

Оглавление

Урок 0. Вступление	5
Почему именно asp.net mvc	5
Для работы нам понадобится	5
Оглавление	5
Урок 1. Начало	6
Начало	6
Global.asax	11
Package Manager Console	12
NLog	13
WebActivator	15
Log2Console	16
Урок 2. Dependency Injection	18
Unity	20
Autofac	21
Castle Windsor	21
Маленький подитог	22
Объекты областей (Ninject)	22
Lifetime Manager в Unity	22
Использование Ninject в asp.net mvc	23
DependencyResolver	26
Итог	26
Урок 3. Работа с БД	27
Что такое БД	27
Создание простой схемы в БД	28
SELECT, INSERT, UPDATE, DELETE	32
LinqToSQL и Linq	33
Создание репозитория IRepository , SqlRepository	37
Сниппеты	38
Proxy	41
Использование БД в asp.net mvc	42
Урок 4. Routing и Bundles	44
Controller и Action	44

BaseController	44
Порядок объявления маршрутов.....	46
Ограничения (Constrains)	47
Areas.....	47
Урок 5. Создание записи в БД.....	50
Введение.....	50
Регистрация	50
Валидация.....	53
Automapping	55
Captcha.....	62
Урок 6. Авторизация.....	67
Кукисы.....	67
Авторизация	68
Урок 7. Bootstrap, jQuery, Ajax.....	78
Twitter Bootstrap и css.....	78
Структура html-страницы.	81
Структура js файлов	84
Минификация ресурсных файлов	85
Установка jQuery.....	89
Firebug (Firefox) и Developer Tool (Chrome).....	89
Селекторы и обход	95
События	96
Атрибуты и значения.....	97
Основные манипуляции.....	97
Ajax.....	97
Ajax-login форма.	98
Итог	102
Урок 8. View, Razor, страница ошибки.	103
Основа	103
Razor.....	103
PageableData.....	105
Helper (PagerHelper).....	107
SearchEngine	110
Extension	112
Динамические формы	115

Правила перенаправления.....	118
Другие ActionResult	121
RssActionResult.....	121
Урок 9. Configuration и загрузка файлов	123
IConfig (и реализация).	123
Создание своих типов ConfigSection	125
MailSettings	126
Простая загрузка файлов.....	129
Загрузка файла (ов) с помощью Ajax.....	132
Создание превью.....	135
Получение файлов по ссылке.....	137
Урок А. Уведомление и рассылка	138
SmtpClient и MailNotify	138
Более сложный случай	141
SmsNotify.....	143
Отдельный поток.....	145
Урок В. Json.....	147
Json и Json.net.....	147
Документация.....	148
Работа с facebook.....	152
Клиентский код/Серверный код (Access-Control-Allow-Origin)	158
Урок С. Многоязычный сайт.....	162
Проблемы многоязычного сайта	162
Routing	163
Ресурсы сайта	164
База данных.....	169
Админка.....	175
Переключение между языками	185
Неверный формат, перевод на русский	186
Итог	186
Урок D. Scaffolding	187
Scaffolding. Начало.....	187
T4.....	188
MVCscaffolding.....	190
Снова шаблоны, EnvDTE.CodeType.....	195

Описание скаффолдеров.....	199
Итог	200
Урок Е. Тестирование	201
Тестирование, принцип TDD, юнит-тестирование и прочее.	201
Установить NUnit	203
Mock.....	205
TestConfig	212
Authentication	216
MockHttpContext.....	217
Проверка валидации.....	219
Проверка авторизации.....	223
Интегрированное тестирование	225
Генерация данных	231
Итог	233
Урок F. Работа как она есть	234
О главном	234
О принципах	234
Техническое задание.....	236
Структура базы, webTemplate и Scaffolding	237
Собственный ритм.....	238

ASP. NET MVC

Урок 0. Вступление.

Я пишу сайты на asp.net mvc. В этих 16 главах я хочу рассказать, как я это делаю. Это некий учебник-справочник всех тех знаний, которые я накопил в течение трех лет.

Почему именно asp.net mvc

ASP.NET MVC я люблю потому что:

- Это .net. Я знаю .net и C#.
- Это компилируемый код.
- Это не ASP.NET WebForms, я работаю с html-кодом.
- Используется MVC-паттерн.
- Visual Studio – самое популярное средство разработки, в котором есть IntelliSense (что это? Очень краткое определение).
- Отличные инструменты отладки.

Всё это позволяет мне быстро и грамотно разрабатывать приложения. Главное – быстро.

