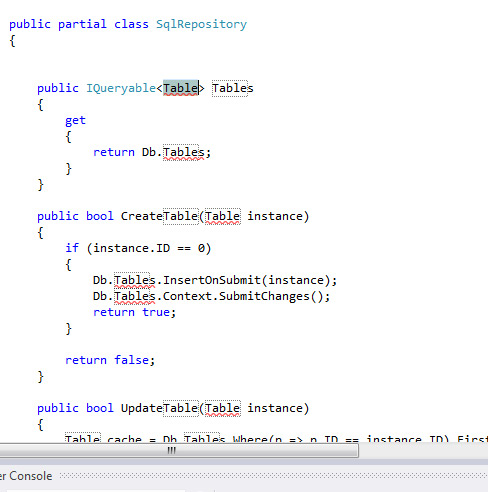
Для того чтобы добавить code-snippet откроем TOOLS -> Code Snippet Manager… (Ctrl-K, B). В открывшемся окне нажимаем Import и импортируем оба сниппета в My Code snippet:



Finish, OK.

Используем для таблиц User и UserRole.





Осталось прописать только поля для Update[имя таблицы], но это уже меньше работы.

### Proxy

Как видно классы, которые мы используем являются partial, поэтому их можно дополнить, создадим подобно SqlRepository папку Proxy, где будем размещать partial классы. Например для класса User создадим метод, который автоматически генерирует строку требуемую для активации пользователя:

public partial class User

{

public static string GetActivateUrl()

{

return Guid.NewGuid().ToString("N");

}

}

Используем это:

public bool CreateUser(User instance)

{

if (instance.ID == 0)

{

**instance.AddedDate = DateTime.Now;**

**instance.ActivatedLink = User.GetActivateUrl();**

Db.Users.InsertOnSubmit(instance);

Db.Users.Context.SubmitChanges();

return true;

}

return false;

}

### Использование в asp.net mvc

Добавим строку доступа к БД в web.Config:

<connectionStrings>

<add name="ConnectionString" connectionString="Data Source=SATURN-PC;Initial Catalog=LessonProject;Integrated Security=True;Pooling=False" providerName="System.Data.SqlClient" />

</connectionStrings>

Проинициализируем работу с БД в Ninject:

private static void RegisterServices(IKernel kernel)

{

kernel.Bind<LessonProjectDbDataContext>().ToMethod(c => new LessonProjectDbDataContext(ConfigurationManager.ConnectionStrings["ConnectionString"].ConnectionString));

kernel.Bind<IRepository>().To<SqlRepository>().InRequestScope();

}

Как видно тут мы используем область InRequestScope(), т.е. каждый запрос будет использовать отдельный объект SqlRepository. Это позволит избежать коллизий при исполнении, так как одновременно к БД может происходить много запросов.

Объявляем IRepository в контроллере:

public class HomeController : Controller

{

[Inject]

public IRepository Repository { get; set; }

public ActionResult Index()

{

var roles = Repository.Roles.ToList();

return View(roles);

}

}

И обновляем View:

@model IList<LessonProject.Model.Role>

@{

ViewBag.Title = "LessonProject";

Layout = "~/Views/Shared/\_Layout.cshtml";

}

<h2>LessonProject</h2>

<p>

@foreach (var role in Model)

{

<div class="item">

<span class="id">

@role.ID

</span>

<span class="name">

@role.Name

</span>

<span class="Code">

@role.Code

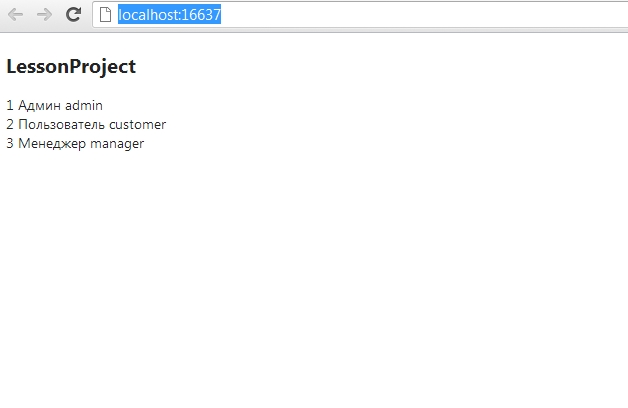
</span>

</div>

}

</p>

Получаем хороший результат:



## Урок 4. Routing и Bundles

Цель урока: Изучить инициализацию маршрутизации. Деление на Areas в приложении. Принципы создания маршрутизации.

### Controller и Action.

Веб-сайт состоит из страниц. Вообще веб-сайт состоит не из страниц, а из ответов на запросы, но какую-то определенную структуру мы ходим иметь.

Собственно, у нас есть маршрутизатор, который должен определить, какой метод у какого контроллера вызвать. Поэтому 2 основных параметра, которые обязательно должны быть – это controller и action. Рассмотрим в App\_Start/RouteConfig.cs как задаюется шаблон маршрутов:

routes.MapRoute(

name: "Default",

url: "{controller}/{action}/{id}",

defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }

);

*Примечание: инициализацию api маршрутизации мы рассмотрим позже.*

Таким образом, url = «/Role/Create/2» будет означать, что мы находим контроллер RoleController, в этом контроллере находим Create метод, который может принимать (а может и не принимать) параметр id. И если он принимает параметр id, то id = 2 или даже id = “2” в зависимости что за тип будет.

Defaults обозначает, что если строка будет “/Role/Create” – то в случае что Create метод с параметром id и по умолчанию не стоит значение, или не может быть создано default(), то по возможности будет выбран другой метод (мы же полиморфны). Иначе будет сгенерирована ошибка, что не найден метод готовый принять такой запрос.

public ActionResult Index(int? id)

{

//ok

return View();

}

…

public ActionResult Index(int id = 0)

{

//ok

return View();

}

…

public ActionResult Index(int id)

{

//fail

return View();

}

В случае url = “/Role” будет вызван метод Index в контроллере RoleController.

В случае url = “/” будет вызван метод Index в контроллере HomeController.

Рассмотрим на примерах.

### BaseController

Создадим несколько контроллеров, но для того чтобы не создавать постоянно доступ к репозиторию, превоначально создадим базовый контроллер BaseController:

public abstract class BaseController : Controller

{

[Inject]

public IRepository Repository { get; set; }

}

И

public class HomeController : BaseController

{

public ActionResult Index()

{

return View();

}

}

View Home/Index.cshtml

@{

ViewBag.Title = "LessonProject";

Layout = "~/Views/Shared/\_Layout.cshtml";

}

<h2>LessonProject</h2>

<p>

<div class="menu">

<a href="@Url.Action("Index", "Role")">Роли</a>

@Html.ActionLink("Пользователи", "Index", "User")

</div>

</p>

Добавим для просмотра RoleController:

public class RoleController : BaseController

{

public ActionResult Index()

{

var roles = Repository.Roles.ToList();

return View(roles);

}

}

И

@model IList<LessonProject.Model.Role>

@{

ViewBag.Title = "Roles";

Layout = "~/Views/Shared/\_Layout.cshtml";

}

<h2>Roles</h2>

<p>

@foreach (var role in Model)

{

<div class="item">

<span class="id">

@role.ID

</span>

<span class="name">

@role.Name

</span>

<span class="Code">

@role.Code

</span>

</div>

}

</p>

И такой же UserController собственно сделаем.

Рассмотрим как задаются маршруты с помощью Url.Action() и Html.ActionLink().

Url.Action() – принимает параметры, первым – action, потом – controller, потом через new {} – можно задавать и перечислять все остальные.

Html.ActionLink() – формирует тег <a>, первый параметр – наименование ссылки, второй – action, третьим – controller, четвертым(или пятым) параметром идут другие атрибуты тега, если мы хотим добавить другие параметры для маршрутизации – мы должны явно указать 5м параметром null. Т.е.

@Html.ActionLink(“Пользователь под номером 1”, “Item”, “User”, new {id = 1}, **null**)

Если не указать null:

@Html.ActionLink(“Пользователь под номером 1”, “Item”, “User”, new {id = 1})

то ссылка будет выглядеть так:

<a href=”/User/Item?length=4”>Пользователь под номером 1</a>

### Порядок в объявлении маршрутов

Создадим маршрут, который будет расположен ранее и относится только к RoleController:

routes.MapRoute(

name: "Role",

url: "roli/{action}/{id}",

defaults: new { controller = "Role", action = "Index", id = UrlParameter.Optional }

);

Однозначно его задавать, но не наименованием, а строкой “roli/{action}/{id}”, т.е. тут задается только action, а controller объявляется в defaults

Результат. Ссылка стала:

<a href="/roli">Роли</a>

Уберем из defaults action=”Index”:

<a href="/roli/Index">Роли</a>

Поместим после объявления “Defaults”:

<a href="/Role">Роли</a>

Так как с помощью вышестоящего правила уже можно однозначно задать путь маршрута.

Еще надо рассмотреть один метод ктороый называется IgnoreRoute, он указывает маршрутизатору, что если url подходит под шаблон, то нужно вернуть ресурс, который расположен по тому адресу, а не пытаться находить контроллер.

### Ограничения (Constrains)

Мы можем добавить в маршрутизацию ограничения запросов браузера, которые соответствуют особому маршруту. Например, id должен быть в нашем случае числовым:

routes.MapRoute(

name: "Role",

url: "roli/{action}/{id}",

defaults: new { controller = "Role", action = "Index", id = UrlParameter.Optional },

**constraints : new {id = @"\d+"}**

);

При передаче id, которое не соответствует данному условию, будет или выбран другой маршрут или метод не будет найден и ссылка будет битой:

Для:

<a href="@Url.Action("Index", "Role", new { id = "privet" })">Роли</a>

Будет:

<a href="/Role/Index/privet">Роли</a>

А для:

<a href="@Url.Action("Index", "Role", new { id = "1" })">Роли</a>

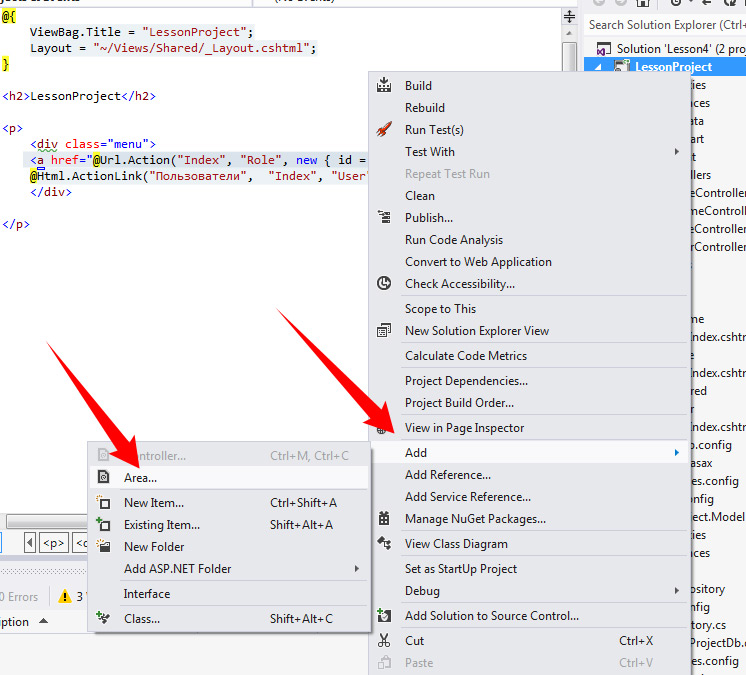
Будет:

<a href="/roli/Index/1">Роли</a>

*Примечание: Более подробно про ограничения можно узнать тут:* [*http://stephenwalther.com/archive/2008/08/07/asp-net-mvc-tip-30-create-custom-route-constraints.aspx*](http://stephenwalther.com/archive/2008/08/07/asp-net-mvc-tip-30-create-custom-route-constraints.aspx)

### Areas

Дабы разделить какие-то более общие по свойствам функциональные модули веб-приложения. Например, форум отдельно от всего сайта. Мы же поделим на часть Admin – где будет админка, и всё остальное, которое будет называться Default.



Сделаем следующие действия:

* Переименуем \_Default в Default везде.
* Перенесем свои контроллеры (кроме BaseController) в папку Areas/Default/Controllers
* Переименуем namespace для контроллеров в LessonProject.Areas.Default.Controllers
* Исправляем DefaultAreaRegistration:

Тут важно обратить внимание на новый параметр для задания маршрутов: namespaces, он указывает, из каких namespace можно выбирать контроллеры для разбора маршрута:

public class DefaultAreaRegistration : AreaRegistration

{

public override string AreaName

{

get

{

return "Default";

}

}

public override void RegisterArea(AreaRegistrationContext context)

{

context.MapRoute(

name : "default",

url : "{controller}/{action}/{id}",

defaults : new { controller = "Home", action = "Index", id = UrlParameter.Optional },

namespaces : new [] { "LessonProject.Areas.Default.Controllers" }

);

}

}

* В Global.asax есть строка AreaRegistration.RegisterAllAreas(); которая регистрирует все найденные объявления area, но она нам не подходит, так как если DefaultArea зарегистрировать быстрее AdminArea, то будет срабатывать маршрутизация Default, а в админку мы уже не сможем попасть поэтому исправляем:

var adminArea = new AdminAreaRegistration();

var adminAreaContext = new AreaRegistrationContext(adminArea.AreaName, RouteTable.Routes);

adminArea.RegisterArea(adminAreaContext);

var defaultArea = new DefaultAreaRegistration();

var defaultAreaContext = new AreaRegistrationContext(defaultArea.AreaName, RouteTable.Routes);

defaultArea.RegisterArea(defaultAreaContext);

* Регистрацию маршрутов убираем (не api).

Запускаем.

### API

Разобраться и описать как работает апи

<http://www.asp.net/web-api/overview/web-api-routing-and-actions/routing-in-aspnet-web-api>

## Урок 5. Создание записи в БД

Цель: Отследить весь путь создания записи в БД и вывода его. Вывод ошибок. Валидация. Мапперы. Написание атрибута валидации. Капча. Создание данных в БД.

### Введение

Наконец-то переходим к одному из самых важных уроков, собственно, где будет рассказано про создание записей. Любое действие на сайте, от сложных, когда мы заполняем регистрационную анкету, до простых, когда ставим лайк – выполняются следующие действия:

* Post\get запрос на сайт
* Авторизация и аутентификация
* Проверка введенных данных (валидация) на правильность
* Если проверка введенных данных показала, что введенные данные неверны, то в заполняемую форму выводится предупреждение.
* Если проверка введенных данных показал, что введенные данные верны, то данные сохраняются в БД и выводится страница с подтверждением.

### Регистрация

Сделаем форму для регистрации пользователя. При регистрации пользователя пользователь должен распознать капчу и повторить ввод пароля. Но начнем без этого. Создадим метод Register в контроллере UserController и View.

public ActionResult Register()

{

var newUser = new User();

return View(newUser);

}

Создаем и передаем во View новый объект User, так как полей у нас пока только 2 для заполнения создаем View:

@using (Html.BeginForm("Register", "User", FormMethod.Post, new { @class = "form-horizontal" }))

{

<fieldset>

<div class="control-group">

<label class="control-label" for="Email">

Email

</label>

<div class="controls">

@Html.ValidationMessage("Email")

@Html.TextBox("Email", Model.Email)

</div>

</div>

<div class="control-group">

<label class="control-label" for="FirstName">

Password

</label>

<div class="controls">

@Html.ValidationMessage("Password")

@Html.Password("Password", Model.Password)

</div>

</div>

<div class="form-actions">

<button type="submit" class="btn btn-primary">

Register

</button>

@Html.ActionLink("Cancel", "Index", null, null, new { @class = "btn" })

</div>

</fieldset>

}

Все эти дивы дивные, fieldset и buttonы сделаны по подобию, как это описано в фреймворке bootstrap (далее будем изучать).

Изучим основные Html-вставки:

Html.BeginForm("Register", "User", FormMethod.Post, new { @class = "form-horizontal" })

- формирует тег <form action=”/User/Register” method=”post” class=”form-horizontal”> и закрывает его после вызова Dispose() (закрытие кавычек using() {})

@Html.TextBox("Email", Model.Email)

- формирует тег <input type=”text” name=”Email” value=”@Model.Email”> (т.е. в значение тега записывается значение Email переданного объекта)

@Html.ValidationMessage("Password")

- выводит тег ошибки если есть такая

@Html.Password("Password", Model.Password)

- выводит тег <input type=”password” name=”Password” value=”@Model.Password”>

После нажатия на кнопку Register идет Http-запрос типа POST (так как FormMethod.Post). И передает данные Email=&Password=.

Создадим метод Register принимающий в качестве параметра тип User, и пометим его атрибутом HttpPost, а предыдущий атрибутом HttpGet. Контроллер различает, какой из типов запроса сейчас происходит и перенаправляет на тот, который необходим:

[HttpGet]

public ActionResult Register()

{

var newUser = new User();

return View(newUser);

}

[HttpPost]

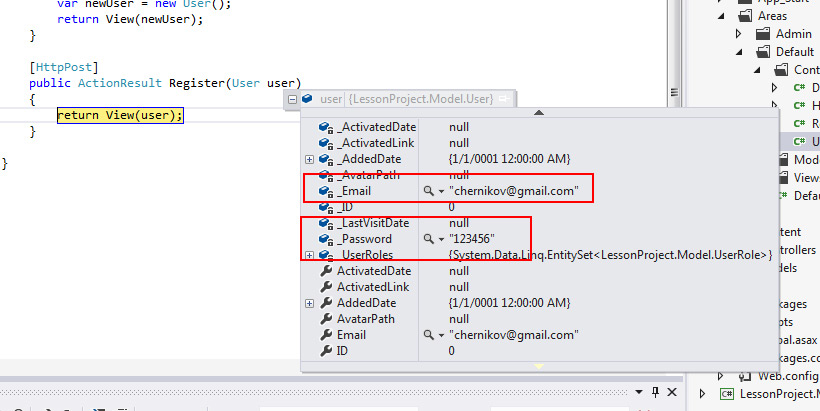
public ActionResult Register(User user)

{

return View(user);

}

Сделаем точку останова на втором методе Register и проверим, какой объект приходит к нам:



Видим что поля Email и Password заполнены, остальные остались нулевыми или по умолчанию (default).

Так как мы должны принять еще 2 поля (провтор пароля и капчу) то добавим это поля в наш User partial class:

public partial class User

{

public static string GetActivateUrl()

{

return Guid.NewGuid().ToString("N");

}

**public string ConfirmPassword { get; set; }**

**public string Captcha { get; set; }**

}

Добавим поля во View:

<div class="control-group">

<label class="control-label" for="FirstName">

Confirm Password

</label>

<div class="controls">

@Html.ValidationMessage("ConfirmPassword")

@Html.Password("ConfirmPassword", Model.ConfirmPassword)

</div>

</div>

<div class="control-group">

<label class="control-label" for="FirstName">

Captcha

</label>

</div>

<div class="control-group">

<label class="control-label" for="FirstName">

Тут картинка 1234

</label>

<div class="controls">

@Html.ValidationMessage("Captcha")

@Html.TextBox("Captcha", Model.Captcha)

</div>

</div>

Капчу пока не будем делать, просто она будет равна 1234.

### Валидация

Ок, вот условия для правильности данных:

* Поле email не нулевое
* Email – это корректно введенный адрес почты, т.е. с собачкой
* Email добавляемый в БД - уникальный
* Пароль не нулевой
* Пароли совпадают
* Капча равна 1234

Если какое-то из этих условий не соблюдается, то выдается ошибка.

#### IValidatableObject

Так как у нас класс User немного partial, то мы можем реализовать для него IValidatableObject интерфейс, для этого правда придется добавить в проект System.Component.DataAnnotation. Это не очень хорошо, так как эта сборка необходима для валидации, а валидация – это прерогатива контроллеров в MVC. Так что мы тут немного нарушаем принцип, потом будет еще одна загвоздка, но о ней чуть позже.

Класс User:

public partial class User : **IValidatableObject**

{

public static string GetActivateUrl()

{

return Guid.NewGuid().ToString("N");

}

public string ConfirmPassword { get; set; }

public string Captcha { get; set; }

**public IEnumerable<ValidationResult> Validate(ValidationContext validationContext)**

**{**

**//Не нулевой Email**

**if (string.IsNullOrWhiteSpace(Email))**

**{**

**yield return new ValidationResult("Введите email", new string[] {"Email"});**

**}**

**//корректный Email**

**var regex = new Regex(@"\w+([-+.']\w+)\*@\w+([-.]\w+)\*\.\w+([-.]\w+)\*", RegexOptions.Compiled);**

**var match = regex.Match(Email);**

**if (!(match.Success && match.Length == Email.Length))**

**{**

**yield return new ValidationResult("Введите корректный email", new string[] { "Email" });**

**}**

**//пароль не нулевой**

**if (string.IsNullOrWhiteSpace(Password))**

**{**

**yield return new ValidationResult("Введите пароль", new string[] { "Password" });**

**}**

**//пароли совпадают**

**if (Password != ConfirmPassword)**

**{**

**yield return new ValidationResult("Пароли не совпадают", new string[] { "ConfirmPassword" });**

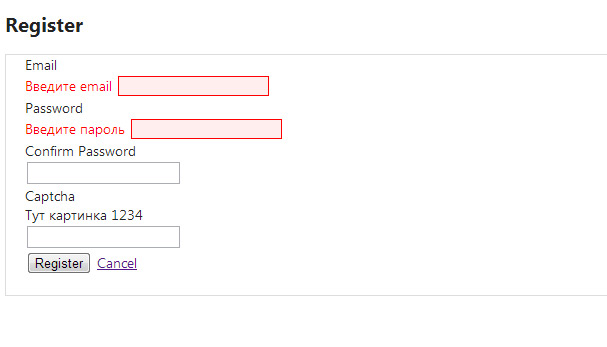
**}**

**}**

}

Мы смогли **сделать** проверку на 4 из 6 правил валидации, но ничего проверим пока так, а остальные добавим в контроллере непосредственно.

Выполняем форму, получаем:



Т.е. обе наши ошибки были отловлены.

Есть 2 стандартных метода вывести ошибку: это Html.ValidationMessage(“ErrorField”) и Html.ValidationSummary(). Первый выводит ошибку, связанную с конкретным неверное введенным полем, а второе выведет все (или все оставшиеся) ошибки.

Добавляем в контроллер проверку на капчу и проверку на существование Email в БД:

if (user.Captcha != "1234")

{

ModelState.AddModelError("Captcha", "Текст с картинки введен неверно");

}

var anyUser = Repository.Users.Any(p => string.Compare(p.Email, user.Email) == 0);

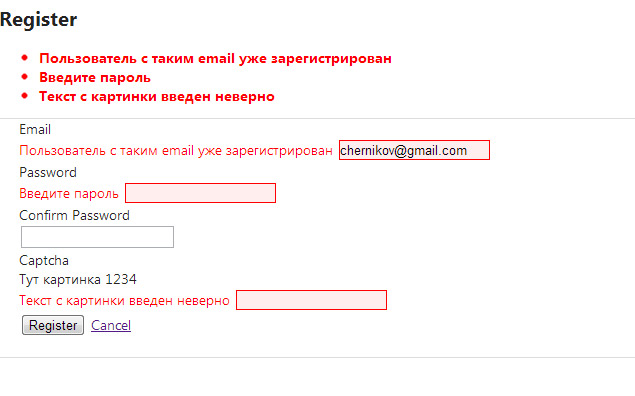
if (anyUser)

{

ModelState.AddModelError("Email", "Пользователь с таким email уже зарегистрирован");

}

И результат:



Что же, с задачей мы справились, но в дальнейшем, мы с таким способом приобретем несколько проблем:

* Класс User всегда будет содержать проверку на необходимость пароля и идентичность паролей, а например, при изменении данных в личном кабинете, мы вообще не должны вводить пароль. Т.е. необходимо будет вводить другие поля, которые будут обозначать – это регистрация, это смена пароля, это изменение данных.
* Валидацию мы сделали частично в Model-части и частично в Controller-части – это не совсем хрестоматийно.

Есть решение, мы создаем класс, который является представлением класса User, организует валидацию. Мы назовем его UserView и создадим в папке Models/ViewModels:

public class UserView

{

public int ID { get; set; }

public string Email { get; set; }

public string Password { get; set; }

public string ConfirmPassword { get; set; }

public string Captcha { get; set; }

public string AvatarPath { get; set; }

}

### Automapping

Прежде чем приступить к использованию этого класса, заметим, что это не совсем удобно. Мы создали совершенно другой класс но добавлять в БД мы должны класс User, а это означает, что в каком-то месте программы мы должны передавать от объекта UserView в User поля, так и наоборот, а при большом кол-ве объектов и полей – это немного рутинно, к тому же подобное у нас уже есть в функции Update[Table] в репозитории. Но есть выход. Существуют так называемые мапперы object-to-object.

Одним из самых популярных является automapper (<http://automapper.org/>). Собственно эта библиотека берет на себя работу по переводу одного объекта в другой, и как мы дальше увидим, там еще есть много других вкусных плюшек.

Устанавливаем Automapper:

Install-Package AutoMapper

Так как всё что мы используем, мы стараемся сильно не связывать, то организуем интерфейс + реализацию и зарегистрируем это в Ninject и выведем использование в контроллер.

Создаем в /Mappers

public interface IMapper

{

object Map(object source, Type sourceType, Type destinationType);

}

Реализация:

public class CommonMapper : IMapper

{

static CommonMapper()

{

Mapper.CreateMap<User, UserView>();

Mapper.CreateMap<UserView, User>();

}

public object Map(object source, Type sourceType, Type destinationType)

{

return Mapper.Map(source, sourceType, destinationType);

}

}

Регистрация (пусть будет как объект одиночка):

kernel.Bind<IMapper>().To<CommonMapper>().InSingletonScope();

В BaseController:

public abstract class BaseController : Controller

{

[Inject]

public IRepository Repository { get; set; }

**[Inject]**

**public IMapper ModelMapper { get; set; }**

}

Ок, теперь изменим UserController (и View) с использованием UserView:

[HttpGet]

public ActionResult Register()

{

**var newUserView = new UserView();**

**return View(newUserView);**

}

[HttpPost]

public ActionResult Register(**UserView userView**)

{

if (userView.Captcha != "1234")

{

ModelState.AddModelError("Captcha", "Текст с картинки введен неверно");

}

var anyUser = Repository.Users.Any(p => string.Compare(p.Email, userView.Email) == 0);

if (anyUser)

{

ModelState.AddModelError("Email", "Пользователь с таким email уже зарегистрирован");

}

if (ModelState.IsValid)

{

**var user = (User)ModelMapper.Map(userView, typeof(UserView), typeof(User));**

//TODO: Сохранить

}

return View(userView);

}

И в Register.cshtml изменится первая строка:

@model LessonProject.Models.ViewModels.UserView

#### Атрибуты

Для UserView будем использовать для валидации атрибуты.

Добавим сборку:

using System.ComponentModel.DataAnnotations;

public class UserView

{

public int ID { get; set; }

[Required(ErrorMessage="Введите email")]

public string Email { get; set; }

[Required(ErrorMessage="Введите пароль")]

public string Password { get; set; }

[Compare("Password", ErrorMessage="Пароли должны совпадать")]

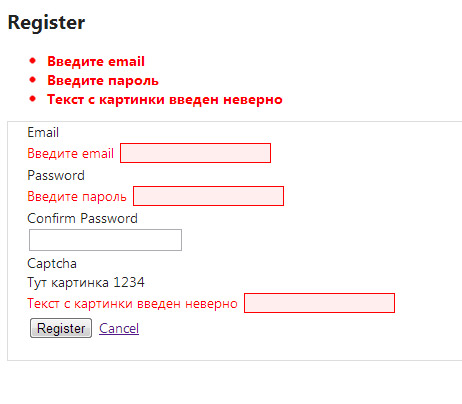
public string ConfirmPassword { get; set; }

public string Captcha { get; set; }

public string AvatarPath { get; set; }

}

Проверяем:



Мы смогли описать тут 5 из 6 правил валидации. Правило касающееся правильности введенного email – нет. Напишем же для этого свой класс атрибут – проверяющий корректность введенного email:

[AttributeUsage(AttributeTargets.Property, AllowMultiple = false)]

public class ValidEmailAttribute : ValidationAttribute

{

public override bool IsValid(object value)

{

if (value == null)

{

return true;

}

if (!(value is string))

{

return true;

}

var source = value as string;

if (string.IsNullOrWhiteSpace(source))

{

return true;

}

var regex = new Regex(@"\w+([-+.']\w+)\*@\w+([-.]\w+)\*\.\w+([-.]\w+)\*", RegexOptions.Compiled);

var match = regex.Match(source);

return (match.Success && match.Length == source.Length);

}

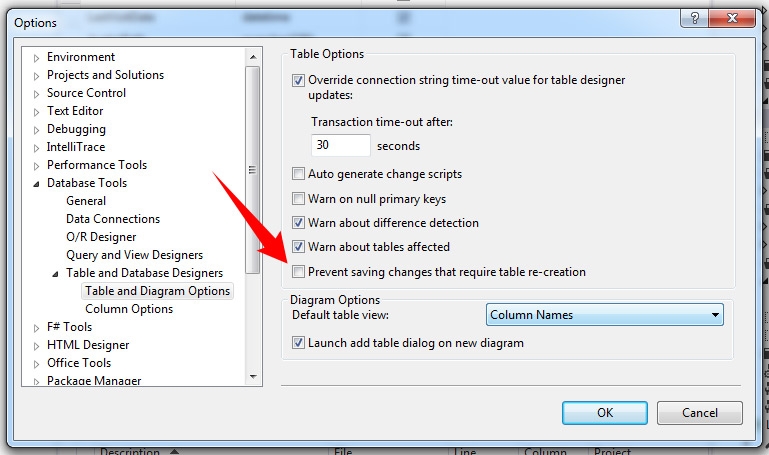
}

Вначале проверяем, что полученный объект есть строка, и строка не пустая, иначе возвращаем истину в проверке. Тут срабатывает правило, что «мы у инопланетян документы не проверяем», т.е. пока нет достаточных условий для проверки – мы не проверяем, а проверять будут другие атрибуты. Потом же с помощью регулярного выражения – проверяем. При желании в интернете можно найти более полную проверку регулярным выражением с использованием всех доменов первого уровня.

*Примечание: Можно подключить DataAnnotationsExtensions чтобы не писать самому нужные атрибуты (*[*http://dataannotationsextensions.org/*](http://dataannotationsextensions.org/)*)*

Добавим пользователю поле дня рождения. Да прям сейчас, и посмотрим, как можно реализовать выбор даты.

1. Добавляем поле в БД. Birthdate datetime null.

*Примечание: возможно надо будет снять эту галочку, чтобы спокойно изменять структуру БД:* 

1. В данных выставим всем записям значения 2012-1-1
2. Изменим поле Birthdate на datetime not null
3. Удаляем из LessonProjectDb.dbml таблицу User и заново переносим из Server Explorer
4. В SqlRepository/User.cs добавляем строку в UpdateUser():

public bool UpdateUser(User instance)

{

User cache = Db.Users.Where(p => p.ID == instance.ID).FirstOrDefault();

if (cache != null)

{

**cache.Birthdate = instance.Birthdate;**

cache.AvatarPath = instance.AvatarPath;

cache.Email = instance.Email;

Db.Users.Context.SubmitChanges();

return true;

}

return false;

}

1. В UserView у нас будет совершенно другое представление о поле Bithdate. И об этом чуть подробнее отдельно.

Выбор дня рождения у нас будет таким:



Тут надо решить несколько задач. Первая из них – выпадающий список, его организация. В Html (который мы еще потом подробно рассмотрим) есть DropDownList – который реализует выпадающий список. Параметры такие:

@Html.DropDownList(string name, IEnumerable<SelectListItem> selectList)

Смотрим SelectListItem:

public class SelectListItem

{

public SelectListItem();

public bool Selected { get; set; }

public string Text { get; set; }

public string Value { get; set; }

}

Ок, для выбора например из 1 - apple, 2 – orange (выбран), 3 - banana мы должны сделать:

public IEnumerable<SelectListItem> SelectFruit

{

get

{

yield return new SelectListItem() { Value = "1", Text = "apple", Selected = false };

yield return new SelectListItem() { Value = "2", Text = "orange", Selected = true };

yield return new SelectListItem() { Value = "3", Text = "banana", Selected = false };

}

}

И передать в DropDownList() вторым параметром, первый параметр – name. Этому name присвоится значение Value при подтверждении (сабмите) формы.

Ок, создадим реализацию для выбора дня рождения:

public int BirthdateDay { get; set; }

public int BirthdateMonth { get; set; }

public int BirthdateYear { get; set; }

public IEnumerable<SelectListItem> BirthdateDaySelectList

{

get

{

for (int i = 1; i < 32; i++)

{

yield return new SelectListItem

{

Value = i.ToString(),

Text = i.ToString(),

Selected = BirthdateDay == i

};

}

}

}

public IEnumerable<SelectListItem> BirthdateMonthSelectList

{

get

{

for (int i = 1; i < 13; i++)

{

yield return new SelectListItem

{

Value = i.ToString(),

Text = new DateTime(2000, i, 1).ToString("MMMM"),

Selected = BirthdateMonth == i

};

}

}

}

public IEnumerable<SelectListItem> BirthdateYearSelectList

{

get

{

for (int i = 1910; i < DateTime.Now.Year; i++)

{

yield return new SelectListItem

{

Value = i.ToString(),

Text = i.ToString(),

Selected = BirthdateYear == i

};

}

}

}

И во View:

<div class="control-group">

<label class="control-label" for="FirstName">

Birth date

</label>

<div class="controls">

@Html.DropDownList("BirthdateDay", Model.BirthdateDaySelectList)

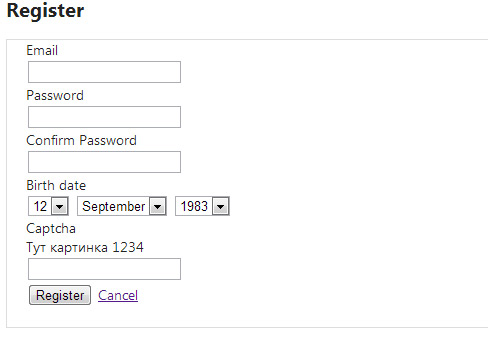
@Html.DropDownList("BirthdateMonth", Model.BirthdateMonthSelectList)

@Html.DropDownList("BirthdateYear", Model.BirthdateYearSelectList)

</div>

</div>

Запустим приложение и поставим точку останова на приеме данных и проверим как мы получаем данные для полей даты рождения для объекта UserView:





Ок, теперь осталось правильно передать их в объект User. Опишем эту передачу во описании маппинга:

Mapper.CreateMap<User, UserView>()

.ForMember(dest => dest.BirthdateDay, opt => opt.MapFrom(src => src.Birthdate.Day))

.ForMember(dest => dest.BirthdateMonth, opt => opt.MapFrom(src => src.Birthdate.Month))

.ForMember(dest => dest.BirthdateYear, opt => opt.MapFrom(src =>src.Birthdate.Year));

Mapper.CreateMap<UserView, User>()

.ForMember(dest => dest.Birthdate, opt => opt.MapFrom(src => new DateTime(src.BirthdateYear, src.BirthdateMonth, src.BirthdateDay)));

Здесь мы задаем правила однозначного перевода из свойств BirthdateDay, BirthdateMonth, BirthdateYear в Birthdate и обратно.

### Captcha

Для создания капчи, мы используем отдельный класс, который создаст нам картинку с цифрами и выведет как картинку. Сами цифры будут сохранены в сессионные данные. Про сессию мы чуть дальше рассмотрим. Сейчас надо знать только что сессия однозначно определяет пользователя.

Создадим Tools/CaptchaImage.cs

/// <summary>

/// Генерация капчи

/// </summary>

public class CaptchaImage

{

public const string CaptchaValueKey = "CaptchaImageText";

public string Text

{

get { return text; }

}

public Bitmap Image

{

get { return image; }

}

public int Width

{

get { return width; }

}

public int Height

{

get { return height; }

}

// Internal properties.

private string text;

private int width;

private int height;

private string familyName;

private Bitmap image;

// For generating random numbers.

private Random random = new Random();

public CaptchaImage(string s, int width, int height)

{

text = s;

SetDimensions(width, height);

GenerateImage();

}

public CaptchaImage(string s, int width, int height, string familyName)

{

text = s;

SetDimensions(width, height);

SetFamilyName(familyName);

GenerateImage();

}

// ====================================================================

// This member overrides Object.Finalize.

// ====================================================================

~CaptchaImage()

{

Dispose(false);

}

// ====================================================================

// Releases all resources used by this object.

// ====================================================================

public void Dispose()

{

GC.SuppressFinalize(this);

Dispose(true);

}

// ====================================================================

// Custom Dispose method to clean up unmanaged resources.

// ====================================================================

protected virtual void Dispose(bool disposing)

{

if (disposing)

// Dispose of the bitmap.

image.Dispose();

}

// ====================================================================

// Sets the image aWidth and aHeight.

// ====================================================================

private void SetDimensions(int aWidth, int aHeight)

{

// Check the aWidth and aHeight.

if (aWidth <= 0)

throw new ArgumentOutOfRangeException("aWidth", aWidth, "Argument out of range, must be greater than zero.");

if (aHeight <= 0)

throw new ArgumentOutOfRangeException("aHeight", aHeight, "Argument out of range, must be greater than zero.");

width = aWidth;

height = aHeight;

}

// ====================================================================

// Sets the font used for the image text.

// ====================================================================

private void SetFamilyName(string aFamilyName)

{

// If the named font is not installed, default to a system font.

try

{

Font font = new Font(aFamilyName, 12F);

familyName = aFamilyName;

font.Dispose();

}

catch (Exception)

{

familyName = FontFamily.GenericSerif.Name;

}

}

// ====================================================================

// Creates the bitmap image.

// ====================================================================

private void GenerateImage()

{

// Create a new 32-bit bitmap image.

Bitmap bitmap = new Bitmap(width, height, PixelFormat.Format32bppArgb);

// Create a graphics object for drawing.

Graphics g = Graphics.FromImage(bitmap);

g.SmoothingMode = SmoothingMode.AntiAlias;

Rectangle rect = new Rectangle(0, 0, width, height);

// Fill in the background.

HatchBrush hatchBrush = new HatchBrush(HatchStyle.SmallConfetti, Color.LightGray, Color.White);

g.FillRectangle(hatchBrush, rect);

// Set up the text font.

SizeF size;

float fontSize = rect.Height + 1;

Font font;

// Adjust the font size until the text fits within the image.

do

{

fontSize--;

font = new Font(familyName, fontSize, FontStyle.Bold);

size = g.MeasureString(text, font);

} while (size.Width > rect.Width);

// Set up the text format.

StringFormat format = new StringFormat();

format.Alignment = StringAlignment.Center;

format.LineAlignment = StringAlignment.Center;

// Create a path using the text and warp it randomly.

GraphicsPath path = new GraphicsPath();

path.AddString(text, font.FontFamily, (int)font.Style, font.Size, rect, format);

float v = 4F;

PointF[] points =

{

new PointF(random.Next(rect.Width) / v, random.Next(rect.Height) / v),

new PointF(rect.Width - random.Next(rect.Width) / v, random.Next(rect.Height) / v),

new PointF(random.Next(rect.Width) / v, rect.Height - random.Next(rect.Height) / v),

new PointF(rect.Width - random.Next(rect.Width) / v, rect.Height - random.Next(rect.Height) / v)

};

Matrix matrix = new Matrix();

matrix.Translate(0F, 0F);

path.Warp(points, rect, matrix, WarpMode.Perspective, 0F);

// Draw the text.

hatchBrush = new HatchBrush(HatchStyle.LargeConfetti, Color.LightGray, Color.DarkGray);

g.FillPath(hatchBrush, path);

// Add some random noise.

int m = Math.Max(rect.Width, rect.Height);

for (int i = 0; i < (int)(rect.Width \* rect.Height / 30F); i++)

{

int x = random.Next(rect.Width);

int y = random.Next(rect.Height);

int w = random.Next(m / 50);

int h = random.Next(m / 50);

g.FillEllipse(hatchBrush, x, y, w, h);

}

// Clean up.

font.Dispose();

hatchBrush.Dispose();

g.Dispose();

// Set the image.

image = bitmap;

}

}

Ок, суть такова что в свойство Image генерируется картинка состоящая из цифр (которые как бы сложно распознать) методом GenerateImage().

Теперь сделаем метод вывода UserController.Captcha():

public ActionResult Captcha()

{

Session[CaptchaImage.CaptchaValueKey] = new Random(DateTime.Now.Millisecond).Next(1111, 9999).ToString();

var ci = new CaptchaImage(Session[CaptchaImage.CaptchaValueKey].ToString(), 211, 50, "Arial");

// Change the response headers to output a JPEG image.

this.Response.Clear();

this.Response.ContentType = "image/jpeg";

// Write the image to the response stream in JPEG format.

ci.Image.Save(this.Response.OutputStream, ImageFormat.Jpeg);

// Dispose of the CAPTCHA image object.

ci.Dispose();

return null;

}

Что тут происходит:

* В сессии создаем случайное число от 1111 до 9999.
* Создаем в ci объект CatchaImage
* Очищаем поток вывода
* Задаем что mime-тип этого http-ответа будет “image/jpeg” т.е. картинка формата jpeg.
* Сохраняем bitmap в выходной поток с форматом ImageFormat.Jpeg
* Освобождаем ресурсы Bitmap
* Возвращаем null так как основная информация уже передана в поток вывода

Запрашиваем картинку из Register.cshtml:

<label class="control-label" for="FirstName">

<img src="@Url.Action("Captcha", "User")" alt="captcha" />

</label>

Проверка:

if (userView.Captcha != (string)Session[CaptchaImage.CaptchaValueKey])

{

ModelState.AddModelError("Captcha", "Текст с картинки введен неверно");

}

Вот и всё, закончили, добавляем создание записи и проверяем, как работает:

if (ModelState.IsValid)

{

var user = (User)ModelMapper.Map(userView, typeof(UserView), typeof(User));

**Repository.CreateUser(user);**

**return RedirectToAction("Index");**

}

## Урок 6. Авторизация.

Цель урока: Изучить способ авторизации через Cookie, использование стандартных атрибутов доступа к контроллеру и методу контроллера. Использование IPrincipal. Создание собственного модуля (IHttpModule) и собственного фильтра IActionFilter.

*Небольшое отступление: На самом деле в asp.net mvc все учебники рекомендуют пользоваться уже придуманной системой авторизации, которая называется* ***AspNetMembershipProvider****, она была описана в статье* <http://habrahabr.ru/post/142711/>*, но как-то там это объяснено с точки зрения – «нажимай и не понимай что там внутри». При первом знакомстве с asp.net mvc меня это смутило. Далее в этой статье* <http://habrahabr.ru/post/143024/> *- сказано, что пользоваться этим провайдером – нельзя.*  *Тут же мы достаточно глубоко изучаем всякие хитрые asp.net mvc стандартные приемы, так что это один из основных уроков.*

### Кукисы

Кукисы – это часть информации, отсылаемая сервером браузеру, и которую браузер возвращает обратно серверу вместе с каждым (почти каждым) запросом.

Сервер в заголовок ответа пишет:

Set-Cookie: value[; expires=date][; domain=domain][; path=path][; secure]

Например:

HTTP/1.1 200 OK  
Content-type: text/html  
Set-Cookie: name=value  
Set-Cookie: name2=value2; Expires=Wed, 09-Jun-2021 10:18:14 GMT

Браузер (если не истекло время действия кукиса) при каждом запросе:

GET /spec.html HTTP/1.1  
Host: www.example.org  
Cookie: name=value; name2=value2  
Accept: \*/\*

Устанавливаем cookie:

public ActionResult Index()

{

var cookie = new HttpCookie()

{

Name ="test\_cookie",

Value = DateTime.Now.ToString("dd.MM.yyyy"),

Expires = DateTime.Now.AddMinutes(10),

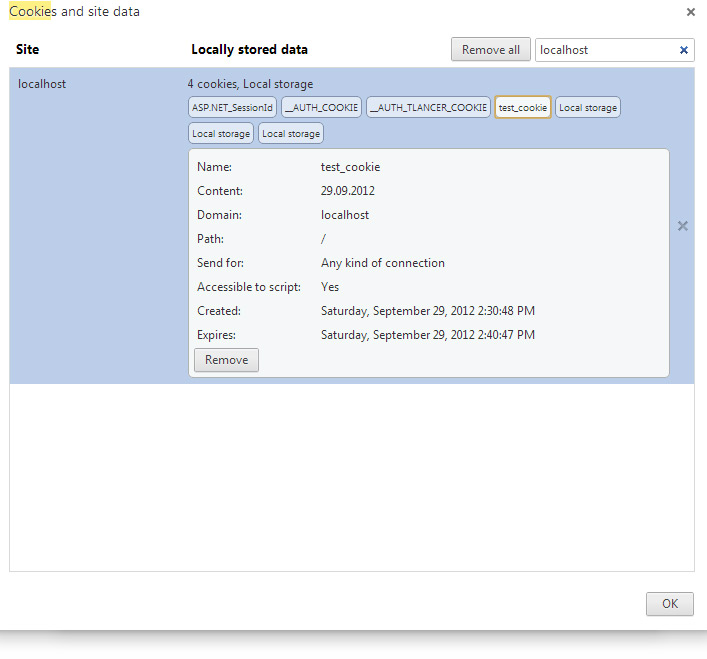
};

Response.SetCookie(cookie);

return View();

}

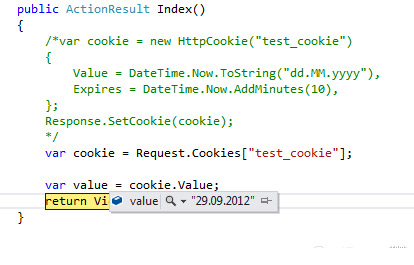
В Chrome проверяем установку:



Для получения кукисов:

var cookie = Request.Cookies["test\_cookie"];

Делаем точку останова и проверяем:



*Примечание: Тут подробнее можно изучить кукисы:*

<http://www.nczonline.net/blog/2009/05/05/http-cookies-explained/>

### Авторизация

Ок, в нашем случае авторизация будет основана на использовании кукисов. Поэтому изучим следующие положения:

* **FormsAuthenticationTicket** – мы воспользуемся этим классом, чтобы не хранить данные в открытом виде
* Нужно реализовать интерфейс **IPrincipal** и установить в HttpContext.User для проверки ролей и IIdentity интерфейса.
* Для интерфейса **IIdentity** сделать реализацию
* Вывести в **BaseController** в свойство **CurrentUser** значение пользователя, который сейчас залогинен.

Приступим.