Для работы нам понадобится

1. Visual Studio 2012 с установленным asp.net mvc 4 (<http://www.asp.net/mvc/mvc4>)
2. MS SQL Server для работы с БД (<http://www.microsoft.com/en-us/sqlserver/editions/2012-editions/express.aspx>)
3. Умение работать с Mercurial или Git (<http://habrahabr.ru/post/108443/>)
4. Знания по C#

Оглавление

- Урок 1. Мы просто создадим и запустим проект. И немного изучим NuGet, NLog и Logger.
- Урок 2. Изучение Dependency Injection. Изучим различные реализации. Ninject, Unity, Autofac
- Урок 3. Работа с БД. SQL-команды. LinqToSql
- Урок 4. Маршруты и связки. Структура asp.net mvc – приложения
- Урок 5. Создание записи в БД через веб-интерфейс. Валидация данных. Automapping
- Урок 6. Авторизация (и почему мы не используем стандартный MembershipProvider)
- Урок 7. Html, css, Bootstrap, jquery. Справочные данные о клиентской части
- Урок 8. View, Razor. Изучаем View-engine Razor. Дополняем наше приложение страницей с обработкой ошибки
- Урок 9. Configuration, и работа с загрузкой файлов. Обработка изображений
- Урок А. Работа с почтой и sms
- Урок В. Json, работа с этим форматом. Json.net
- Урок С. Создание мультиязычного сайта

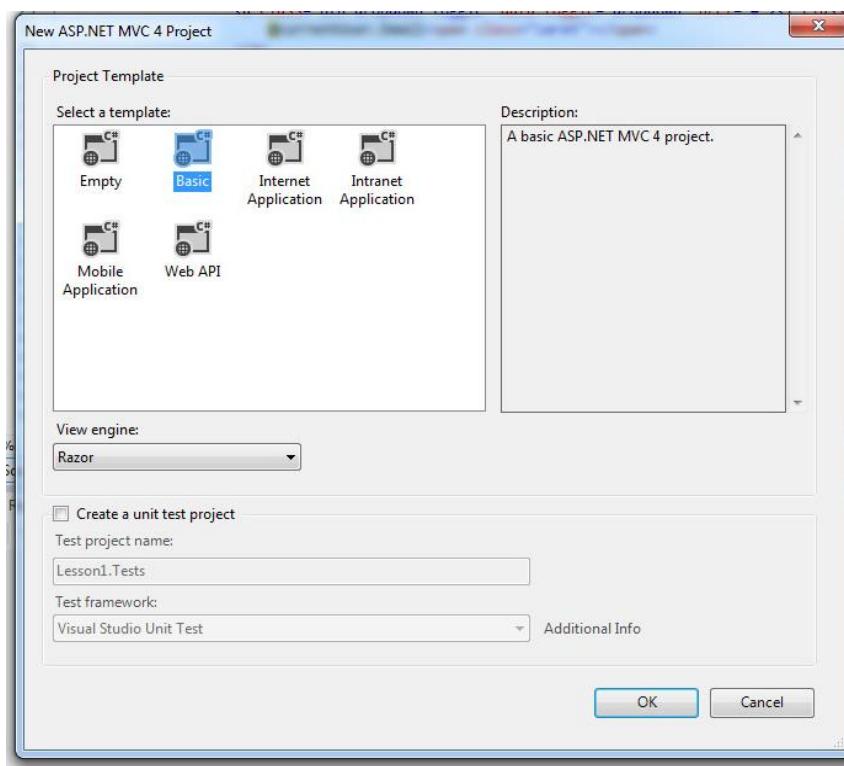
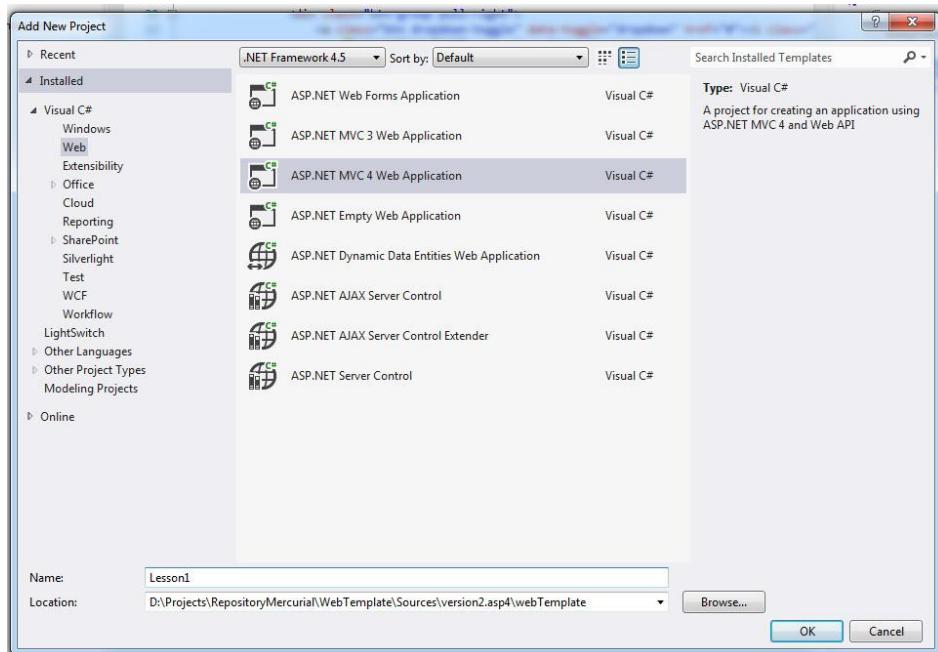
- Урок D. Scaffolding
- Урок E. Тестирование
- Урок F. Работа как она есть. Мои принципы работы. Как писать ТЗ.