Создадим интерфейс IAuthentication и его реализацию CustomAuthentication

public interface IAuthentication

{

/// <summary>

/// Конекст (тут мы получаем доступ к запросу и кукисам)

/// </summary>

HttpContext HttpContext { get; set; }

User Login(string login, string password, bool isPersistent);

User Login(string login);

void LogOut();

IPrincipal CurrentUser { get; }

}

Реализация:

public class CustomAuthentication : IAuthentication

{

private static NLog.Logger logger = NLog.LogManager.GetCurrentClassLogger();

private const string cookieName = "\_\_AUTH\_COOKIE";

public HttpContext HttpContext { get; set; }

[Inject]

public IRepository Repository { get; set; }

#region IAuthentication Members

public User Login(string userName, string Password, bool isPersistent)

{

User retUser = Repository.Login(userName, Password);

if (retUser != null)

{

CreateCookie(userName, isPersistent);

}

return retUser;

}

public User Login(string userName)

{

User retUser = Repository.Users.FirstOrDefault(p => string.Compare(p.Email, userName, true) == 0);

if (retUser != null)

{

CreateCookie(userName);

}

return retUser;

}

private void CreateCookie(string userName, bool isPersistent = false)

{

var ticket = new FormsAuthenticationTicket(

1,

userName,

DateTime.Now,

DateTime.Now.Add(FormsAuthentication.Timeout),

isPersistent,

string.Empty,

FormsAuthentication.FormsCookiePath);

// Encrypt the ticket.

var encTicket = FormsAuthentication.Encrypt(ticket);

// Create the cookie.

var AuthCookie = new HttpCookie(cookieName)

{

Value = encTicket,

Expires = DateTime.Now.Add(FormsAuthentication.Timeout)

};

HttpContext.Response.Cookies.Set(AuthCookie);

}

public void LogOut()

{

var httpCookie = HttpContext.Response.Cookies[cookieName];

if (httpCookie != null)

{

httpCookie.Value = string.Empty;

}

}

private IPrincipal \_currentUser;

public IPrincipal CurrentUser

{

get

{

if (\_currentUser == null)

{

try

{

HttpCookie authCookie = HttpContext.Request.Cookies.Get(cookieName);

if (authCookie != null && !string.IsNullOrEmpty(authCookie.Value))

{

var ticket = FormsAuthentication.Decrypt(authCookie.Value);

\_currentUser = new UserProvider(ticket.Name, Repository);

}

else

{

\_currentUser = new UserProvider(null, null);

}

}

catch (Exception ex)

{

logger.Error("Failed authentication: " + ex.Message);

\_currentUser = new UserProvider(null, null);

}

}

return \_currentUser;

}

}

#endregion

}

Суть сводится к следующему, мы при инициализации запроса получаем доступ к HttpContext.Request.Cookies и инициализируем UserProvider:

var ticket = FormsAuthentication.Decrypt(authCookie.Value);

\_currentUser = new UserProvider(ticket.Name, Repository);

Для авторизации в IRepository добавлен новый метод IRepository.Login. Реализация в SqlRepository:

public User Login(string email, string password)

{

return Db.Users.FirstOrDefault(p => string.Compare(p.Email, email, true) == 0 && p.Password == password);

}

UserProvider собственно реализует интерфейс IPrincipal (в котором есть проверка ролей и доступ к IIdentity).

Рассмотрим класс UserProvider:

public class UserProvider : IPrincipal

{

private UserIndentity userIdentity { get; set; }

#region IPrincipal Members

public IIdentity Identity

{

get

{

return userIdentity;

}

}

public bool IsInRole(string role)

{

if (userIdentity.User == null)

{

return false;

}

return userIdentity.User.InRoles(role);

}

#endregion

public UserProvider(string name, IRepository repository)

{

userIdentity = new UserIndentity();

userIdentity.Init(name, repository);

}

public override string ToString()

{

return userIdentity.Name;

}

Наш UserProvider знает про то, что его IIdentity классом есть UserIdentity, а поэтому знает про класс User, внутри которого мы реализуем метод InRoles(role):

public bool InRoles(string roles)

{

if (string.IsNullOrWhiteSpace(roles))

{

return false;

}

var rolesArray = roles.Split(new[] { "," }, StringSplitOptions.RemoveEmptyEntries);

foreach (var role in rolesArray)

{

var hasRole = UserRoles.Any(p => string.Compare(p.Role.Code, role, true) == 0);

if (hasRole)

{

return true;

}

}

return false;

}

В метод InRoles мы ожидаем что придет запрос о ролях, которые допущены к ресурсу, разделенные запятой, т.е. например “admin,moderator,editor”, если хотя бы одна из ролей есть у нашего User – то возвращаем – истину, доступ у него есть. Сравниваем по полю Role.Code, а не Role.Name.

Рассмотрим класс UserIdentity:

public class UserIndentity : IIdentity

{

public User User { get; set; }

public string AuthenticationType

{

get

{

return typeof(User).ToString();

}

}

public bool IsAuthenticated

{

get

{

return User != null;

}

}

public string Name

{

get

{

if (User != null)

{

return User.Email;

}

//иначе аноним

return "anonym";

}

}

public void Init(string email, IRepository repository)

{

if (!string.IsNullOrEmpty(email))

{

User = repository.GetUser(email);

}

}

}

В IRepository добавляем новый метод GetUser(email). Реализация для SqlRepository.GetUser():

public User GetUser(string email)

{

return Db.Users.FirstOrDefault(p => string.Compare(p.Email, email, true) == 0);

}

Ок, всё почти готово, выведем CurrentUser в BaseController:

[Inject]

public IAuthentication Auth { get; set; }

public User CurrentUser

{

get

{

return ((UserIndentity)Auth.CurrentUser.Identity).User;

}

}

Да, это немного не правильно, так как тут сильное связывание. Поэтому сделаем так. Введем еще один интерфейс IUserProvider, с которого мы будем требовать вернуть нам авторизованного User:

public interface IUserProvider

{

User User { get; set; }

}

…

public class UserIndentity : IIdentity, **IUserProvider**

{

/// <summary>

/// Текщий пользователь

/// </summary>

public User User { get; set; }

…

[Inject]

public IAuthentication Auth { get; set; }

public User CurrentUser

{

get

{

return ((IUserProvider)Auth.CurrentUser.Identity).User;

}

}

Ок, теперь попробуем инициализировать это всё.

Вначале добавим наш IAuthentication + CustomAuthentication в регистрацию к Ninject:

kernel.Bind<IAuthentication>().To<CustomAuthentication>().InRequestScope();

Потом создадим модуль, который будет будет на событие AuthenticateRequest совершать действие авторизации:

public class AuthHttpModule : IHttpModule

{

public void Init(HttpApplication context)

{

context.AuthenticateRequest += new EventHandler(this.Authenticate);

}

private void Authenticate(Object source, EventArgs e)

{

HttpApplication app = (HttpApplication)source;

HttpContext context = app.Context;

var auth = DependencyResolver.Current.GetService<IAuthentication>();

**auth.HttpContext = context;**

**context.User = auth.CurrentUser;**

}

public void Dispose()

{

}

}

Вся соль в выделенных строках (**auth.HttpContext = context и context.User = auth.CurrentUser**), как только наш модуль авторизации узнает о контексте и содержащихся в нем кукисах, то моментально получает доступ к имени, по имени он в репозитории получает данные, кто это и возвращает в BaseController. Но не сразу это всё, а по требованию.

Подключаем модуль в Web.config:

<system.web>

…

<httpModules>

<add name="AuthHttpModule" type="LessonProject.Global.Auth.AuthHttpModule"/>

</httpModules>

</system.web>

План таков:

* Наверху показываем авторизован пользователь или (если авторизован, то его email и ссылка на выход, если нет, то ссылки на вход и регистрацию)
* Создаем форму для входа
* Если пользователь правильно ввел данные – то авторизуем его и отправляем на главную страницу
* Если пользователь выходит – то убиваем его авторизацию

Поехали. Добавляем Html.Action(“UserLogin”, “Home”) – это partial view (т.е. кусок кода, который не имеет Layout) – т.е. выводится где прописан, а не в RenderBody().

\_Layout.cshtml:

<body>

**<div class="navbar navbar-fixed-top">**

**<div class="navbar-inner">**

**<div class="container">**

**<ul class="nav nav-pills pull-right">**

**@Html.Action("UserLogin", "Home")**

**</ul>**

**</div>**

**</div>**

**</div>**

@RenderBody()

HomeController.cs:

public ActionResult UserLogin()

{

return View(CurrentUser);

}

UserLogin.cshtml:

@model LessonProject.Model.User

@if (Model != null)

{

<li>@Model.Email</li>

<li>@Html.ActionLink("Выход", "Logout", "Login")</li>

}

else

{

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

<li>@Html.ActionLink("Регистрация", "Register", "User")</li>

}

Контроллер входа выхода LoginController:

public class LoginController : DefaultController

{

[HttpGet]

public ActionResult Index()

{

return View(new LoginView());

}

[HttpPost]

public ActionResult Index(LoginView loginView)

{

if (ModelState.IsValid)

{

var user = Auth.Login(loginView.Email, loginView.Password, loginView.IsPersistent);

if (user != null)

{

return RedirectToAction("Index", "Home");

}

ModelState["Password"].Errors.Add("Пароли не совпадают");

}

return View(loginView);

}

public ActionResult Logout()

{

Auth.LogOut();

return RedirectToAction("Index", "Home");

}

}

LoginView.cs:

public class LoginView

{

[Required(ErrorMessage = "Введите email")]

public string Email { get; set; }

[Required(ErrorMessage = "Введите пароль")]

public string Password { get; set; }

public bool IsPersistent { get; set; }

}

Страница для входа Index.cshtml:

@model LessonProject.Models.ViewModels.LoginView

@{

ViewBag.Title = "Вход";

Layout = "~/Areas/Default/Views/Shared/\_Layout.cshtml";

}

<h2>Вход</h2>

@using (Html.BeginForm("Index", "Login", FormMethod.Post, new { @class = "form-horizontal" }))

{

<fieldset>

<legend>Вход</legend>

<div class="control-group">

<label class="control-label" for="Email">

Email</label>

<div class="controls">

@Html.TextBox("Email", Model.Email, new { @class = "input-xlarge" })

<p class="help-block">Введите Email</p>

@Html.ValidationMessage("Email")

</div>

</div>

<div class="control-group">

<label class="control-label" for="Password">

Пароль</label>

<div class="controls">

@Html.Password("Password", Model.Password, new { @class = "input-xlarge" })

@Html.ValidationMessage("Password")

</div>

</div>

<div class="form-actions">

<button type="submit" class="btn btn-primary">

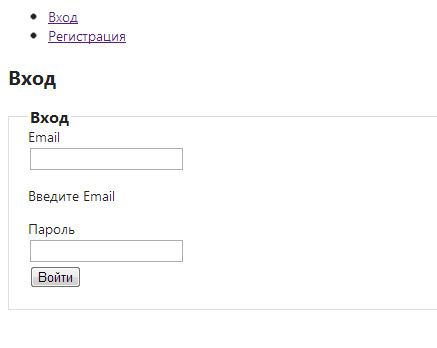
Войти</button>

</div>

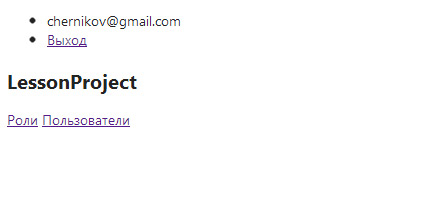
</fieldset>

}

Запускаем и проверяем:



После авторизации:



## Урок 7. Bootstrap, jQuery, Ajax

Цель: Bootstrap и дополнительный css. Определить правила работы с html, js и css файлами. Папка Media. Структура js-файлов. Использование jQuery, основные моменты, изучение селекторов, событий и др. addClass, removeClass, attr, data, динамическое создание dom-объекта, ajax.

Мы наконец-то приступаем к более детальному изучению клиентской части, которая уже в меньшей степени связана с asp.net mvc, но всё равно важна для веб-разработки.

### Twitter Bootstrap

Twitter Bootstrap – это css-фреймворк. Т.е. набор инструментов для создания блоков, кнопок, меток, форм и навигации. Наше приложение мы будем основывать на этом фрейморке.

Подробнее тут:

<http://twitter.github.com/bootstrap/>

Установим bootstrap:

Install-Package Twitter.Bootstrap

Удалим Jquery.UI:

Uninstall-Package Jquery.UI.Combined

Добавим в BundleConfig bootstap и уберем оттуда jquery.UI (App\_Start/BundleConfig.cs):

public class BundleConfig

{

public static void RegisterBundles(BundleCollection bundles)

{

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

"~/Scripts/jquery-1.\*"));

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

"~/Scripts/modernizr-\*"));

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

"~/Scripts/bootstrap\*"));

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

.Include("~/Content/site.css") /\* не перепутайте порядок \*/

.Include("~/Content/bootstrap\*"));

}

}

Важно помнить про порядок приоритета задания стилей:

* Наименее низким приоритетом обладают встроенные стили браузера
* Пользовательский стиль в браузере
* Наследуемые стили от родительских элементов
* Далее идут css стили заданные во внешних файлах, тут правило важнее (при прочих равных условиях), если файл содержащий правило находится после файла содержащий предыдущее. Это же правило относится и к взаимному расположению правил внутри документа.
* Далее по стоимости применения селекторов:
  + Теги имеют наименьший приоритет:

h1, div, p

* + Классы и псевдоклассы. Селектор:

.item { }, :hover { }

* + Аттрибуты [attr=”value”]. Пример:

[type=”checkbox”] {}

* + Идентификатор ID – наивысший приоритет. Типа

#main { }

* Еще более высоким приоритетом является задание стиля прямо в style теге:

<div style=”width : 203px”></div>

* И наиболее высоким приоритетом является стиль с сопроводительным словом *!important*.

Основная работа в задании стилей идет с помощью тегов, классов (псевдоклассов), и атрибутов. Использовать *!important* не рекомендуется как и задавать стили в атрибуте style и использовать атрибут ID.

Для задания стилей будем использовать css-файл site.css. Так как в bootstrap уже заданы стили для базовых форм удалим этот блок и оставим файл (Content/Site.css):

/\* Styles for validation helpers

-----------------------------------------------------------\*/

.field-validation-error {

color: #f00;

}

.field-validation-valid {

display: none;

}

.input-validation-error {

border: 1px solid #f00;

background-color: #fee;

}

.validation-summary-errors {

font-weight: bold;

color: #f00;

}

.validation-summary-valid {

display: none;

}

Это css стили которые используются для вывода ошибок в методах Html.ValidationMessage(), Html.ValidationSummary().

Ок, теперь определим правила по которым мы будем создавать свой стиль:

- Запрещено использовать селектор через id, типа #Main. Это связано с тем, что хотя браузеры и обрабатывают все элементы содержащий данный атрибут, но в правилах DOM атрибут ID должен быть уникальный во всей странице.

- Классы для офомления используют тип SmallCaps: some-class-name, все с маленькой буквы, для разделения используется “-”. Собственно, так, как это в самом CSS и используется.

- Классы предназначенные только для управления (использование исключительно в js-коде) для разделения используют тип lowerCamelCase: someClassName. Собственно, как и соглашения по написанию кода в js или в jquery.

- Атрибуты id задаются типом UpperCamelCase: id=”SomeButton”.

- Не используйте классы общего значения типа .list, .item, .button, .text, .user, .image, .container, .wrapper на верхнем уровне.

- Не усложняйте слова. **Не** используйте конструкцию типа: .main-container-left-part-wrapper-list.

- Задайте популярные стили, если это необходимо в общем описании типа: .error {color : red }, .left { float : left }. Хотя для этого лучше использовать стили bootstrap (предварительно их выучив) .pull-left и pull-right.

- Используйте каскадное описание от общего к частному:

.list

{

}

.list .item

{

}

- используйте селекторы для задания глубины применения стиля

.list > .item

{

}

- и для тегов с множественным классом:

.item.odd

{

}

для

<div class="item odd"></div>

- используйте следующую структуру документа

/\* Главные определения стилей для сайта

- основной шрифт

- междустрочное расстояние

- размер шрифта

- основные цвета ссылок и текста

- h1, h2, h3 ...

- a

- p

\*/

/\* Вспомогательные классы:

- .error

- .field-validation-error и др.

- .show-me { border : 1px solid red; }

- .hidden {display : none; }

- .relative { position : relative; }

\*/

/\* Основные вспомогательные окна и стили

- .popup-window

- .popup-window .close

- .popup-window .wrapper

- .popup-window .header

- .error-message

- .info-message

\*/

/\* Основные стили сайта

- header

- .logo

- .user-login

- nav.main-menu

- .main-content

- footer

\*/

/\*Главная страница \*/

/\* Страница входа \*/

/\* Страница ошибки \*/

/\* Остальные страницы… \*/

### Структура html-страницы.

Подключим стили и js файлы к основному layout файлу (/Areas/Default/Views/Shared/\_Layout.cshtml):

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8" />

<meta name="viewport" content="width=device-width" />

<title>@ViewBag.Title</title>

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

@RenderSection("styles", required: false)

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

</head>

<body>

<div class="navbar navbar-fixed-top">

<div class="navbar-inner">

<div class="container">

<ul class="nav nav-pills pull-right">

@Html.Action("UserLogin", "Home")

</ul>

</div>

</div>

</div>

@RenderBody()

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

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

@RenderSection("scripts", required: false)

</body>

</html>

Что как тут происходит:

1. Мы получаем request и по запросу мы определяем маршрут /Default/Home/Index.
2. У данного контроллера/метода есть стандартный View – /Home/Index.cshtml

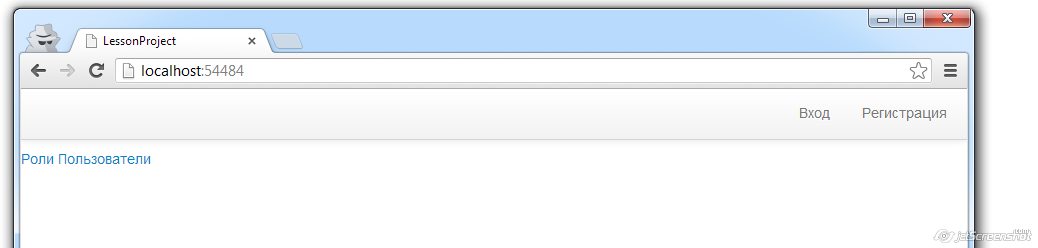
В начале файла объявляется, что он будет включен в Layout:

@{

ViewBag.Title = "LessonProject";

Layout = "~/Areas/Default/Views/Shared/\_Layout.cshtml";

}

Указанный layout выведет данные с помощью @RenderBody(). Запускаем:  


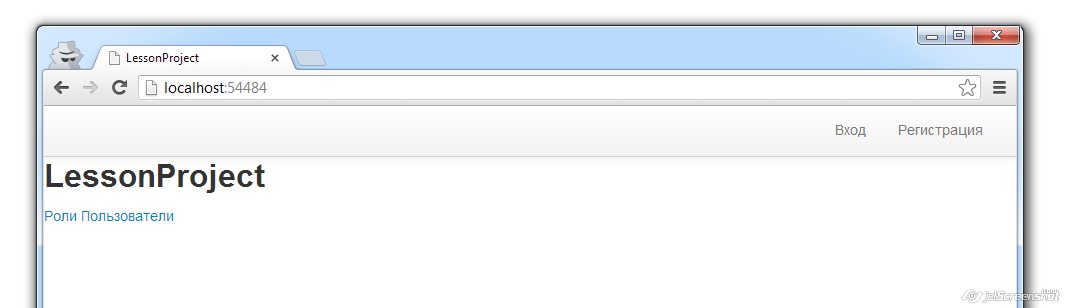
Видно, что body заехал под панель навигации. В нашем файле переопределяем body:

body

{

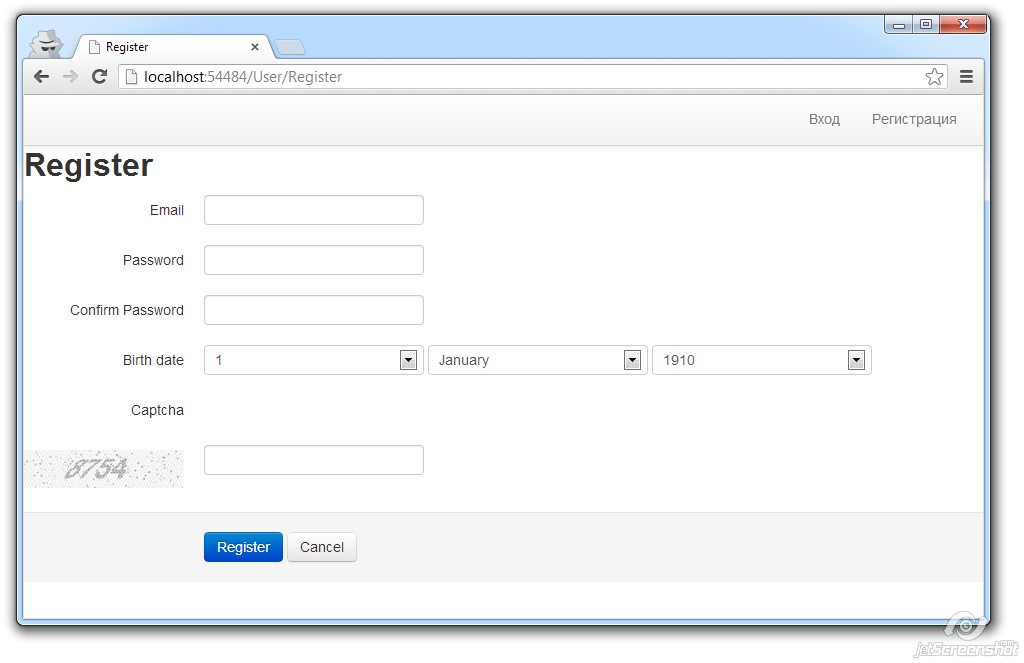
padding-top : 40px !important;

}



Гораздо лучше.

Ок, так как мы использовали классы для формы bootstap ранее, то регистрация у нас выглядит теперь так:



Поправим выбор даты рождения, добавим классы в @Html.DropDownList() (/Areas/Default/Views/User/Register.cshtml):

…

<div class="controls">

@Html.DropDownList("BirthdateDay", Model.BirthdateDaySelectList**, new { @class = "select-day" }**)

@Html.DropDownList("BirthdateMonth", Model.BirthdateMonthSelectList**, new { @class = "select-month" }**)

@Html.DropDownList("BirthdateYear", Model.BirthdateYearSelectList**, new { @class = "select-year" }**)

</div>

Так как вероятно, мы еще где-то будем использовать эту конструкцию по выбору даты (хотя не факт), то это более общее, чем частное (которое относится именно к регистрации):

.select-day

{

width : 50px;

}

.select-month

{

width : 90px;

}

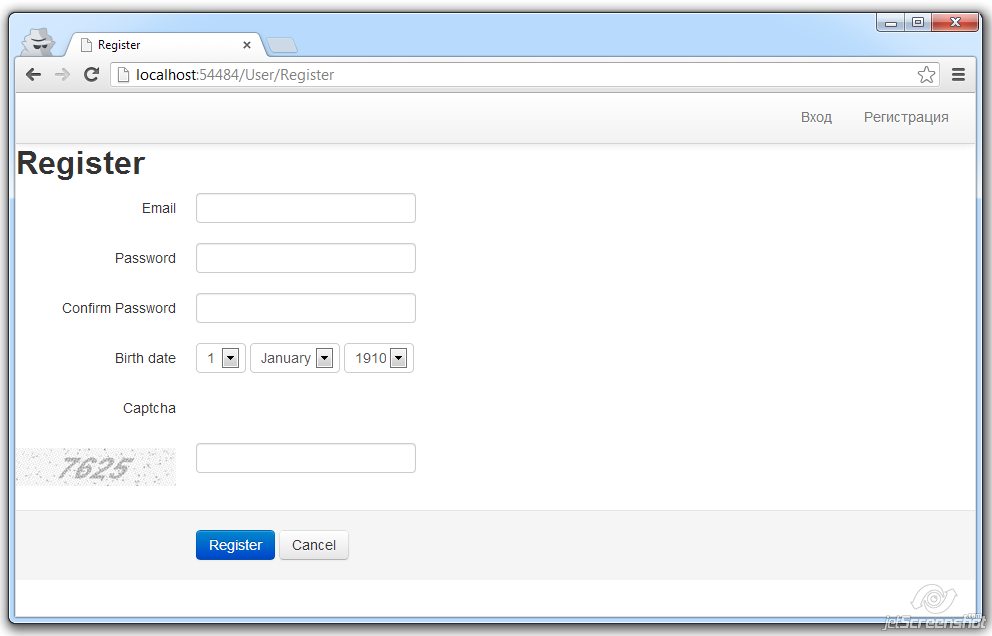
.select-year

{

width : 70px;

}

Проверяем:



Уиии!

### Структура js файлов

Теперь перейдем к описанию js файлов. Мы используем jquery как основной фреймворк по работе с клиентской частью кода. Один наш пользовательский js файл (назовем его /Scripts/common.js) будет вызываться всегда. В него будут добавлены те функции, которые будут присутствовать на любой странице. Остальные js-файлы будут вызываться опционально.

Чтобы не путаться создадим 2 папки «admin» и «default» в /Scripts.

Все файлы будут иметь уникальные имена, которые будут записаны в SmallCase формате, и будут относиться к определенной странице (в основном). Например: user-register.js – файл который будет включен в страницу User/Register.cshtml:

@section scripts {

@Scripts.Render("/Scripts/default/user-register.js")

}

Эта секция выведется в тот месте где описана в \_Layout.cshtml:

…

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

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

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

</body>

В /App\_Start/BundleConfig.cs тем временем добавим описание:

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

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

Все пользовательские js классы, за исключением плагинов будут иметь следующую структуру:

function FunctionName() {

\_this = this;

this.init = function () {

/\* установка обработки нажатий или иных манипуляций \*/

$("button").click(function () {

var id = $(this).attr("id");

\_this.saySomething(id);

});

}

/\* другие публичные методы\*/

this.saySomething = function (id) {

alert("Пыщь-пыщь! : " + id);

}

/\* другие приватные методы \*/

function saySomething (id) {

alert("Пыщь-пыщь! Но тссс!: " + id);

}

}

var functionName = null;

$().ready(function () {

functionName = new FunctionName();

functionName.init();

});

Рассмотрим подробнее:

1. function FunctionName имеет имя в стиле UpperCamelCase по имени файла (Common и UserRegister)
2. \_this = this; Сохранение ссылки на данную функцию, чтобы была возможность использовать внутри delegate-функций
3. this.init – внешняя (public) функция, где будет происходить инициализация обработки.
4. var functionName = null – создание глобальной переменной. Возможно использовать из других файлов.
5. $().ready() – вызывается после того, как сформирована DOM-структура. JQuery функция.
6. functionName = new FunctionName(); - создаем объект класса.
7. functionName.init(); - инициализируем его.

### Минификация ресурсных файлов

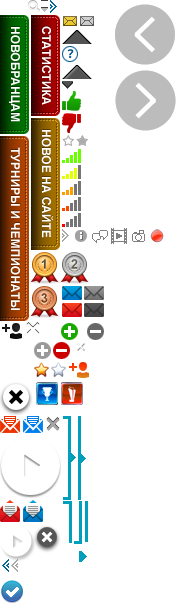
Так как ресурсные файлы, со временем всё больше вырастают, и с другой стороны идет развитие мобильного интернета для мобильных устройств, в которых количество передаваемых данных играет не последнюю роль. Минификация получения страницы сводится к следующим вещам:

* Уменьшение кол-ва запросов к ресурсным файлам
* Уменьшение ресурсных файлов.

Уменьшение кол-ва запросов к изображениям, особенно маленьким. Это делается двумя способами:

1. Использование спрайта. На большой холст добавляется много элементов оформления сразу. После чего в css прописывают смещение и ширину\высоту части этого холста.

Например рисунок:



И его использование в html:

<div class="label label-new sprite"></div>

И css:

.sprite

{

background: url("/Media/images/sprite.png");

overflow: hidden;

text-indent: -9999px;

}

.box .label

{

position: absolute;

width: 29px;

right: -29px;

top: 35px;

}

.box .label-new

{

background-position: 0 -15px;

height: 119px;

}

1. Использовать для очень маленьких изображений (иконок) объявления напрямую в css c помощью base64.

Для этого картинку переводят в base64 например на <http://webcodertools.com/imagetobase64converter>

После чего добавляют в css по типу:

background-image: url(data:image/png;base64,);

Конечно, при загрузке контентных иконок или изображений данный способ не применим.

Для css и js применяется минификация файла, т.е. убираются пробелы и разделители и используются более краткие локальные переменные. Файл или изначально подготавливается минифицированным как для jquery библиотеки или минифицируется на сервере.

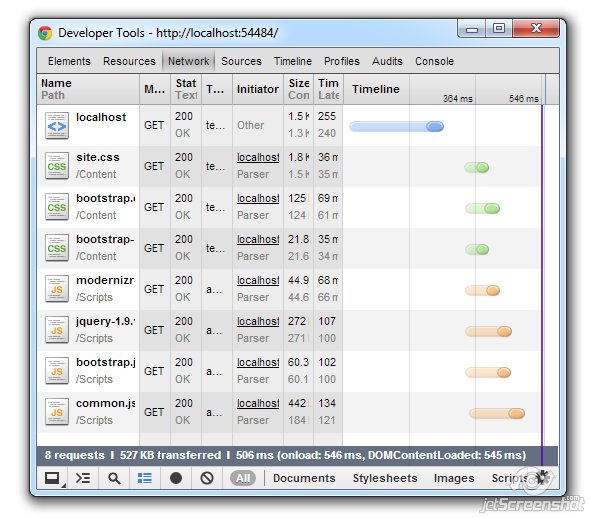
Для включения минификации необходимо в Web.config файле изменить директиву compilation:

<compilation debug="false" targetFramework="4.5" />

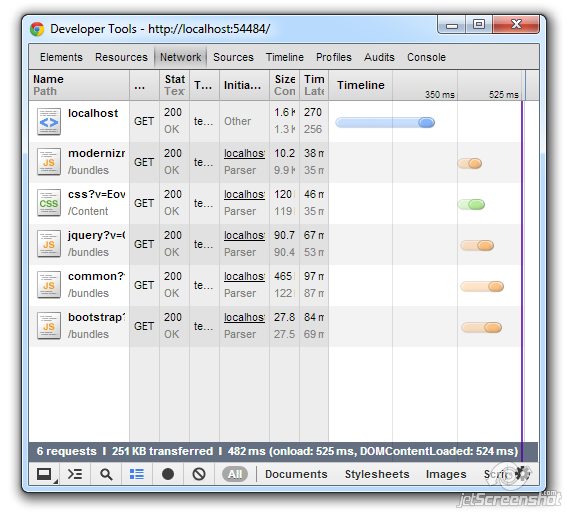
Или напрямую включить в /App\_Start/BundleConfig.cs:

BundleTable.EnableOptimizations = true;

Проверим:



До оптимизации 527 КБ



После оптимизации 251 КБ

Т.е. больше, чем в 2 раза. На самом деле эта величина может быть как меньше, так и больше в зависимости какие ресурсы мы грузим. Если есть кеширование, то страница при дальнейшей работе загружает только незначительную часть новых файлов.

### Установка jQuery

Изначально jquery уже установлен, но если фреймворк обновился, а они это делают часто, то очевидно, что необходимо обновить его:

Install-Package JQuery

Далее, мы до этого убрали JQueryUI (<http://jqueryui.com/>), так как собираемся datepicker, modal использовать то что предлагает bootstrap, но несколько вещей в JQueryUI есть, которые нам в принципе пригодятся, это элементы Interaction. Установим только их выборочно:

1. Так как Install-Package JQuery.UI.Interactions не прокатило, так как jquery нужна версии < 1.6 то установим вручную.
2. Выберем custom скачивание с сайта <http://jqueryui.com/download/>
3. Выбираем только Сore и Interactions
4. Скачиваем
5. Переносим css файлы в /Content/css
6. Переносим js файлы в /Scripts (jquery-1.9.1-min.js переносить нет необходимости)
7. Подключаем в BundleConfig

bundles.Add(new ScriptBundle("~/bundles/jqueryui")

.Include("~/Scripts/jquery-ui-1.\*"));

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

.Include("~/Content/jquery-ui-1\*"));

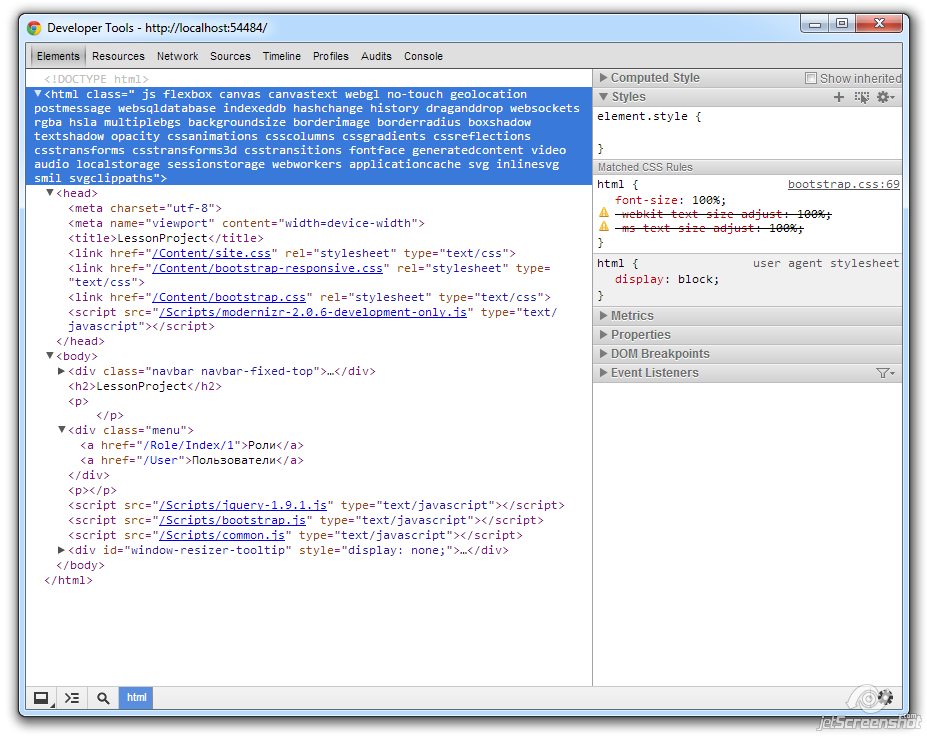
1. При необходимости объявляем на страницах.
2. Готово!

### Firebug (Firefox) и Developer Tool (Chrome)

Для удобства отладки в Firefox есть расширение Firebug, а в Chrome есть встроенный механизм отладки Developer Tool. Я рассмотрю пример с Developer Tool. Вызывается по нажатию клавиш –Shift-Ctrl-I.

Изучим его немного:

* Вкладка Elements.



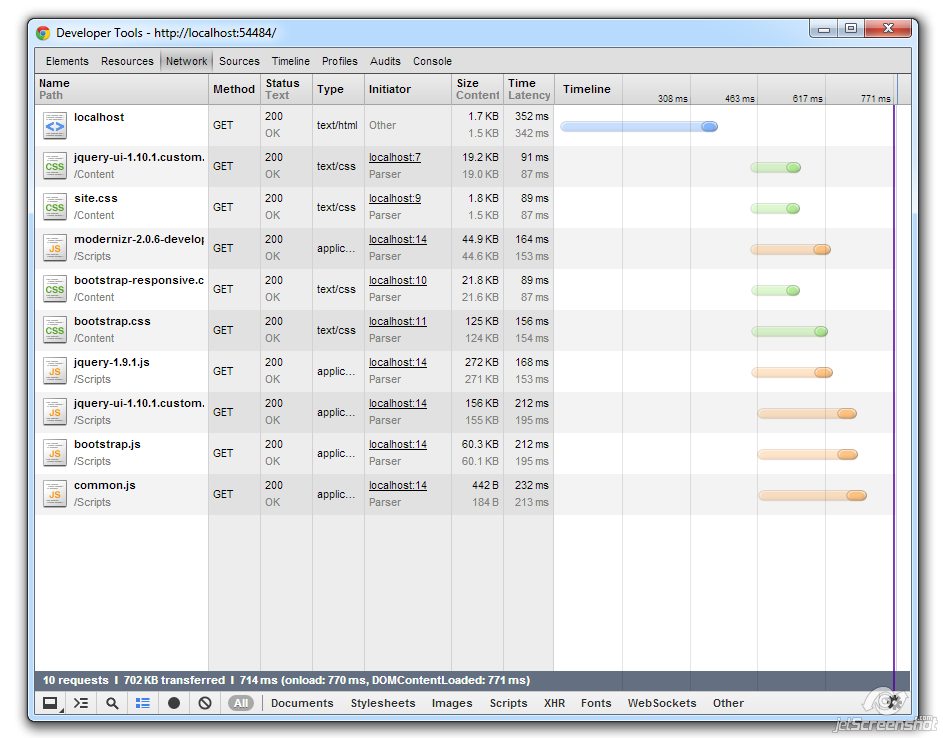
Здесь мы в левой части можем увидет DOM – дерево элементов. В правой части стили. Стили, как и элементы можно редактировать прямо на ходу, и изменения будут тут же отображаться в браузере в редактируемой странице. Очень удобно когда надо поправить расположение объектов на несколько пикселей.

* Вкладка Resources



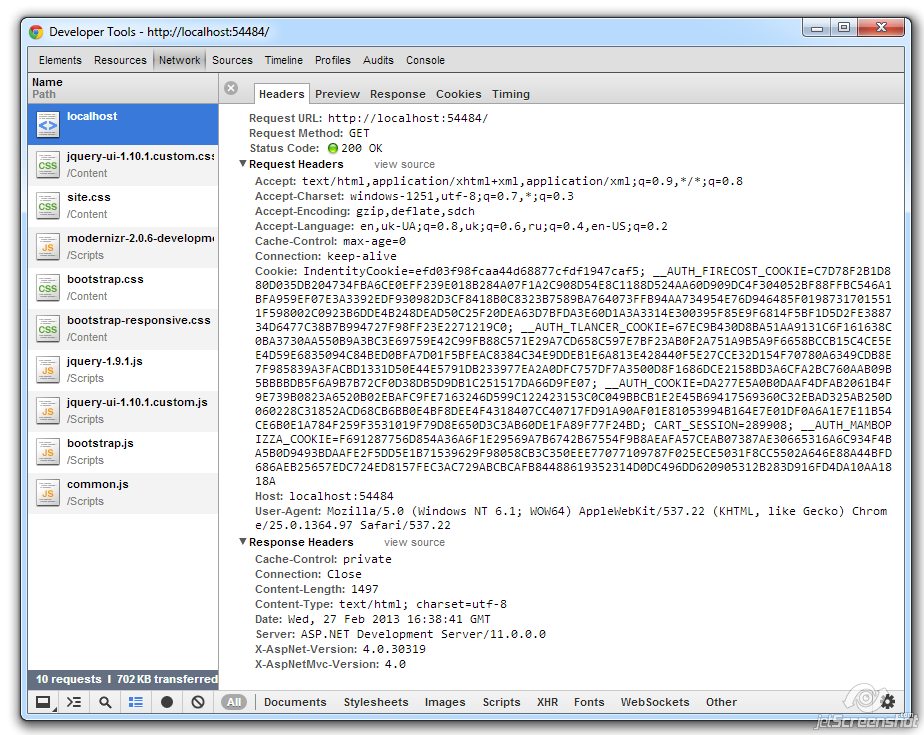
В левой части располагаются все ресурсы, которые были загружены. А в правой части – можно просмотреть эти ресурсы

* Вкладка Network



Показывает запросы и тайминг этих запросов. Обозначает разными цветами document, css, js файлы, изображениея. Показывает размеры файлов.

При клике на запрос можно подробнее рассмотреть HTTP-запрос, например:

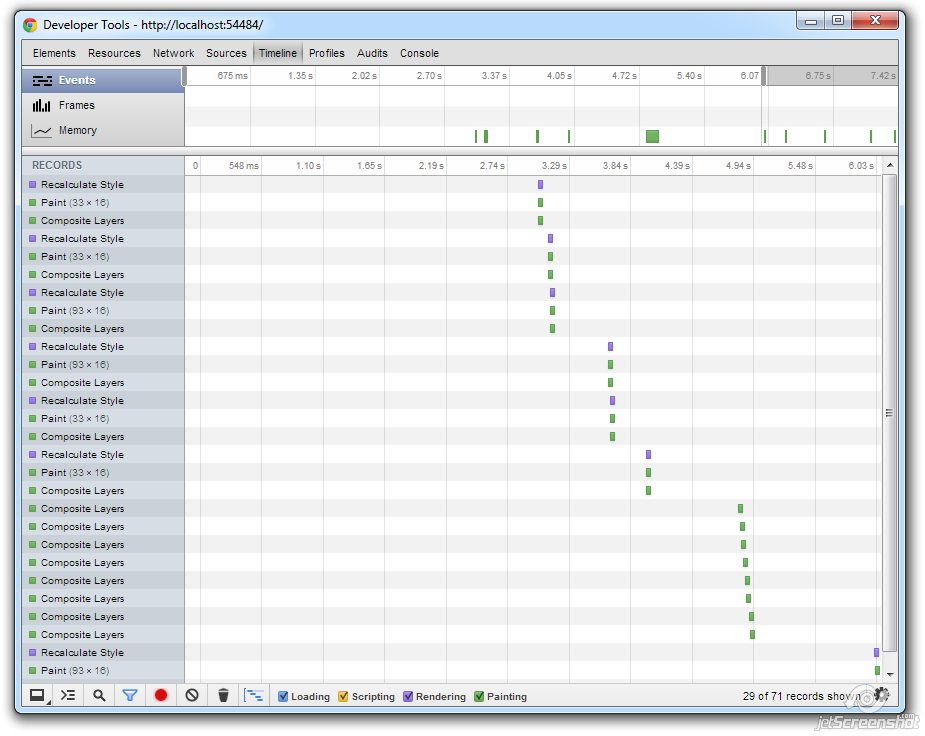


* Вкладка Sources



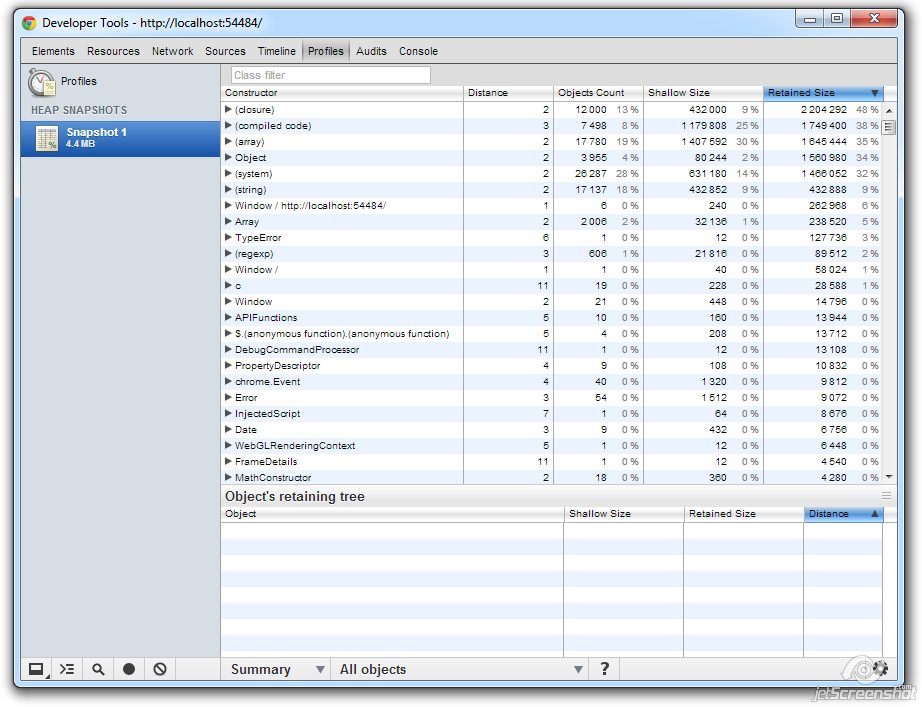
Одна из основных вкладок. Тут можно просматривать js файлы, устанавливать точки останова, и выполнять отладку приложения. Горячие клавиши F9, F10, F11 как в VisualStudio для отладки.

* Вкладка Timeline



Показывает происходящие в браузере события. Я ее никогда не использовал.

* Вкладка Profiles



Для профилирования, т.е. нахождения вообще нетривиальных ошибок. Также никогда не использовал.

* Вкладка Audits



На этой вкладке можно проверить страницу на оптимизацию. Т.е. браузер рекомендует сделать некоторые действия, которые приведут к уменьшению кода передаваемого от сервера, сжатия, удаление лишних строк css.

* Вкладка Console



Одновременно и командная строка и окно вывода протоколирования. Для вывода лога нужно вызвать команду console.log(“message”).

*Осторожно используйте эту команду в IE, так как когда консоль не открыта, то выдает ошибку.*

### Селекторы и обход

JQuery – это тот инструмент, который помогает нам разрабатывать клиентский код под разные браузеры. К тому же является простой и логичной библиотекой.

В основе всего лежить селектор. Селектор позволяет выбрать множество элементов находящихся в DOM (document object model) и произвести над ними действия, такие как – назначить обработчик события, изменить местопложение, изменить атрибуты, удалить выбранные элементы, добавить в выбранные элементы текст или html, создать объект.

Основное правило пишется как:

$([“правило селектора”][, область выбора])

Если область выбора не задана, то ищется по всему документу: $(document). Это корневой узел всего DOM.

Если область выбора задана, то ищется только в границе этого узла.

Правило селектора задается по принципам назначения css свойств:

* $(“div”) – выбор всех div-элементов
* $(“.class”) – выбор всех элементов с имеющимся классом class
* $(“.class .class1”) – выбор всех элементов с имеющимся классом class1 содержащихся в классе class
* $(“.class.class1”) – выбор всех элементов с имеющимся классом class1 и class
* $(“#Id1”) – выбор элемента (одного) с id =Id1
* $(“[type=’password’]”) – выбор элемента с атрибутом type=’password’
* $(“div”, $(“#MainPopup”)) – выбор всех элементов div содержащихся в элементе с id=MainPopup
* $(“input[type=’checkbox’]:checked”) – выбор элементов ввода типа checkbox, которые отмечены галочкой.

Для проверки найден или нет элемент, можно использовать свойство length:

if($("#Id1").length == 0)

{

alert("Элемент не найден")

}

Для продвижения вверх по дереву элементов от выбранного можно использовать функции closest(), parent() или parents():

* parent() – возвращает непосредственного родителя (и только его)
* parents(selector) – возвращает множество всех родителей (если не указан селектор вплоть до body и html)
* closest(selector) – возвращает ближайшей элемента, который соотвествует селектору, причем сам элемент может быть этим элементом.

### События

Для обработки событий мы назначаем события на элементы селектора. Например:

$(".button").click(function () {

alert("Button clicked");

});

Какие есть события:

* События браузера
  + .error() – произошла ошибка выполнения скрипта в браузере
  + .resize() – изменился размер контейнера, к которому данное событие было назначено.
  + .scroll() – контент внутри контейнера «проскроллили» (тут надо понимать автоматический скроллинг, т.е. если у элемента стиль назначен overflow:scroll и содержимое проскроллили).
* Загрузки документа
  + .load() – загружен элемент. Например, при изменении (назначении) src у тега img.
  + .ready() – вызывается при окончании загрузки DOM документа. Мы его используем постоянно при инициализации классов.
  + .uload() – вызывается, когда пользователь хочет закрывает страницу.
* События формы
  + .blur() – при потере фокуса у поля ввода
  + .change() – при изменении выбора у выпадающего списка или списка множественного выбора
  + .focus() – при приобретении фокуса у поля ввода
  + .select() – при выделении текста или части текста в поле ввода
  + .submit() – при подтверждении формы ввода
* События клавиатуры
  + .focusin() – при приобретении фокуса (аналог focus)
  + .focusout() – при потере фокуса (аналог blur)
  + .keydown() – при нажатии клавиши в момент пока клавиша нажата
  + .keypress() – при нажатии и отпускании клавиши
  + .keyup() – при отпускании клавиши
* События мыши
  + .click() – при клике левой кнопки мыши на элементе
  + .dblclick() – при двойном клике мыши
  + .hover() – при наведении мыши
  + .mousedown(), .mouseup(), .mouseenter(), .mouseleave(), .mousemove(), .mouseout(), .mouseover() – все эти события реагируют на соотвествующие действия мыши относительно элементов.

Надо помнить, что события не распространяются на вновь созданные элементы, хотя и подпадающие под выбор селектора, который выполнялся ранее. Но если мы по второму разу проинициализируем, то на элементах где уже была назначена обработка события, событие будет выполняться дважды.

Для задания постоянной глобальной обработки нужно использовать следующую конструкцию:

$(document).on("click", ".button", function () {

alert("Button clicked");

});

### Атрибуты и значения

Для работы с атрибутами элемента в основном используются функции:

* .val(value) – задать значение атрибута value, для конструкции select\option – выбирает элемент
* .val() – получить значение атрибута value, или значение, если это поле ввода
* .attr(“attr”) - получить значение атрибута attr
* .attr(“attr”, “value”) – задать значение value атрибуту attr
* .data(“id”) – получить значение атрибута data-id
* .data(“id”, “20”) – задать значение 20 атрибуту data-id
* .css(“width”, “120px”) – задать значение стиля width:120px
* .css(“width”) – получить значение width
* .addClass()/.removeClass()/.toggleClass() – добавить/удалить/переключить класс в элементах селектора

### Основные манипуляции

Для работы с элементами рассмотрим следующие функции:

* .show()\.hide() – показать\скрыть
* .html(htmltext) – в элементах innerHtml задать html текст
* .text(text) – в тексте элементов задать text
* .empty() – очистить элементы селектора
* .remove() – удалить элементы селектора
* .append() – добавить html-текст или элементы селектора после всех своих листьев
* .prepend() - добавить html-текст или элементы селектора перед всеми своими листьями
* .after()/.before() – вставить html-текст или элементы после элемента/перед элементом
* .appendTo()/.prependTo() – добавить выбранный элемент в конец/начало листьев заданного элемента

### Ajax

Рассмотрим основную и главную функцию (в 99% случаев я обходился только ей).

$.ajax({

type: "GET",

url: "/ajaxUrl",

data: { id: id },

beforeSend: function () {

/\* что-то сделать перед \*/

},

success: function (data) {

/\* обработать результат \*/

},

error: function () {

/\* обработать ошибку \*/

}

});

Есть и другие параметры, но к ним прибегать стоит в нескольких случаях.

Рассмотрим подробнее параметры:

* **type.** GET или POST запрос.
* **url.** Если его не задать, или будет по умолчанию, то ajax запрос будет отправлен на текущую страницу
* **data.** В формате json или get-подобная строка, типа «id=1&value=2». Можно использовать серилизацию формы:

data: $("form").serialize()

При этом надо помнить, что при передаче множества одинаковых значений чекбоксов нужно устанавливать параметр

traditional : true

* **beforeSend.** Событие генерирующеся перед непосредственно отправкой формы.
* **success.** Событие, которое обозначает, что всё хорошо и в data содержится результат выполнения
* **error.** Событие, которое возникает, если ответ от сервера был отличный от 200 OK.

### Ajax-login форма.

Куча теории, пора бы и к практике переходить. Создадим вторую форму входа, которая будет способствовать быстрому входу на сайте. При клике на «Вход» мы переходим не на страницу Входа, а выскакивает попап окошко с предложением ввести логин прямо сейчас. При ошибочном вводе, форма выдает предупреждение. Обычную форму по адресу /Login оставляем, она нам понадобится.

Попап формы могут использоваться часто, так что будем считать это стандартной процедурой – вызвать Popup по адресу такому-то. Так как попап – всегда один, то создадим для него контейнер в \_Layout.cshtml (/Areas/Default/Views/Shared/\_Layout.cshtml):

<div id="PopupWrapper"></div>

Добавим функциональность в common.js (/Scripts/common.js):

this.showPopup = function (url, callback)

{

$.ajax({

type: "GET",

url: url,

success: function (data)

{

$(".modal-backdrop").remove();

var popupWrapper = $("#PopupWrapper");

popupWrapper.empty();

popupWrapper.html(data);

var popup = $(".modal", popupWrapper);

$(".modal", popupWrapper).modal();

callback(popup);

}

});

}

, где .modal() – это функция из bootstrap.js.

Так как Вход у нас на каждой странице то следущую функциональность добавляем тоже в common.js:

this.init = function () {

$("#LoginPopup").click(function () {

\_this.showPopup("/Login/Ajax", function (modal)

{

});

});

}

Добавим в контроллере обработчик (/Areas/Default/Controller/LoginController.cs):

[HttpGet]

public ActionResult Ajax()

{

return View(new LoginView());

}

[HttpPost]

public ActionResult Ajax(LoginView loginView)

{

if (ModelState.IsValid)

{

var user = Auth.Login(loginView.Email, loginView.Password, loginView.IsPersistent);

if (user != null)

{

return RedirectToAction("Index", "Home");

}

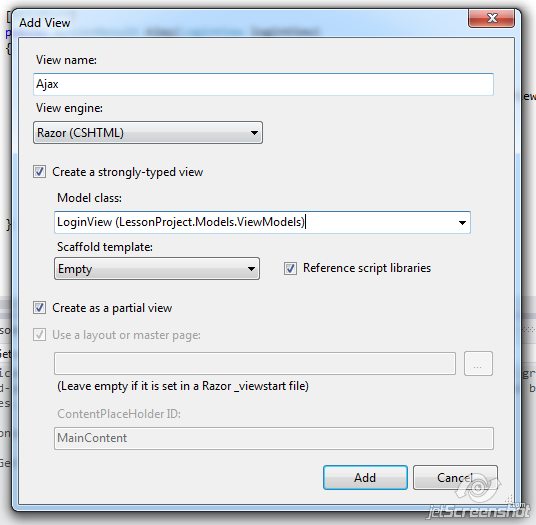
ModelState["Password"].Errors.Add("Пароли не совпадают");

}

return View(loginView);

}

Он полностью аналогичен Index, только будет вызываться другой View – «Ajax», создадим его (/Areas/Default/Views/Login/Ajax.cshtml):



@model LessonProject.Models.ViewModels.LoginView

<div class="modal hide fade" tabindex="-1" role="dialog" aria-hidden="true">

<div class="modal-header">

<button type="button" class="close" data-dismiss="modal" aria-hidden="true">×</button>

<h3 id="myModalLabel">Login</h3>

</div>

<div class="modal-body">

@using (Html.BeginForm("Ajax", "Login", FormMethod.Post, new { @class = "form-horizontal", **id = "LoginForm"** }))

{

<fieldset>

<legend>Вход</legend>

<div class="control-group">

<label class="control-label" for="Email">

Email</label>

<div class="controls">

@Html.TextBox("Email", Model.Email, new { @class = "input-xlarge" })

<p class="help-block">Введите Email</p>

@Html.ValidationMessage("Email")

</div>

</div>

<div class="control-group">

<label class="control-label" for="Password">

Пароль</label>

<div class="controls">

@Html.Password("Password", Model.Password, new { @class = "input-xlarge" })

@Html.ValidationMessage("Password")

</div>

</div>

</fieldset>

}

</div>

<div class="modal-footer">

<button class="btn" data-dismiss="modal" aria-hidden="true">Close</button>

<button class="btn btn-primary" **id="LoginButton"**>Login</button>

</div>

</div>

Обратите внимание на id формы LoginForm и id кнопки LoginButton

Изменим вызов в UserLogin.cshtml (/Areas/Default/Views/Home/UserLogin.cshtml):

<li><span class="btn btn-link" id="LoginPopup">Вход</span></li>

В common.js добавим обработку LoginButton, при вызове установим обработчик события на $(“#LoginButton”).click(…) (/Scripts/common.js):

this.init = function () {

$("#LoginPopup").click(function () {

\_this.showPopup("/Login/Ajax", initLoginPopup);

});

}

…

function initLoginPopup(modal) {

$("#LoginButton").click(function () {

$.ajax({

type: "POST",

url: "/Login/Ajax",

data : $("#LoginForm").serialize(),

success: function (data) {

showModalData(data);

initLoginPopup(modal);

}

});

});

}

function showModalData(data, callback) {

$(".modal-backdrop").remove();

var popupWrapper = $("#PopupWrapper");

popupWrapper.empty();

popupWrapper.html(data);

var popup = $(".modal", popupWrapper);

$(".modal", popupWrapper).modal();

if (callback != undefined) {

callback(popup);

}

}

Обратите внимание на рекурсивный вызов initLoginPopup. И тут заключается дилемма. Так как при удачном входе, нам не надо чтобы в PopupWrapper грузилась новая страница (или страница с ошибкой), а только чтобы страница обновилась.

Для этого сделаем хитрость. В /Areas/Default/Views/Shared/ добавим \_Ok.cshtml. Суть которого будет перезагружать страницу:

<script>

window.location.reload();

</script>

При удачном входе мы загружаем этот View. При добавлении в дерево DOM в строке

popupWrapper.html(data);

скрипт запустится и перезагрузит страницу не дожидаясь остальных вызовов. Изменим контроллер (/Areas/Default/Controllers/LoginController.cs):

var user = Auth.Login(loginView.Email, loginView.Password, loginView.IsPersistent);

if (user != null)

{

return View("\_Ok");

}

Проверяем, работает!

### Итог

Мы рассмотрели основные принципы верстки и клиентской части, это только лишь малая толика того, что вообще можно знать о верстке, стилях и программировании в клиентской части.

Мы научились пользоваться отладчиком в Chrome и создавать ajax запрос. Подробнее рассмотрите этот вопрос в дальнейшем.

Полезные ссылки:

<http://jquery.com>

<http://habrahabr.ru/post/161895/>

<http://habrahabr.ru/post/154687/>

<http://twitter.github.com/bootstrap/>

<http://habrahabr.ru/post/160177/>

## Урок 8. View, Razor, страница ошибки.

Цель: Научиться делать вывод данных в html, использование Razor. Helperы. PageableData. Динамические формы. RedirectToLogin, RedirectToNotFoundPage. Страница ошибки. RssActionResult.

### Основа

Итак, как устроена часть View.

В контроллере все action-методы возвращают тип ActionResult. И для вывода результата мы используем:

return View(modelData);

Основными параметрами View может быть:

* Имя, обычно оно совпадает с именем action-метода. В случае если надо вызвать иной по имени View, то используется конструкция return View(“ViewName”, modelData).
* Данные для отображения во View. Необязательный параметр. При передаче во View этот объект данных будет обозначаться Model. Для связывания типа данных во View указывается ожидаемый тип данных:

@model LessonProject.Model.User

* Layout. Необязательный параметр. При указании этого параметра по данной строке найдется страница-контейнер и вызовется. View-часть будет обработана методом RenderBody()

Layout = "~/Areas/Default/Views/Shared/\_Layout.cshtml"

В \_Layout.cshtml:

</div>

@RenderBody()

<div id="PopupWrapper"></div>

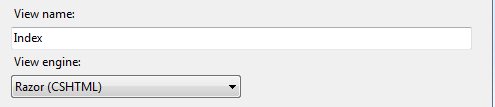
Выбор, какой же View использовать происходит следующим образом:

* Ищется в папке /Areas/[Area]/Views/[ControllerName]/
* Ищется в папке /Areas/[Area]/Views/Shared/
* Ищется в папке /Views/[ControllerName]/
* Ищется в папке /Views/Shared/

Ок, приступим к изучению.

### Razor

При создании View есть два выбора движка, это ASPX и Razor. Aspx мы не будем рассматривать и использовать в дальнейшем. О нем и поговорим.



ASPX был громозким движком с тегами <% %> для выполнения кода и <%: %> для вывода данных.

Razor использует конструкцию @Model.Name. Т.е. всё, что начинается с @ переводит в режим или исполнения кода или вывода данных @foreach() {…}, или @if() { … } else { … }:

@if (Model.Any())

{

<p>Список</p>

}

@foreach (var role in Model)

{

<div class="item">

<span class="id">

@role.ID

</span>

<span class="name">

@role.Name

</span>

<span class="Code">

@role.Code

</span>

</div>

}

Внутри { } находятся теги – это дает знак, что это шаблон. Для простого выполнения кода внутри шаблона используем структуру @{ code }, для корректного вывода данных внутри атрибутов или текстом конструкция @(string result):

@{

int i = 0;

}

@foreach (var role in Model)

{

<div class="item @(i % 2 == 0 ? "odd" : "")">

<span class="id">

@role.ID

</span>

<span class="name">

@role.Name

</span>

<span class="Code">

@role.Code

</span>

</div>

i++;

}

Чтобы вывести не теговый текст нужно использовать псевдотеги <text></text>:

@foreach (var role in Model)

{

@role.Name<text>, </text>

}

Для вывода html-текста – или должна возвращаться MvcHtmlString или использовать конструкцию @Html.Raw(html-string-value), иначе текст будет выведен с экранированием тегов.

### PageableData

Рассмотрим постраничный вывод таблицы из БД. Проанализируем:

1. Контроллер должен в параметрах получить значение страницы, которую мы будем выводить
2. По умолчанию это будет первая страница
3. При выводе мы должны знать
   1. Список элементов БД, которые выводим
   2. Количество страниц
   3. Текущую страницу

Создадим Generic-класс PageableData (/Models/Info/PageableData.cs):

public class PageableData<T> where T : class

{

protected static int ItemPerPageDefault = 20;

public IEnumerable<T> List { get; set; }

public int PageNo { get; set; }

public int CountPage { get; set; }

public int ItemPerPage { get; set; }

public PageableData(IQueryable<T> queryableSet, int page, int itemPerPage = 0)

{

if (itemPerPage == 0)

{

itemPerPage = ItemPerPageDefault;

}

ItemPerPage = itemPerPage;

PageNo = page;

var count = queryableSet.Count();

CountPage = (int)decimal.Remainder(count, itemPerPage) == 0 ? count / itemPerPage : count / itemPerPage + 1;

List = queryableSet.Skip((PageNo - 1) \* itemPerPage).Take(itemPerPage);

}

}

По умолчанию кол-во выводимых значений на странице – 20, но мы можем изменить этот параметр в конструкторе. Передаем IQueryable<T> и вычисляем кол-во страниц CountPage. Используя PageNo выбираем страницу:

List = queryableSet.Skip((PageNo - 1) \* itemPerPage).Take(itemPerPage);

В контроллере используем:

public class UserController : DefaultController

{

public ActionResult Index(int page = 1)

{

var data = new PageableData<User>(Repository.Users, page, 30);

return View(data);

}

…

Во View используем данный класс:

@model **LessonProject.Models.Info.PageableData<LessonProject.Model.User>**

@{

ViewBag.Title = "Users";

Layout = "~/Areas/Default/Views/Shared/\_Layout.cshtml";

}

<h2>Users</h2>

<p>

@foreach (var user in Model**.List**)

{

<div class="item">

<span class="id">

@user.ID

</span>

<span class="email">

@user.Email

</span>

<span class="activateDate">

@user.AddedDate

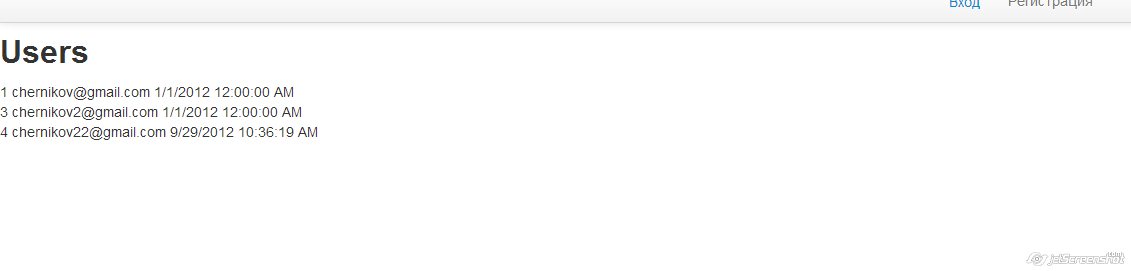
</span>

</div>

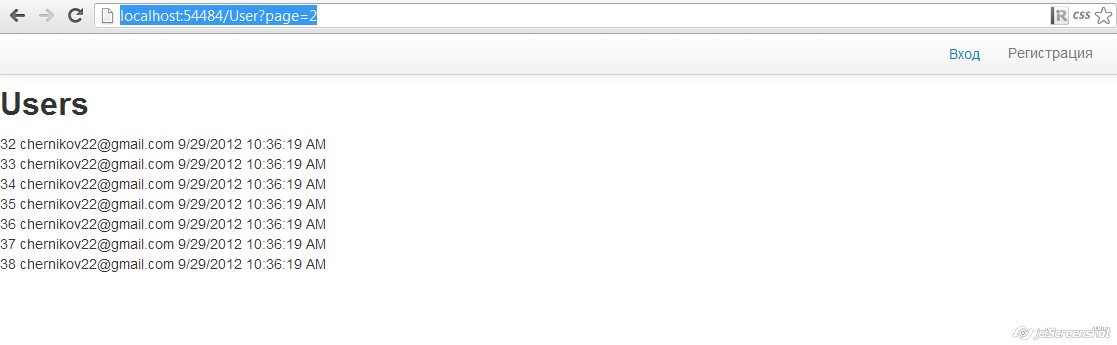
}

</p>

Запускаем – проверяем (<http://localhost:54484/User>)



Для продолжения сгенерируем несколько больше данных (просто ctrl-c, ctrl-v в таблице в Server Explorer)



Перейдем к созданию Helperа пагинатора, который нам даст возможность пролистывать этот список.

### Helper (PagerHelper)

Так как мы используем bootstrap, то и на базе его будем делать пагинатор. По коду он выглядит так:

<div class="pagination">

<ul>

<li><a href="#">Prev</a></li>

<li><a href="#">1</a></li>

<li><a href="#">2</a></li>

<li><a href="#">3</a></li>

<li><a href="#">4</a></li>

<li><a href="#">5</a></li>

<li><a href="#">Next</a></li>

</ul>

</div>

Нас интересует только внутренняя часть <ul></ul>.

Helper создается как Extension для класса System.Web.Mvc.HtmlHelper. План таков:

* Вывести Prev (сделать активным если надо)
* Вывести ссылки на первые три страницы 1, 2, 3
* Вывести троеточие, если необходимо
* Вывести активной ссылку текущей страницы
* Вывести троеточие, если необходимо
* Вывести последние три страницы
* Вывести Next (сделать активной если надо)
* Заключить всё в ul и вывести как MvcHtmlString

Код будет выглядеть так:

public static MvcHtmlString PageLinks(this HtmlHelper html, int currentPage, int totalPages, Func<int, string> pageUrl)

{

StringBuilder builder = new StringBuilder();

//Prev

var prevBuilder = new TagBuilder("a");

prevBuilder.InnerHtml = "&laquo;";

if (currentPage == 1)

{

prevBuilder.MergeAttribute("href", "#");

builder.AppendLine("<li class=\"active\">" + prevBuilder.ToString() + "</li>");

}

else

{

prevBuilder.MergeAttribute("href", pageUrl.Invoke(currentPage - 1));

builder.AppendLine("<li>" + prevBuilder.ToString() + "</li>");

}

//По порядку

for (int i = 1; i <= totalPages; i++)

{

//Условие что выводим только необходимые номера

if (((i <= 3) || (i > (totalPages - 3))) || ((i > (currentPage - 2)) && (i < (currentPage + 2))))

{

var subBuilder = new TagBuilder("a");

subBuilder.InnerHtml = i.ToString(CultureInfo.InvariantCulture);

if (i == currentPage)

{

subBuilder.MergeAttribute("href", "#");

builder.AppendLine("<li class=\"active\">" + subBuilder.ToString() + "</li>");

}

else

{

subBuilder.MergeAttribute("href", pageUrl.Invoke(i));

builder.AppendLine("<li>" + subBuilder.ToString() + "</li>");

}

}

else if ((i == 4) && (currentPage > 5))

{

//Троеточие первое

builder.AppendLine("<li class=\"disabled\"> <a href=\"#\">...</a> </li>");

}

else if ((i == (totalPages - 3)) && (currentPage < (totalPages - 4)))

{

//Троеточие второе

builder.AppendLine("<li class=\"disabled\"> <a href=\"#\">...</a> </li>");

}

}

//Next

var nextBuilder = new TagBuilder("a");

nextBuilder.InnerHtml = "&raquo;";

if (currentPage == totalPages)

{

nextBuilder.MergeAttribute("href", "#");

builder.AppendLine("<li class=\"active\">" + nextBuilder.ToString() + "</li>");

}

else

{

nextBuilder.MergeAttribute("href", pageUrl.Invoke(currentPage + 1));

builder.AppendLine("<li>" + nextBuilder.ToString() + "</li>");

}

return new MvcHtmlString("<ul>" + builder.ToString() + "</ul>");

}

Добавим Namespace LessonProject.Helper в объявленные во View. Это можно сделать двумя способами:

* В самом View

@using LessonProject.Helper;

* В Web.config (рекомендуется)

<configSections>

…

<sectionGroup name="system.web.webPages.razor" type="System.Web.WebPages.Razor.Configuration.RazorWebSectionGroup, System.Web.WebPages.Razor, Version=2.0.0.0, Culture=neutral, PublicKeyToken=31BF3856AD364E35">

<section name="host" type="System.Web.WebPages.Razor.Configuration.HostSection, System.Web.WebPages.Razor, Version=2.0.0.0, Culture=neutral, PublicKeyToken=31BF3856AD364E35" requirePermission="false" />

<section name="pages" type="System.Web.WebPages.Razor.Configuration.RazorPagesSection, System.Web.WebPages.Razor, Version=2.0.0.0, Culture=neutral, PublicKeyToken=31BF3856AD364E35" requirePermission="false" />

</sectionGroup>

</configSections>

+

<system.web.webPages.razor>

<pages pageBaseType="System.Web.Mvc.WebViewPage">

<namespaces>

<add namespace="LessonProject.Helper" />

</namespaces>

</pages>

</system.web.webPages.razor>

Добавляем пагинатор во View:

<div class="pagination">

@Html.PageLinks(Model.PageNo, Model.CountPage, x => Url.Action("Index", new {page = x}))

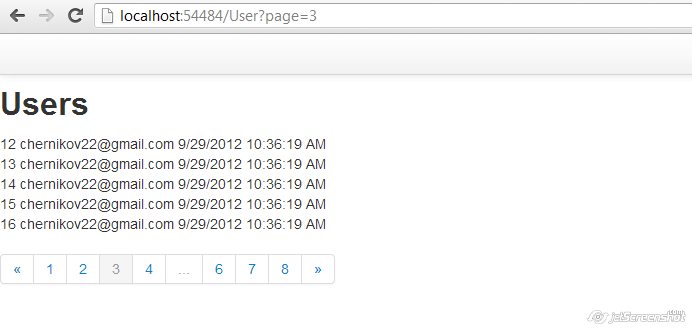
</div>

Обратите внимание на конструкцию

x => Url.Action("Index", new {page = x})

Это делегат, который возвращает ссылку на страницу. Url.Action() – формирует ссылку на страницу /User/Index с параметром page = x.

Вот что у нас получилось (уменьшил кол-во вывода на странице до 5 чтобы образовалось больше страниц):



### SearchEngine

Следующим шагом к просмотру данных будет создание поиска. Поиск будет простой по совпадению подстроки в одном из полей данных. Входной параметр – searchString.

public ActionResult Index(int page = 1, string searchString = null)

{

if (!string.IsNullOrWhiteSpace(searchString))

{

//тут Поиск

return View(data);

}

else

{

var data = new PageableData<User>(Repository.Users, page, 5);

return View(data);

}

}

Создадим класс SearchEngine, который принимает значения IQueryable<User>, и строку поиска, а возвращает данные по поиску (/Global/SearchEngine.cs):

Первым делом создадим классы по очистке строки запроса, никаких тегов и убираем разделители типа [,], {,}, (,):

/// <summary>

/// The regex strip html.

/// </summary>

private static readonly Regex RegexStripHtml = new Regex("<[^>]\*>", RegexOptions.Compiled);

private static string StripHtml(string html)

{

return string.IsNullOrWhiteSpace(html) ? string.Empty :

RegexStripHtml.Replace(html, string.Empty).Trim();

}

private static string CleanContent(string content, bool removeHtml)

{

if (removeHtml)

{

content = StripHtml(content);

}

content =

content.Replace("\\", string.Empty).

Replace("|", string.Empty).

Replace("(", string.Empty).

Replace(")", string.Empty).

Replace("[", string.Empty).

Replace("]", string.Empty).

Replace("\*", string.Empty).

Replace("?", string.Empty).

Replace("}", string.Empty).

Replace("{", string.Empty).

Replace("^", string.Empty).

Replace("+", string.Empty);

var words = content.Split(new[] { ' ', '\n', '\r' }, StringSplitOptions.RemoveEmptyEntries);

var sb = new StringBuilder();

foreach (var word in

words.Select(t => t.ToLowerInvariant().Trim()).Where(word => word.Length > 1))

{

sb.AppendFormat("{0} ", word);

}

return sb.ToString();

}

Создаем поиск:

public static IEnumerable<User> Search(string searchString, IQueryable<User> source)

{

var term = CleanContent(searchString.ToLowerInvariant().Trim(), false);

var terms = term.Split(new[] { ' ' }, StringSplitOptions.RemoveEmptyEntries);

var regex = string.Format(CultureInfo.InvariantCulture, "({0})", string.Join("|", terms));

foreach (var entry in source)

{

var rank = 0;

if (!string.IsNullOrWhiteSpace(entry.Email))

{

rank += Regex.Matches(entry.Email.ToLowerInvariant(), regex).Count;

}

if (rank > 0)

{

yield return entry;

}

}

}

В первой строке очищаем строку запроса. Создаем regex для поиска. В данном случае мы ищем только в поле Email у пользователей.

Как это работает:

* При вводе слова в поиске например «cher [2]» вначале убираем разделители, получаем «cher 2».
* Создаем regex = (cher|2).
* Просматриваем весь список переданный через IQueryable<User>
* Если есть совпадение, то выносим его в IEnumerable - yield return entry

Изменяем Action (/Areas/Default/Controller/UserController.cs):

public ActionResult Index(int page = 1, string searchString = null)

{

ViewBag.Search = searchString;

if (!string.IsNullOrWhiteSpace(searchString))

{

var list = SearchEngine.Search(searchString, Repository.Users).AsQueryable();

var data = new PageableData<User>(list, page, 5);

return View(data);

}

else

{

var data = new PageableData<User>(Repository.Users, page, 5);

return View(data);

}

}

Добавляем форму поиска во View:

@{

ViewBag.Title = "Users";

Layout = "~/Areas/Default/Views/Shared/\_Layout.cshtml";

var searchString = (string)ViewBag.Search;

}

<h2>Users</h2>

@using (Html.BeginForm("Index", "User", FormMethod.Post, new { @class = "form-search" }))

{

@Html.TextBox("searchString", searchString ?? "", new { @class = "input-medium search-query" })

<button type="submit" class="btn">Поиск</button>

}

Обратите внимание на ViewBag, это dynamic контейнер, им можно пользоваться для передачи второстепенных данных.

Добавим в пагинатор строку поиска:

@Html.PageLinks(Model.PageNo, Model.CountPage, x => Url.Action("Index", new {page = x, searchString}))

### Extension

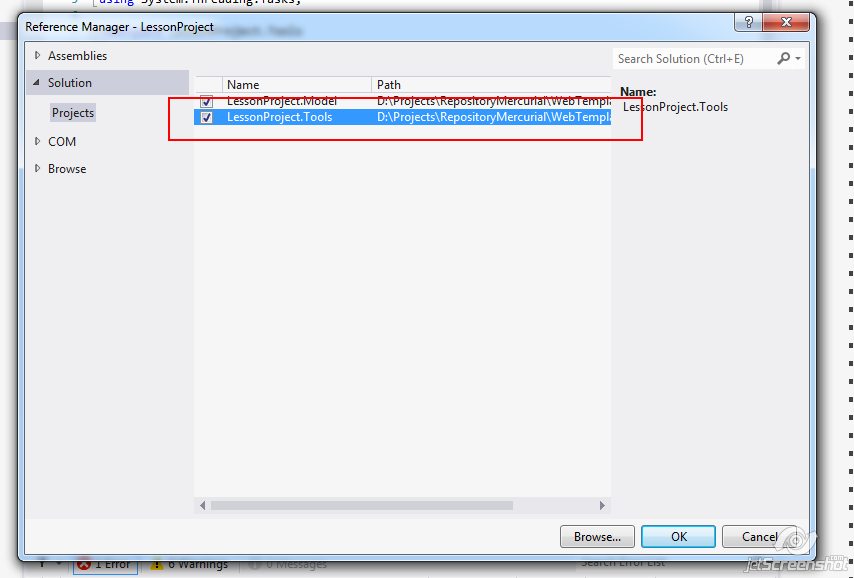
Расширения для строк или целых чисел очень удобно использовать в проекте. Мы сделаем несколько реализаций, но вынесем это в отдельный проект. Добавим проект LessonProject.Tools class WebExtensions:

public static class WebExtensions

{

}

Пропишем в reference LessonProject.



Многострочные данные хранятся с разделителем \r\n и при выводе в тексте эти разделители не учитываются. Необходимо создать функцию NlToBr(). Создаем:

public static MvcHtmlString NlToBr(this string source)

{

if (string.IsNullOrWhiteSpace(source))

{

return new MvcHtmlString(string.Empty);

}

return new MvcHtmlString(source.Replace(Environment.NewLine, "<br />"));

}

Добавляем объявление в Web.config:

<add namespace="LessonProject.Tools" />

Можно использовать:

@Model.Description.NlToBr()

Так же создадим расширение Teaser, которое урезает строку до максимального количества символов, и ставит «…» после, если строка оказалась длиннее.

public static string Teaser(this string content, int length, string more = "...")

{

if (string.IsNullOrWhiteSpace(content))

{

return string.Empty;

}

if (content.Length < length)

{

return content;

}

return content.Substring(0, length) + more;

}

Использовать можно:

@Model.Description.Teaser(120, “>>>”)

Следующее расширение относится к целым числам и подставляет одно из слов в определении для 1, 2 или 5. Например 1 год, 2 года, 5 лет:

public static string CountWord(this int count, string first, string second, string five)

{

if (count % 10 == 1 && (int)(count / 10) != 1)

{

return first;

}

if (count % 10 > 1 && count % 10 < 5 && ((int)(count / 10) % 10) != 1)

{

return second;

}

return five;

}

Использовать можно так:

@year @year.CountWord(“год”, “года”, “лет”)

### Динамические формы

Я уже писал [статью](http://habrahabr.ru/post/88766/) по этому поводу. Но то был asp.net mvc первый, и сейчас все еще проще можно сделать.

Значит, проблема заключается в следующем. Мы на post-action в контроллере принимаем объект, у которого заранее не известно кол-во полей. Например, это будет следующая структура:

public class Ownership

{

public string Name { get; set; }

public int Price { get; set; }

}

public class Customer

{

public int ID { get; set; }

public string Name { get; set; }

public Dictionary<string,Ownership> Ownerships { get; set; }

}

Можно использовать и List вместо Dictionary, но в будущем будут проблемы с валидацией для определенного элемента.

Создадим контроллер:

public class CustomerController : DefaultController

{

[HttpGet]

public ActionResult Edit(int id)

{

return View(new Customer()

{

Ownerships = new Dictionary<string, Ownership>()

});

}

[HttpPost]

public ActionResult Edit(Customer customer)

{

if (ModelState.IsValid)

{

}

return View(customer);

}

}

В БД мы не будем ничего записывать, так что тут – это просто заглушка. Добавляем View:

@model LessonProject.Models.Info.Customer

@{

ViewBag.Title = "Edit";

Layout = "~/Areas/Default/Views/Shared/\_Layout.cshtml";

}

@section scripts {

@Scripts.Render("/Scripts/default/customer-edit.js")

}

<h2>Edit</h2>

@using (Html.BeginForm("Edit", "Customer", FormMethod.Post, new { @class = "form-horizontal" }))

{

<fieldset>

@Html.Hidden("ID", Model.ID)

<div class="control-group">

<label class="control-label" for="Email">

Name</label>

<div class="controls">

@Html.TextBox("Name", Model.Name, new { @class = "input-xlarge" })

@Html.ValidationMessage("Name")

</div>

</div>

<div id="OwnershipListWrapper">

<div class="btn" id="AddOwnership">Добавить</div>

@foreach (var keyValuePair in Model.Ownerships)

{

**@Html.Partial("OwnershipItem", keyValuePair)**

}

</div>

<div class="form-actions">

<button type="submit" class="btn btn-primary">

Ок</button>

</div>

</fieldset>

}

keyValuePair переносим в PartialView (/Areas/Default/Views/Customer/OwnershipItem.cshtml):

**@model KeyValuePair<string, LessonProject.Models.Info.Ownership>**

<div class="OwnershipWrapper">

<div class="btn remove-line">Удалить</div>

<div class="control-group">

<label class="control-label">

Имя

</label>

<div class="controls">

@Html.TextBox("Ownerships[" + Model.Key + "].Name", Model.Value.Name, new { @class = "input-xlarge" })

@Html.ValidationMessage("Ownerships[" + Model.Key + "].Name")

</div>

</div>

<div class="control-group">

<label class="control-label">

Цена

</label>

<div class="controls">

@Html.TextBox("Ownerships[" + Model.Key + "].Price", Model.Value.Price, new { @class = "input-xlarge" })

@Html.ValidationMessage("Ownerships[" + Model.Key + "].Price")

</div>

</div>

</div>

Js-обработчик как и ранее состоит из обработки кнопки добавления и удаления (/Scripts/default/customer-edit.js):

function CustomerEdit() {

\_this = this;

this.ajaxAddOwnership = "/Customer/AddOwnership";

this.init = function () {

$("#AddOwnership").click(function () {

$.ajax({

type: "GET",

url: \_this.ajaxAddOwnership,

success: function (data) {

$("#OwnershipListWrapper").append(data);

}

})

});

$(document).on("click", ".remove-line", function () {

$(this).closest(".OwnershipWrapper").remove();

});

}

}

var customerEdit = null;

$().ready(function () {

customerEdit = new CustomerEdit();

customerEdit.init();

});

При нажатии на кнопку добавить мы получаем по ajax-запросу часть уже сформированный и добавим к списку. При удалении по клику просто удалим ряд значений найдя ближайший OwnershipWrapper. Обратите внимание на создание глобального обработчика для remove-line. Это необходимо для того, чтобы динамически созданные кнопки тоже обрабатывали этот клик.

Добавим обработчик в CustomerController используя уже созданный нами View OwnershipItem.cshtml:

public ActionResult AddOwnership()

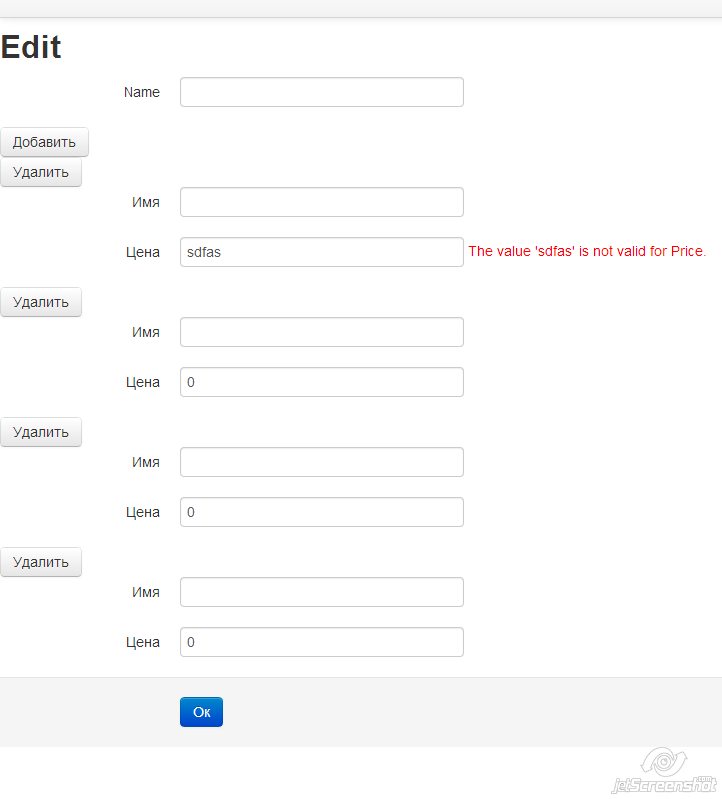
{

return View("OwnershipItem", new KeyValuePair<string, Ownership>(

Guid.NewGuid().ToString("N"),

new Ownership()));

}



### Правила перенаправления

Есть две ситуации, который подобны, но по-разному обрабатываются. Речь идет об перенаправлении страницы на страницу входа и об ошибке 404 (страница не найдена).

Итак, когда мы пытаемся получить по прямой ссылке доступ к админке, но в данный момент мы не залогинены, или залогинены, но не имеет прав на это – то нас переправит на страницу указанную в Web.config в секции authentication:

<authentication mode="Forms">

<forms loginUrl="~/Login" timeout="2880" />

</authentication>

Если мы пытаемся открыть несуществующую страницу, т.е. по данным роутинга не найден контроллер и action-метод, то нас перенаправляют на страницу, которая указывается в секции customErrors в Web.config:

<customErrors mode="On" redirectMode="ResponseRedirect" defaultRedirect="~/Error">

<error statusCode="403" redirect="~/Error" />

<error statusCode="404" redirect="~/NotFoundPage" />

</customErrors>

То же происходит и по обработки ошибки. На боевом сервере не очень хорошо выдавать «желтый экран смерти», где указана причина возникновения ошибки.

Но иногда, перенаправление на страницу 404 или на страницу входа мы должны сделать самостоятельно.

Это может возникнуть, когда проверка роли выполнена, но пользователь берется за управление чужого ресурса, при обращении к которому, естественно, в доступе ему должен быть отказано.

Или второй, и более частый случай, когда по битой ссылке не находится ресурс, в данном случае, надо переправить на NotFoundPage.

Добавим свойства в BaseController (/Controllers/BaseController.cs):

protected static string ErrorPage = "~/Error";

protected static string NotFoundPage = "~/NotFoundPage";

protected static string LoginPage = "~/Login";

public RedirectResult RedirectToNotFoundPage

{

get

{

return Redirect(NotFoundPage);

}

}

public RedirectResult RedirectToLoginPage

{

get

{

return Redirect(LoginPage);

}

}

protected override void OnException(ExceptionContext filterContext)

{

base.OnException(filterContext);

filterContext.Result = Redirect(ErrorPage);

}

Теперь можно использовать таким образом в UserController (/Areas/Default/Controllers/UserController.cs):

[Authorize]

public ActionResult Edit(int id)

{

var user = Repository.Users.FirstOrDefault(p => p.ID == id);

if (user != null)

{

if (CurrentUser.InRoles("admin") || CurrentUser.ID == id)

{

//Разрешено редактирование

return View(user);

}

return RedirectToLoginPage;

}

return RedirectToNotFoundPage;

}

Обратите внимание, что изначально необходимо быть авторизованным в системе, но после редактирование разрешено только самому пользователю, котого эти данные или пользователю с правами админа.

Добавим обработку ошибки и 404-страницу. Создадим контроллер, но не будем его наследовать от BaseController, чтобы ненароком не зацепить еще какую-то ошибку. Добавим пути маршрутов в обработку (/Areas/Default/DefaultAreaRegistration.cs):

context.MapRoute(

null,

url: "Error",

defaults: new { controller = "Error", action = "Index", id = UrlParameter.Optional },

namespaces: new[] { "LessonProject.Areas.Default.Controllers" }

);

context.MapRoute(

null,

url: "NotFoundPage",

defaults: new { controller = "Error", action = "NotFoundPage", id = UrlParameter.Optional },

namespaces: new[] { "LessonProject.Areas.Default.Controllers" }

);

Контроллер (/Areas/Default/Controllers/ErrorController.cs):

public class ErrorController : Controller

{

public ActionResult Index()

{

Response.StatusCode = (int)HttpStatusCode.InternalServerError;

return View();

}

public ActionResult NotFoundPage()

{

Response.StatusCode = (int)HttpStatusCode.NotFound;

return View();

}

}

Добавляем простейшую страницу ошибки View (/Areas/Default/Views/Error/Index.cshtml):

@{

Layout = null;

}

<!DOCTYPE html>

<html>

<head>

<meta name="viewport" content="width=device-width" />

<title>Index</title>

</head>

<body>

<div>

Ошибка <br />

<a href="/">Вернуться на главную</a>

</div>

</body>

</html>

Аналогичная страница 404 (/Areas/Default/Views/Error/NotFoundPage.cshtml).

### Другие ActionResult

ActionResult не обязательно может возвращать View. Рассмотрим другие:

* Content(result) – возвращает простой текст в http-ответе
* JsonResult(object) – возвращает объект в формате json (далее рассмотрим)
* FileContentResult() – возвращает ресурс для скачивания
* FilePathResult() – возвращает файл, расположенный по указанному пути
* FileStreamResult() – возвращает в файле записанный поток
* RedirectResult() – переправляет на указанную страницу
* RedirectToRouteResult() – переправляет на страницу по новому указанному пути

### RssActionResult

RSS – это XML формат, предназначенный для описание лент новостей, статей, блогов. У нас пока нет постов, так что мы просто создадим RssActionResult. В System.ServiceModel.Syndication есть набор инструментов для работы с RSS. Нам необходимы SyndicationFeed и SyndicationItem. Подключим в reference System.ServiceModel, создадим новый контроллер (/Areas/Default/Controllers/FeedController.cs):

public class FeedController : DefaultController

{

public ActionResult Index()

{

var host = Request.Url;

var feed =

new SyndicationFeed("Site RSS",

"",

new Uri(host.AbsoluteUri + "/Feed"));

var items = new List<SyndicationItem>();

var item = new SyndicationItem(

"Title",

"content",

new Uri("http://" + host + "/some-link-url"),

"Title",

DateTime.Now

);

items.Add(item);

feed.Items = items;

return View();

}

}

Абсолютно тестовые данные, одно значение на нерабочую ссылку. Создадим RssActionResult (/Global/RssActionResult.cs):

public class RssActionResult : ActionResult

{

public SyndicationFeed Feed { get; set; }

public override void ExecuteResult(ControllerContext context)

{

context.HttpContext.Response.ContentType = "application/rss+xml";

var rssFormatter = new Rss20FeedFormatter(Feed);

using (var writer = XmlWriter.Create(context.HttpContext.Response.Output))

{

rssFormatter.WriteTo(writer);

}

}

}

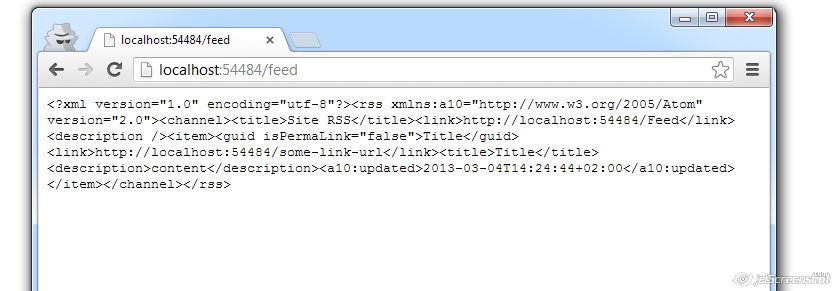
Основным параметром является Feed, куда передается уже сформированный Feed. Сам ActionResult вызывает ExecuteResult(). Мы используем Rss20FeedFormatted для записи xml в Response.Output. Response.Output – это stream, куда записывая данные обрабатываются браузером как ответ. Для того чтобы браузер понимал, какой именно тип данных мы передаем мы присваиваем в Response.ContentType = “application/rss+xml”.

Возвращаем наш RssActionResult (/Areas/Default/Controllers/FeedController.cs):

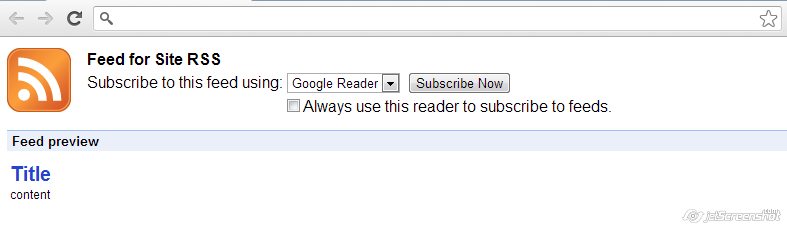
…

return new RssActionResult {Feed = feed };

Запускаем:



Если установить [RSS Subscription Extension (by Google)](https://chrome.google.com/webstore/detail/rss-subscription-extensio/nlbjncdgjeocebhnmkbbbdekmmmcbfjd) то оно отобразит наш RSS для подписки:



Можете попробовать реализовать Sitemap вывод. Вот протокол <http://www.sitemaps.org/protocol.html> описывающий формат.

## Урок 9. Configuration и загрузка файлов

Цель: Научиться использовать файл конфигурации Web.config. Application section, создание своих ConfigSection и IConfig. Научится загружать файлы, использование file-uploader для загрузки файла, последующая обработка файла.

В этом уроке мы рассмотрим работу с конфигурационным файлом Web.config. Это xml файл и нем хранятся настройки программы. Рассмотрим подробнее, из чего состоит этот файл:

* **configSection**. Это секция отвечает за то, какие классы будут разбирать далее объявленные секции. Состоит из атрибута name - это тег далее объявленной секция и type – к какому классу относится.
* **connectionStrings**. Это секция отвечает за работу с указанием строк инициализаций соединений с базами данных.
* **appSettings**. Секция параметров типа key/value.
* **system.web, system.webServer.** Секции параметров для работы веб-приложения.
* **runtime**. Секция по настройке в режиме выполнения. Определение зависимостей между dll.
* **Остальные секции.** Другие секции с параметрами, которые были объявлены в configSection.

### IConfig (и реализация).

Как и Repository конфигуратор будем создавать как сервис. Создаем IConfig и Config-реализацию в папке Global (/Global/Config/IConfig.cs):

public interface IConfig

{

string Lang { get; }

}

И

public class Config : IConfig

{

public string Lang

{

get

{

return "ru";

}

}

}

Добавляем строку в RegisterServices (/App\_Start/NinjectWebCommon.cs):

kernel.Bind<IConfig>().To<Config>().InSingletonScope();

И выводим в BaseController:

[Inject]

public IConfig Config { get; set; }

Ок, теперь сделаем в инициализации контроллера переопеределение CultureInfo в потоке (/Controllers/BaseController.cs):

protected override void Initialize(System.Web.Routing.RequestContext requestContext)

{

try

{

var cultureInfo = new CultureInfo(Config.Lang);

Thread.CurrentThread.CurrentCulture = cultureInfo;

Thread.CurrentThread.CurrentUICulture = cultureInfo;

}

catch (Exception ex)

{

logger.Error("Culture not found", ex);

}

base.Initialize(requestContext);

}

И добавим вывод даты в Index.cshtml (/Areas/Default/Views/Home/Index.cshtml):

@DateTime.Now.ToString("D")

Получаем вывод:



Ок, теперь по-настоящему свяжем это с Web.Config. Добавим в Web.config в appSettings строку:

<add key="Culture" value="ru" />

В Config.cs (/Global/Config/Config.cs):

public string Lang

{

get

{

return ConfigurationManager.AppSettings["Culture"] as string;

}

}

Запускаем – результат тот же, теперь изменим значение в Web.config на fr:

<add key="Culture" value="fr" />

Получаем дату:

mardi 5 mars 2013

Отлично! Можете попробовать еще с несколькими языками. Список сокращений находится тут <http://msdn.microsoft.com/en-us/goglobal/bb896001.aspx>

### Создание своих типов ConfigSection

Следующей частью мы рассмотрим создание своих собственных ConfigSection. В этойглаве мы будем реализовывать загрузку файлов и создание превью, и нам понадобятся следующие данные. Первые, это зависимость mime-type от расширения, и иконка файлов (для скачивания, например):

* расширение
* mime-type
* большая иконка
* маленькая иконка

И второй тип, это данные для создания превью:

* наименование превью (например UserAvatarSize)
* ширина
* высота

Оба типа делаются одинаково, так что я распишу только создание одного из них, и это будет IconSize, для создания превью. Первое, что надо сделать - это создать класс наследуемый ConfigurationElement (/Global/Config/IconSize.cs):

public class IconSize : ConfigurationElement

{

[ConfigurationProperty("name", IsRequired = true, IsKey = true)]

public string Name

{

get

{

return this["name"] as string;

}

}

[ConfigurationProperty("width", IsRequired = false, DefaultValue = "48")]

public int Width

{

get

{

return (int)this["width"];

}

}

[ConfigurationProperty("height", IsRequired = false, DefaultValue = "48")]

public int Height

{

get

{

return (int)this["height"];

}

}

}

Рассмотрим подробнее:

* **ConfigurationProperty** состоит из имени, это имя атрибута в строке
* **IsRequired** – обязательный этот параметр или нет
* **IsKey** – является ли ключем (как превичный ключ в БД)
* **DefaultValue** – значение по умолчанию

Следующий шаг – это создание класса коллекции (так как у нас будет множество элементов) и секции (/Global/Config/IconSize.cs):

public class IconSizesConfigSection : ConfigurationSection

{

[ConfigurationProperty("iconSizes")]

public IconSizesCollection IconSizes

{

get

{

return this["iconSizes"] as IconSizesCollection;

}

}

}

public class IconSizesCollection : ConfigurationElementCollection

{

protected override ConfigurationElement CreateNewElement()

{

return new IconSize();

}

protected override object GetElementKey(ConfigurationElement element)

{

return ((IconSize)element).Name;

}

}

В Web.config добавляем:

<iconConfig>

<**iconSizes**>

<add **name**="Avatar173Size" **width**="173" **height**="176" />

…

</iconSizes>

</iconConfig>

Теперь необходимо объявить класс разбора этой секции в configSection:

<section name="iconConfig" type="LessonProject.Global.Config.IconSizesConfigSection, **LessonProject**" />

Обратите внимание, что в описание type необходимо указать имя dll, в которой он содержится. Это важно, но будет рассмотрено в unit-тестах.

### MailSettings

Создадим одиночный конфиг для настроек по работе с smtp-почтой. Нам в общем-то понадобятся:

* SmtpServer. Имя сервера.
* SmtpPort. Порт, обычно 25й.
* SmtpUserName. Логин для входа.
* SmtpPassword. Пароль, он нужен.
* SmtpReply. Что будет писать в строке Reply-to.
* SmtpUser. Что будет писать в строке From.
* EnableSsl. Да/нет, работает ли по Ssl.

Файл (/Global/Config/MailSetting.cs):

public class MailSetting : ConfigurationSection

{

[ConfigurationProperty("SmtpServer", IsRequired = true)]

public string SmtpServer

{

get

{

return this["SmtpServer"] as string;

}

set

{

this["SmtpServer"] = value;

}

}

[ConfigurationProperty("SmtpPort", IsRequired = false, DefaultValue="25")]

public int SmtpPort

{

get

{

return (int)this["SmtpPort"];

}

set

{

this["SmtpPort"] = value;

}

}

[ConfigurationProperty("SmtpUserName", IsRequired = true)]

public string SmtpUserName

{

get

{

return this["SmtpUserName"] as string;

}

set

{

this["SmtpUserName"] = value;

}

}

[ConfigurationProperty("SmtpPassword", IsRequired = true)]

public string SmtpPassword

{

get

{

return this["SmtpPassword"] as string;

}

set

{

this["SmtpPassword"] = value;

}

}

[ConfigurationProperty("SmtpReply", IsRequired = true)]

public string SmtpReply

{

get

{

return this["SmtpReply"] as string;

}

set

{

this["SmtpReply"] = value;

}

}

[ConfigurationProperty("SmtpUser", IsRequired = true)]

public string SmtpUser

{

get

{

return this["SmtpUser"] as string;

}

set

{

this["SmtpUser"] = value;

}

}

[ConfigurationProperty("EnableSsl", IsRequired = false, DefaultValue="false")]

public bool EnableSsl

{

get

{

return (bool)this["EnableSsl"];

}

set

{

this["EnableSsl"] = value;

}

}

}

Добавим в Web.config:

<section name="mailConfig" type="LessonProject.Global.Config.MailSetting, LessonProject" />

И

<mailConfig

SmtpServer="smtp.gmail.com"

SmtpPort="587"

SmtpUserName="lxndrpetrov"

SmtpPassword="\*\*\*\*\*\*\*\*\*\*"

SmtpReply="lxndrpetrov@gmail.com"

SmtpUser="test"

EnableSsl="true" />

Добавим это теперь в IConfig.cs и Сonfig.cs (/Global/Config/IConfig.cs):

public interface IConfig

{

string Lang { get; }

IQueryable<IconSize> IconSizes { get; }

IQueryable<MimeType> MimeTypes { get; }

MailSetting MailSetting { get; }

}

И

public IQueryable<IconSize> IconSizes

{

get

{

IconSizesConfigSection configInfo = (IconSizesConfigSection)ConfigurationManager.GetSection("iconConfig");

return configInfo.IconSizes.OfType<IconSize>().AsQueryable<IconSize>();

}

}

public IQueryable<MimeType> MimeTypes

{

get

{

MimeTypesConfigSection configInfo = (MimeTypesConfigSection)ConfigurationManager.GetSection("mimeConfig");

return configInfo.MimeTypes.OfType<MimeType>().AsQueryable<MimeType>();

}

}

public MailSetting MailSetting

{

get

{

return (MailSetting)ConfigurationManager.GetSection("mailConfig");

}

}

Мы еще добавим MailTemplates, которые нам понадобятся для рассылки при регистрации, или при напоминании пароля, но там полностью аналогично с IconSize.

### Простая загрузка файлов

Сейчас рассмотрим простой стандартный пример загрузки файла на сервер, и больше никогда не будем пользоваться таким способом. Класс SimpleFileView для взаимодействия (/Models/Info/SimpleFileView.cs):

public class SimpleFileView

{

public **HttpPostedFileBase** UploadedFile { get; set; }

}

Обратите внимание на наименование класса для приема файлов. Итак, создадим контроллер SimpleFileController (/Areas/Default/Controllers/SimpleFileController.cs):

public class SimpleFileController : DefaultController

{

[HttpGet]

public ActionResult Index()

{

return View(new SimpleFileView());

}

[HttpPost]

public ActionResult Index(SimpleFileView simpleFileView)

{

return View(simpleFileView);

}

}

И добавим View:

@model LessonProject.Models.Info.SimpleFileView

@{

ViewBag.Title = "Index";

Layout = "~/Areas/Default/Views/Shared/\_Layout.cshtml";

}

<h2>Index</h2>

@using (Html.BeginForm("Index", "SimpleFile", FormMethod.Post, new **{enctype = "multipart/form-data",** @class = "form-horizontal" }))

{

<fieldset>

<div class="control-group">

<label class="control-label" for="Email">

Загрузите файл:</label>

<div class="controls">

@Html.TextBox("UploadedFile", Model.UploadedFile, new { **type = "file",** @class = "input-xlarge" })

@Html.ValidationMessage("UploadedFile")

</div>

</div>

<div class="form-actions">

<button type="submit" class="btn btn-primary">

Upload</button>

</div>

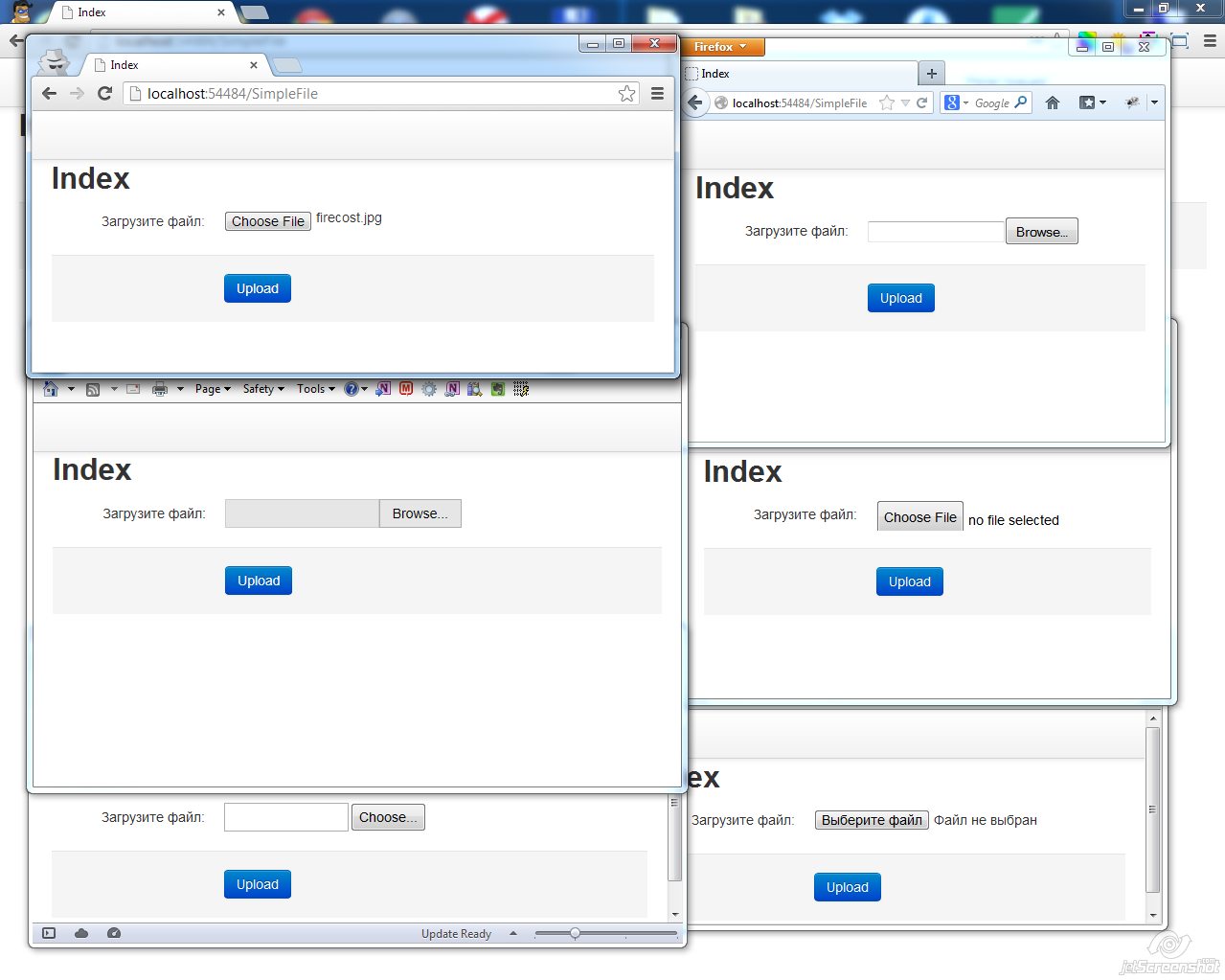
</fieldset>

}

Обратите внимание, на enctype в атрибутах формы и на type в атрибутах TextBox (на самом деле тип еще бывает password, checkbox, radio, но для этого есть соответствующие методы в @Html-классе). Enctype необходимо установить в “multipart/form-data”, чтобы была возможность загрузить большой объем информации.

Загружаем и проверяем. Наш файл благополучно оказывается загруженным, необходимо только сохранить InputStream в некий файл. Но оставим пока так, а рассмотрим недостатки.

Первый недостаток – это то, что во всех браузерах выбор файла выглядит по-разному:



Конечно, дизайнер представляет, что загрузка файлов выполняется как в Safari, а заказчик проверяет в Chrome и IE и начинает спрашивать у разработчиков: «Что за самодеятельность?»

Второй недостаток – это то, что если форма не прошла валидацию, то эти поля необходимо выбрать заново. Т.е. вот такая форма например:

* Ваше имя
* Ваша фамилия
* Электронная почта
* Дата рождения
* Фотография
* Фотография первого разворота паспорта
* Фотография второго разворота паспорта
* Фотография паспрорта с пропиской
* Пароль
* Пароль еще раз
* Капча

Ну и вы там путаете пароль, или капчу не так ввели, или фотография второго разворота паспорта слишком большая или вы забыли перегнать из raw-формата в jpeg.

Ну и фотографию, фотографию первого/второго разворота и прописку и капчу надо вводить заново. Естественно, это совсем не пользователе-дружественно, раздражает заказчика (к тому же дизайнер нарисовал красиво, а выглядит убого).

### Ajax загрузка файла (ов)

Так что, дабы не нарываться на неприятности, определим как должно вести себя загрузка файла:

* Пользователь кликает на «загрузить»
* Открывается форма выбора файла
* Пользователь выбирает файл
* Файл загружается, или выдается ошибка, что что-то не так
* Если даже форма и не проходит валидацию, то файл остается загруженным и его по-второму разу не нужно загружать.

Это называется ajax-загрузка и для это используем fineuploader (<http://fineuploader.com/>). Библиотека платная, но мы скачаем и соберем исходники (у нас же есть bundle!). Скачиваем исходники по ссылке: <https://github.com/valums/file-uploader>. Перемещаем js-файлы в папку /Scripts/fine-uploader. Css-файлы перемещаем в /Content и изображение в /Content/images. Перепишем правильно url в fineuploader.css для изображений:

.qq-upload-spinner {

display: inline-block;

background: url("images/loading.gif");

width: 15px;

height: 15px;

vertical-align: text-bottom;

}

.qq-drop-processing {

display: none;

}

.qq-drop-processing-spinner {

display: inline-block;

background: url("images/processing.gif");

width: 24px;

height: 24px;

vertical-align: text-bottom;

}

Файлы инициализируем в BundleConfig.cs (/App\_Start/BundleConfig.cs):

bundles.Add(new ScriptBundle("~/bundles/fineuploader")

.Include("~/Scripts/fine-uploader/header.js")

.Include("~/Scripts/fine-uploader/util.js")

.Include("~/Scripts/fine-uploader/button.js")

.Include("~/Scripts/fine-uploader/ajax.requester.js")

.Include("~/Scripts/fine-uploader/deletefile.ajax.requester.js")

.Include("~/Scripts/fine-uploader/handler.base.js")

.Include("~/Scripts/fine-uploader/window.receive.message.js")

.Include("~/Scripts/fine-uploader/handler.form.js")

.Include("~/Scripts/fine-uploader/handler.xhr.js")

.Include("~/Scripts/fine-uploader/uploader.basic.js")

.Include("~/Scripts/fine-uploader/dnd.js")

.Include("~/Scripts/fine-uploader/uploader.js")

.Include("~/Scripts/fine-uploader/jquery-plugin.js")

);

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

.Include("~/Content/fineuploader.css"));

Создаем контроллер FileController.cs (/Areas/Default/Controllers/FileController.cs):

public class FileController : DefaultController

{

[HttpGet]

public ActionResult Index()

{

return View();

}

public ActionResult Upload(HttpPostedFileWrapper qqfile)

{

return Json(new { result = "ok", success = true});

}

}

Метод-action Upload принимает строковое значение qqfile, я чуть далее рассмотрю почему так. А счас создадим View для Index. Расскажу план:

* Создаем кнопку, при нажатии на которую мы загружаем файл.
* Файл загружается и создается превью
* Файл и превью сохраняются в файловую систему
* Метод возвращает ссылку, куда были загружены файл и превью, через Json-ответ
* Если файлы не удалось загрузить, то выдается соответствующая ошибка
* Обрабатываем json-результат и уведомляем, что файл и превью загружено
* Всё. Верификация формы и запись в БД тут нет необходимости.

Ок, так что View для Index:

@{

ViewBag.Title = "Index";

Layout = "~/Areas/Default/Views/Shared/\_Layout.cshtml";

}

@section styles {

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

}

@section scripts {

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

@Scripts.Render("~/Scripts/default/file-index.js")

}

<h2>Index</h2>

<fieldset>

<div class="control-group">

<label class="control-label" for="Text">

Image

</label>

<div class="controls">

<div id="UploadImage">

Upload

</div>

</div>

</div>

<div>

<img src="" alt="" id="ImagePreview" />

</div>

</fieldset>

Наша кнопка с id=UploadImage. Добавляем file-index.js файл для обработки (/Scripts/default/file-index.js):

function FileIndex() {

\_this = this;

this.ajaxFileUpload = "/File/Upload";

this.init = function () {

$('#UploadImage').fineUploader({

request: {

endpoint: \_this.ajaxFileUpload

},

}).on('error', function (event, id, name, reason) {

//do something

})

.on('complete', function (event, id, name, responseJSON) {

**alert(responseJSON);**

});

}

}

var fileIndex = null;

$().ready(function () {

fileIndex = new FileIndex();

fileIndex.init();

});

Ок. Теперь обработаем загрузку:

public ActionResult Upload(HttpPostedFileWrapper qqfile)

{

var extension = Path.GetExtension(qqfile.FileName);

if (!string.IsNullOrWhiteSpace(extension))

{

var mimeType = Config.MimeTypes.FirstOrDefault(p => string.Compare(p.Extension, extension, 0) == 0);

//если изображение

if (**mimeType.Name.Contains("image")**)

{

//тут сохраняем в файл

var filePath = Path.Combine("/Content/files", qqfile.FileName);

qqfile.SaveAs(**Server.MapPath(filePath)**);

return Json(new

{

success = true,

result = "error",

data = new

{

filePath

}

});

}

}

return Json(new { error = "Нужно загрузить изображение", success = false });

}

В Content добавим папку files - это будет папка пользовательских данных. Разберем код:

* Получаем qqfile (тут ничего не поменять, это параметр обусловлен fineuploader).
* Из него получаем extension.
* По extension находим mimeType. Для .jpg, .gif, .png – мы получаем mime-type типа «image/…». Таким образом, мы проверяем, что этот файл можно загрузить.
* Далее используя имя файла, составляем абсолютный путь к папке /Content/files (которую мы так заранее создали). Это мы делаем с помощью Server.MapPath.
* Далее сохраняем файл с помощью SaveAs.
* Возвращаем имя файл в json data.filePath.

Ок, проверим что всё загружается и приступим к созданию превью.

### Создание превью

Во-первых, мы немного схитрили с mime-type = «image\...», ведь к ним относится и bmp, и tiff файлы, которые не поддерживаются браузерами.

Так что создадим класс PreviewCreator в проекте LessonProject.Tools (PreviewCreator.cs):

public static class PreviewCreator

{

public static bool SupportMimeType(string mimeType)

{

switch (mimeType)

{

case "image/jpg":

case "image/jpeg":

case "image/png":

case "image/gif":

return true;

}

return false;

}

}

И заменим в FileController.cs (/Areas/Default/Controller/FileController.cs):

if (mimeType != null && PreviewCreator.SupportMimeType(mimeType.Name))

В PreviewCreator уже создано много функций по для создания превью, так что я перечислю разные варианты создания изображения, и подробно разберу один из вариантов. Стоит учесть, что все превью создаются в формате jpeg. Итак, какие есть варианты:

* **Цветной и чернобелый вариант.** Контролируется параметром grayscale (по умолчанию = false)
* **Превью.** (*CreateAndSavePreview*) Если исходное изображение меньше, чем размеры превью, то изображение размещается посередине белого холста. Если по отношению к размерам исходный размер имеет вертикальную ориентированность (квадратик из портретного формата) – вырезаем верхнюю часть. Если же отношение горизонтально ориентированно относительно размера, то вырезаем середину.
* **Аватар.** (*CreateAndSaveAvatar*) Если исходное изображение меньше, чем размеры превью, то изображение просто сохраняется. Если по отношению к размерам исходный размер имеет вертикальную ориентированность (квадратик из портретного формата) – то уменьшаем, по высоте. Если же отношение горизонтально ориентированно относительно размера, то вырезаем середину.
* **Изображение.** (*CreateAndSaveImage*) Если изображение меньше, чем максимальные размеры, то сохраняем исходное. Если же изображение не вписывается в границы, то уменьшаем, чтобы оно не превышало максимальный размер, и сохраняем.
* **По размеру.** (*CreateAndSaveFitToSize*) Если изображение меньше, чем размеры, то оно будет растянуто до необходимых размеров. С потерей качества, конечно же.
* **Обрезать.** (*CropAndSaveImage*) Кроме стандартных параметров передаются координаты для обрезки изображения.

Ок, создадим просто превью (*CreateAndSavePreview*) взяв из конфигурации размеры для создания превью **AvatarSize** (/Areas/Default/Controllers/FileController.cs):

var filePreviewPath = Path.Combine("/Content/files/previews", qqfile.FileName);

var previewIconSize = Config.IconSizes.FirstOrDefault(c => c.Name == "AvatarSize");

if (previewIconSize != null)

{

PreviewCreator.CreateAndSavePreview(qqfile.InputStream, new Size(previewIconSize.Width, previewIconSize.Height), Server.MapPath(filePreviewPath));

}

return Json(new

{

success = true,

result = "error",

data = new

{

filePath,

filePreviewPath

}

});

Запускаем. Загружаем. Файлы должны загрузиться, и создаться превью.

Теперь сделаем обработку во file-index.js (/Scripts/default/file-index.js):

$('#UploadImage').fineUploader({

request: {

endpoint: \_this.ajaxFileUpload

},

})

.on('error', function (event, id, name, reason) {

//do something

})

.on('complete', function (event, id, name, responseJSON) {

**$("#ImagePreview").attr("src", responseJSON.data.filePreviewPath);**

});

Ок, теперь наш файл загружается вместе с превью, всё круто. Путь большого файла также можно передавать отдельно, и записывать, например, в hidden поле и сохранять в дальнейшем в БД как строку.

Что плохо в такой конструкции, это две следующие проблемы:

* что файлы могут быть перезаписаны, но это решается тем, что можно брать только расширение, а имя файлу присваивать отдельно, или [добавлять немного соли](http://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D0%BB%D1%8C_(%D0%BA%D1%80%D0%B8%D0%BF%D1%82%D0%BE%D0%B3%D1%80%D0%B0%D1%84%D0%B8%D1%8F))
* что файлы могут быть загружены и не связаны с БД. Это можно решить тем, что для каждой таблице файлы записывать в отдельную папку, а потом делать поиск и удалять не записанные.

### Получение файлов по ссылке

Ну собственно, есть еще один метод загрузки файла. Файл свободно болтается в интернете, а мы указываем путь к нему (например, при авторизации с facebook), а мы уже выцепляем эту картинку и делаем с ней что хотим. Ну или любой другой файл, не важно.

Это делается так:

var webClient = new WebClient();

var bytes = webClient.DownloadData(url);

var ms = new MemoryStream(bytes);

Где url – путь к файлу. Шустренько? Ладно, можно посложнее с использованием HttpWebRequest:

public ActionResult Export(string uri)

{

HttpWebRequest webRequest = (HttpWebRequest)WebRequest.Create(uri);

webRequest.Method = "GET";

webRequest.KeepAlive = false;

webRequest.PreAuthenticate = false;

webRequest.Timeout = 1000;

var response = webRequest.GetResponse();

var stream = response.GetResponseStream();

var previewIconSize = Config.IconSizes.FirstOrDefault(c => c.Name == "AvatarSize");

var filePreviewPath = Path.Combine("/Content/files/previews", **Guid.NewGuid().ToString("N")** + ".jpg");

if (previewIconSize != null)

{

PreviewCreator.CreateAndSavePreview(stream, new Size(previewIconSize.Width, previewIconSize.Height), Server.MapPath(filePreviewPath));

}

return Content("OK");

}

Тут файл задается через генерацию Guid.NewGuid. Проверяем:

http://localhost[/File/Export?uri=https://st.free-lance.ru/users/chernikov/upload/sm\_f\_81850beffd0d0c89.jpg](http://localhost:54484/File/Export?uri=https://st.free-lance.ru/users/chernikov/upload/sm_f_81850beffd0d0c89.jpg)

Файл загрузился и преобразовался. Всё супер!

Рекомендую пройтись с дебаггером по работе PreviewCreator, чтобы понять, как там всё устроено.

## Урок A. Уведомление и рассылка

Цель: Разобраться в отправлении писем и подтверждающих смс. MailNotify, использование конфигурационного файла. Рассылка через создание отдельного потока.

### SmtpClient и MailNotify

В общем, при разработке сайта мы рано или поздно сталкиваемся с взаимодействием с электронной почтой, будь то активировать пользователя, или напомнить/сбросить пароль, или рассылочку создать.

Определимся, что нам надо:

* Класс, который будет рассылать письма
* Конфигурация smtp берется из IConfig
* Ошибки отправки письма протоколируются
* Существует параметр выключающий работу почты, дабы при работе с боевой базой клиентов не разослать какой-то треш.

Создадим класс, назовем его MailSender. Будет статическим (/Tools/Mail/MailSender.cs):

public static class MailSender

{

private static IConfig \_config;

public static IConfig Config

{

get

{

if (\_config == null)

{

\_config = (DependencyResolver.Current).GetService<IConfig>();

}

return \_config;

}

}

private static NLog.Logger logger = NLog.LogManager.GetCurrentClassLogger();

public static void SendMail(string email, string subject, string body, MailAddress mailAddress = null)

{

try

{

if (Config.EnableMail)

{

if (mailAddress == null)

{

mailAddress = new MailAddress(Config.MailSetting.SmtpReply, Config.MailSetting.SmtpUser);

}

MailMessage message = new MailMessage(

mailAddress,

new MailAddress(email))

{

Subject = subject,

BodyEncoding = Encoding.UTF8,

Body = body,

IsBodyHtml = true,

SubjectEncoding = Encoding.UTF8

};

SmtpClient client = new SmtpClient

{

Host = Config.MailSetting.SmtpServer,

Port = Config.MailSetting.SmtpPort,

UseDefaultCredentials = false,

EnableSsl = Config.MailSetting.EnableSsl,

Credentials =

new NetworkCredential(Config.MailSetting.SmtpUserName,

Config.MailSetting.SmtpPassword),

DeliveryMethod = SmtpDeliveryMethod.Network

};

client.Send(message);

}

else

{

logger.Debug("Email : {0} {1} \t Subject: {2} {3} Body: {4}", email, Environment.NewLine, subject, Environment.NewLine, body);

}

}

catch (Exception ex)

{

logger.Error("Mail send exception", ex.Message);

}

}

}

Рассмотрим подробнее:

* По необходимости инициализируется статически IConfig из DependencyResolver
* Если EnableMain – то начинаем работу с почтой, иначе просто письмо запишем в лог файлик
* Если MailAddress не указан, то инициализируется по данным из конфига
* SmtpClient инициализируется по данным из конфига
* Тело письма – это html
* Кодировка – UTF8
* Если при отправке произошла ошибка – то запишем Exception.Message в лог (тут можно и больше информации собирать, но пока нет необходимости).

Ок, теперь рассмотрим рассылку писем по шаблону. Создадим класс (тоже статический) NotifyMail (/Tools/Mail/NotifyMail.cs):

public static class NotifyMail

{

private static NLog.Logger logger = NLog.LogManager.GetCurrentClassLogger();

private static IConfig \_config;

public static IConfig Config

{

get

{

if (\_config == null)

{

\_config = (DependencyResolver.Current).GetService<IConfig>();

}

return \_config;

}

}

public static void SendNotify(string templateName, string email,

Func<string, string> subject,

Func<string, string> body)

{

var template = Config.MailTemplates.FirstOrDefault(p => string.Compare(p.Name, templateName, true) == 0);

if (template == null)

{

logger.Error("Can't find template (" + templateName + ")");

}

else

{

MailSender.SendMail(email,

subject.Invoke(template.Subject),

body.Invoke(template.Template));

}

}

}

Тут аналогично мы получаем конфиг. При рассылке мы указываем шаблон для рассылки и дальше используем Func<string,string> для формирования темы и тела письма для добавления данных.

Уведомим пользователя о регистрации, используя шаблон Register из Web.config:

<add name="Register" subject="Регистрация на {0}" template="Здравствуйте! &lt;br/&gt;&lt;br/&gt; Перейдите по ссылке &lt;a href='http://{1}/User/Activate/{0}'&gt;http://{1}/User/Activate/{0}&lt;/a&gt;, чтобы подтвертить свой почтовый ящик.&lt;br/&gt;-----&lt;br/&gt;С уважением, команда &lt;a href='http://{1}'&gt;{1}&lt;/a&gt;" />

Заметим, как необходимо экранировать html-теги, чтобы правильно сделать шаблон. Да и нужно учитывать зависимость между шаблоном для string.Format() и количеством параметров. В UserController.cs при регистрации добавим (/Areas/Default/Controllers/UserController.cs:Register):

Repository.CreateUser(user);

**NotifyMail.SendNotify("Register", user.Email,**

**subject => string.Format(subject, HostName),**

**body => string.Format(body, "", HostName));**

return RedirectToAction("Index");

HostName мы добавили в инициализации BaseController (/Controllers/BaseController.cs):

public static string HostName = string.Empty;

protected override void Initialize(System.Web.Routing.RequestContext requestContext)

{

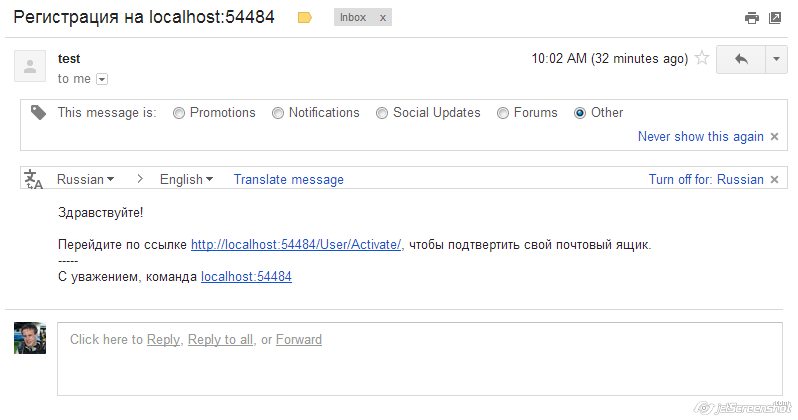
if (requestContext.HttpContext.Request.Url != null)

{

HostName = requestContext.HttpContext.Request.Url.Authority;

} …

Регистрируемся и на нашу почту приходит письмо:



### Более сложный случай

Всё это хорошо, но если нам необходимо сделать что-то типа праздничной рассылки с кучей акционных предложений, то данный формат нам не подойдет. Во-первых, сложно подобный шаблон задавать в Web.config, во-вторых количество параметров не известно. Как и обычные html-шаблоны, шаблон письма было бы чудесно задать во View. Что ж, рассмотрим библиотеку ActionMailer (<http://nuget.org/packages/ActionMailer>):

PM> Install-Package ActionMailer

Successfully installed 'ActionMailer 0.7.4'.

Successfully added 'ActionMailer 0.7.4' to LessonProject.Model.

Отнаследуем MailController от MailerBase:

public class MailController : MailerBase

{

public EmailResult Subscription(string message, string email)

{

To.Add(email);

Subject = "Рассылка";

MessageEncoding = Encoding.UTF8;

return Email("Subscription", message);

}

}

Добавим Subscription.**html**.cshtml View (/Areas/Default/Views/Mail/Subscription.**html**.cshtml):

@model string

@{

Layout = null;

}

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />

</head>

<body>

<div>

<h1>@Model</h1>

</div>

</body>

</html>

Добавляем в Web.config конфигурацию для работы с почтой (Web.config):

<system.net>

<mailSettings>

<smtp deliveryMethod="Network" from="**lxndrpetrov@gmail.com**">

<network host="smtp.gmail.com" port="587" userName="lxndrpetrov" password="\*\*\*\*\*\*" enableSsl="true" />

</smtp>

</mailSettings>

</system.net>

И создаем в UserController.cs тестовый метод (/Areas/Default/Controllers/UserController.cs):

[Authorize]

public ActionResult SubscriptionTest()

{

var mailController = new MailController();

var email = mailController.Subscription("Привет, мир!", CurrentUser.Email);

email.Deliver();

return Content("OK");

}

Запускаем:

<http://localhost/User/SubscriptionTest> - и получаем на почту письмо.

Рассмотрим пример как вместо того чтобы отсылать получить текст письма в строку. Для этого понадобится StreamReader (/Areas/Default/Controllers/UserController.cs):

[Authorize]

public ActionResult SubscriptionShow()

{

var mailController = new MailController();

var email = mailController.Subscription("Привет, мир!", CurrentUser.Email);

using (var reader = new StreamReader(email.Mail.AlternateViews[0].ContentStream))

{

var content = reader.ReadToEnd();

return Content(**content**);

}

return null;

}

В content уже сформированная страница. Запускаем:

<http://localhost/User/SubscriptionShow>

### SmsNotify

В этой главе рассмотрим взаимодействие с помощью смс, а не только почты. Но есть ньюанс – доступ к рассылке предоставляется отдельными сервисами, и тут мы рассмотрим только основные принципы, как необходимо писать модуль для работы с SMS-провайдерами на примере работы с [unisender.ru](http://unisender.ru).

Создадим класс настроек по типу MailSetting (/Global/Config/SmsSetting.cs):

public class SmsSetting : ConfigurationSection

{

[ConfigurationProperty("apiKey", IsRequired = true)]

public string APIKey

{

get

{

return this["apiKey"] as string;

}

set

{

this["apiKey"] = value;

}

}

[ConfigurationProperty("sender", IsRequired = true)]

public string Sender

{

get

{

return this["sender"] as string;

}

set

{

this["sender"] = value;

}

}

[ConfigurationProperty("templateUri", IsRequired = true)]

public string TemplateUri

{

get

{

return this["templateUri"] as string;

}

set

{

this["templateUri"] = value;

}

}

}

Зададим в Web.Config (Web.config):

<configSections>

…

<section name="smsConfig" type="LessonProject.Global.Config.SmsSetting, LessonProject" />

</configSections>

…

<smsConfig

apiKey="\*\*\*\*\*\*\*"

sender="Daddy"

templateUri="http://api.unisender.com/ru/api/sendSms"

/>

</configuration>

Создадим класс SmsSender (/Tools/Sms/SmsSender.cs):

public static class SmsSender

{

private static IConfig \_config;

public static IConfig Config

{

get

{

if (\_config == null)

{

\_config = (DependencyResolver.Current).GetService<IConfig>();

}

return \_config;

}

}

private static NLog.Logger logger = NLog.LogManager.GetCurrentClassLogger();

public static string SendSms(string phone, string text)

{

if (!string.IsNullOrWhiteSpace(Config.SmsSetting.APIKey))

{

return GetRequest(phone, Config.SmsSetting.Sender, text);

}

else

{

logger.Debug("Sms \t Phone: {0} Body: {1}", phone, text);

return "Success";

}

}

private static string GetRequest(string phone, string sender, string text)

{

try

{

HttpWebRequest webRequest = (HttpWebRequest)WebRequest.Create(Config.SmsSetting.TemplateUri);

/// important, otherwise the service can't desirialse your request properly

webRequest.ContentType = "application/x-www-form-urlencoded";

webRequest.Method = "POST";

webRequest.KeepAlive = false;

webRequest.PreAuthenticate = false;

string postData = "format=json&api\_key=" + Config.SmsSetting.APIKey + "&phone=" + phone

+ "&sender=" + sender + "&text=" + HttpUtility.UrlEncode(text);

var ascii = new ASCIIEncoding();

byte[] byteArray = ascii.GetBytes(postData);

webRequest.ContentLength = byteArray.Length;

Stream dataStream = webRequest.GetRequestStream();

dataStream.Write(byteArray, 0, byteArray.Length);

dataStream.Close();

WebResponse webResponse = webRequest.GetResponse();

Stream responceStream = webResponse.GetResponseStream();

Encoding enc = System.Text.Encoding.UTF8;

StreamReader loResponseStream = new

StreamReader(webResponse.GetResponseStream(), enc);

string Response = loResponseStream.ReadToEnd();

return Response;

}

catch (Exception ex)

{

logger.ErrorException("Ошибка при отправке SMS", ex);

return "Ошибка при отправке SMS";

}

}

}

Результат приходит типа:

{"result":{"currency":"RUB","price":"0.49","sms\_id":"1316886153.2\_79859667475"}}

Его можно разобрать и проанализировать. В следующем уроке мы рассмотрим, как работать с json.

### Отдельный поток

Если мы рассылаем электронную почту большому кол-ву людей, то обработка может занять много времени. Для этого я пользуюсь следующим принципом:

* Создаем отдельный поток, который проверяет, если ли исходящие письма готовые к отправке
* При создании рассылки создаются письма и записываются в БД
* Поток проверяет состояние БД на наличие писем
* Одно за другим писем извлекается из БД (может удаляться, может только обнулять содержимое письма (чтобы экономить размер БД).
* Письмо отправляется.
* Возвращается к проверке.

Отдельный поток запускается в Application\_Start. Таймер устанавливается на повторение через 1 минуту:

public class MvcApplication : System.Web.HttpApplication

{

**private static NLog.Logger logger = NLog.LogManager.GetCurrentClassLogger();**

**private Thread mailThread { get; set; }**

protected void Application\_Start()

{

var adminArea = new AdminAreaRegistration();

var adminAreaContext = new AreaRegistrationContext(adminArea.AreaName, RouteTable.Routes);

adminArea.RegisterArea(adminAreaContext);

var defaultArea = new DefaultAreaRegistration();

var defaultAreaContext = new AreaRegistrationContext(defaultArea.AreaName, RouteTable.Routes);

defaultArea.RegisterArea(defaultAreaContext);

FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);

RouteConfig.RegisterRoutes(RouteTable.Routes);

BundleConfig.RegisterBundles(BundleTable.Bundles);

**mailThread = new Thread(new ThreadStart(ThreadFunc));**

**mailThread.Start();**

}

**private static void ThreadFunc()**

**{**

**while (true)**

**{**

**try**

**{**

**var mailThread = new Thread(new ThreadStart(MailThread));**

**mailThread.Start();**

**logger.Info("Wait for end mail thread");**

**mailThread.Join();**

**logger.Info("Sleep 60 seconds");**

**}**

**catch (Exception ex)**

**{**

**logger.ErrorException("Thread period error", ex);**

**}**

**Thread.Sleep(60000);**

**}**

**}**

**private static void MailThread()**

**{**

**var repository = DependencyResolver.Current.GetService<IRepository>();**

**while (MailProcessor.SendNextMail(repository)) { }**

**}**

}

Рассмотрим класс MailProcessor (но не будем его создавать):

public class MailProcessor

{

public static bool SendNextMail(IRepository repository)

{

**var mail = repository.PopMailQueue();**

if (mail != null)

{

MailSender.SendMail(mail.Email, mail.Subject, mail.Body);

return true;

}

return false;

}

}

MailProcessor.SendNextMail(repository) – посылает следующее письмо, если писем нет – возвращает false

Поток MainThread дожидается выполнение MailThread и делает перекур на одну минуту. И далее. Если в БД новых писем нет – дальше курим одну минуту.

## Урок 11. Json

Цель: Научиться работать с форматом json. Инструменты по работе с json. Написание сторонних запросов, авторизация через получение данных от facebook и vkontakte. Ajax в работе с json (авторизацию переписать). API сайта.

### Json и Json.net

*Устанавливаем json.net, разбор json в объекты, берем пример с berry.travel.*

<http://james.newtonking.com/pages/json-net.aspx>

### Работа с facebook и vkontakte

*Пишем регистрацию и авторизацию facebook и vkontakte.AppID и ключи сохраняем в отдельную ветку IConfig.*

### Авторизация (json-mode)

*Переписываем авторизацию на манер возвращения данных посредством json.*

### Написание api

*Пишем минимальное api на сайте. Публикация поста на сайте.*

## Урок 12. Многоязычный сайт

Цель: Научиться просто создавать многоязычные сайты. Структура БД. Ресурсы сайта. Определение языка. Переключение между языками. Работа в админке.

### Проблемы многоязычного сайта

*Берем уже созданный прототип блога и смотрим, какие данные надо поменять. Изменяем структуру БД с использованием двух таблиц: основной и починенной языковой.*

### Ресурсы сайта

*Вывод уже известных строк (статические ресурсы), валидация (строки валидации).*

### Маршрутизация с учетом языка

*Меняем маршрутизацию в зависимости от выбранного языка, язык по умолчанию.*

### Переключение между языками

*Создаем Url.Action который правильно переключает между языками.*

### Языки в админке

*Переключение языков в админке (привязан к админу), указание какие необходимо перевести ресурсы.*

### Неверный формат, перевод на русский

*Показать, как использовать ResourceClassKey - DefaultModelBinder.ResourceClassKey = "Messages"; PropertyValueInvalid и PropertyValueRequired.*

<https://github.com/AutoMapper/AutoMapper/wiki/Getting-started>

## Урок 13. Scaffolding

Цель: Научиться использовать Scaffolding для создания прототипа проекта. Определяем и фиксируем структуру репозитория. Простая и языковая версия класса. Тестируем использование Scaffolder-а, используем «направляющие» атрибуты. Параметры для Scaffolder-а. Создание управляющих атрибутов. Полный цикл создания и управления объекта в админке.

### Scaffolding. Начало.

*Тут пишем много текста о скаффолдинге. CodeTemplate папка. Тестируем на обычном проекте.*

### Репозиторий

*Тут описываем структуру репозитория, папки Proxy и SqlRepository. Многоязычность. Изучаем класс Post (PostLang)*

### Модель данных

*Тут описываем модель данных. Изучаем класс PostView.*

### Контроллер и Index, Edit виды.

*Тут изучаем состояние дел в контроллере (постраничный вывод и поддержка многоязычности). Изучаем виды Index и Edit.*

### Маленький подытог

*Тут опишем какие задачи мы поставили перед собой и как мы собираемся их решить. Создание ManageAttribute. Как будет происходить разработка пошагово.*

### Создание скаффолдеров. Ключевые моменты.

*Создаем скаффолдер, изучаем параметры. Выборка типов полей по атрибутам. Изучение ProviderRepository. Создаем скаффолдер для Model. Дополняем поля модели созданными атрибутами. Скаффолдер для Controller Index и Edit.*

### Проверка

*Создаем новый многоязыковой класс Vine (список вин). Проходим все стадии – БД, репозиторий, скаффолдинг в репозитории, в модели, добавление атрибутами, создание контроллера, создание записи, уборка атрибутов, добавление валидации.*

## Урок 14. Unit-тесты

Цель: Научиться создавать юнит-тесты для кода. Использование юнит-тестирования NUnit (принципы названия метод-[входные данные]-результат). Принцип применения TDD. Mock. Рефакторинг. JUnit.

### Установить NUnit

Установить и пройти краткое содержание TDD Kata (работа со строками). Правила именования тестов. Параметры тестов.

Именовать:

*Method*\_*Scenario*\_*ExpectedBehavior*

* Method – метод [или свойство], который тестируем
* Scenario – сценарий, который мы тестируем
* ExpectedBehavior – ожидаемое поведение

Init -> Act -> Assert

### Mock

Разобраться

### Рефакторинг

Принципы рефакторинга, книга Фаулера – использовать.

### JUnit

Разобраться

### Подключение отдельно конфига

## Урок 15. Создание приложения

Цель: завершающий урок по созданию приложения. Приложение «риэлторское агенство». Анализ необходимых вещей для создания. Написание технического задания. Создание БД. Переименование webTemplate. Применение скаффолдинга. Админка. Основной сайт. Тесты.

### Техническое задание

*Составляем тезисы, раскрываем их. Пишем техническое задание, начинаем с цели проекта. Описываем сущности, разбивая на модули. Расписываем сроки и бюджет.*

### Структура базы

*Создаем структуру БД по техническому заданию.*

### webTemplate и Scaffolding

*Переименовываем проект, применяем скаффолдинг. Запускаем. Тестируем в админке.*

### Основной сайт

*Выводим данные по квартирам, поиск по ключевым позициям. Вывод на карте. Выбор в избранное. Покупка телефонов.*

### Тесты

*Написание юнит-тестов для api-поиска.*

## Идеи

1. TrimModelBinder

//Trim all fields

protected override void SetProperty(ControllerContext controllerContext,

ModelBindingContext bindingContext,

System.ComponentModel.PropertyDescriptor propertyDescriptor,

object value)

{

if (propertyDescriptor.PropertyType == typeof(string))

{

var stringValue = (string)value;

if (!string.IsNullOrEmpty(stringValue))

{

stringValue = stringValue.Trim();

}

value = stringValue;

}

base.SetProperty(controllerContext, bindingContext, propertyDescriptor, value);

}

1. Поддержка абстрактных классов из View

public override object BindModel(ControllerContext controllerContext, ModelBindingContext bindingContext)

{

var modelType = bindingContext.ModelType;

if (modelType.IsAbstract)

{

var modelTypeValue = controllerContext.Controller.ValueProvider.GetValue("**ModelTypeName**");

if (modelTypeValue == null)

{

throw new Exception("View does not contain ModelTypeName");

}

var modelTypeName = modelTypeValue.AttemptedValue;

var type = modelType.Assembly.GetTypes().SingleOrDefault(x => x.IsSubclassOf(modelType) && x.Name == modelTypeName);

if (type != null)

{

var instance = bindingContext.Model ?? base.CreateModel(controllerContext, bindingContext, type);

bindingContext.ModelMetadata = ModelMetadataProviders.Current.GetMetadataForType(() => instance, type);

}

}

return base.BindModel(controllerContext, bindingContext);

1. }