Урок 1. Начало

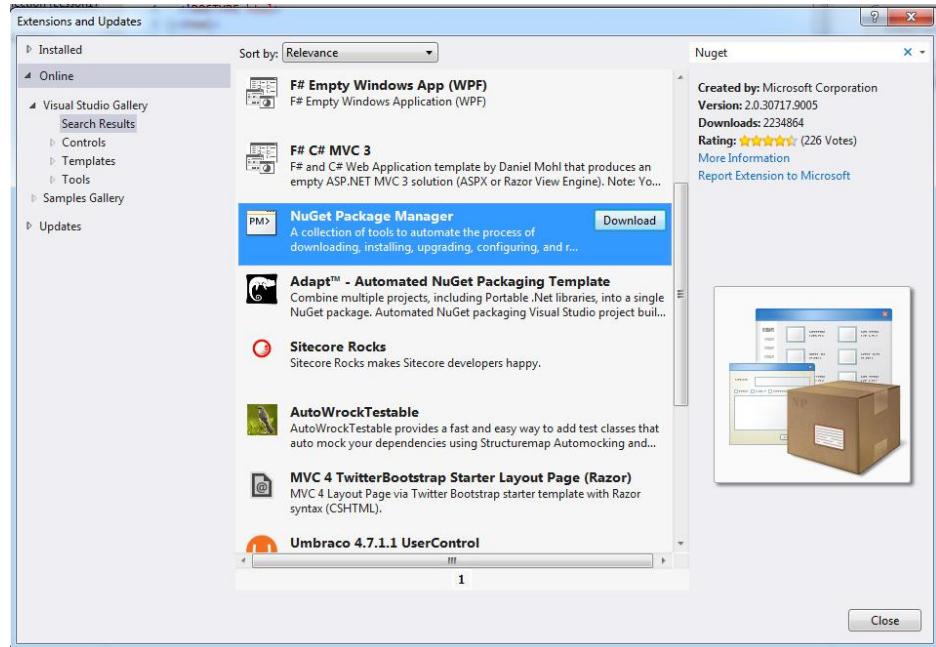
Цель урока: Изучить Global.asax и поведение запуска веб-приложения, обработки веб-запроса.
Изучение Nuget и Подключение протоколирования.

Начало

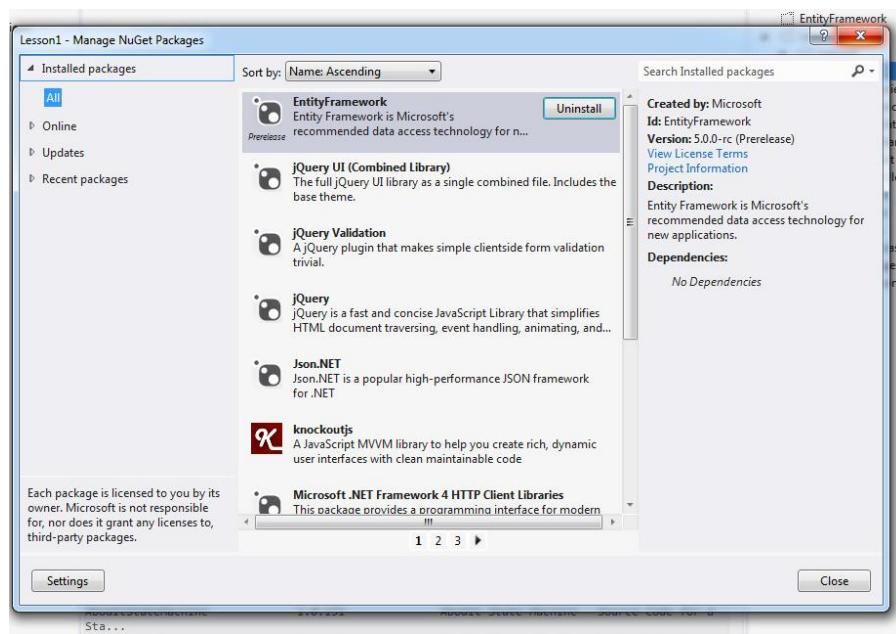
Создадим приложение ASP.NET MVC 4 Web Application «Lesson1».



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 mvc4 приложения в само приложение было добавлено изначально много модулей. Их список мы можем найти, кликнув в Manage NuGet Packages... в контекстном меню проекта



О них по порядку:

- **Entity Framework** – обеспечивает работу с БД
- **jQuery (+ jQuery UI, jQuery Validation)** – популярный javascript framework (о нем еще пойдет речь позже).
- **Json.NET** – классы для работы с json-форматом данных.

- **knockoutjs** - не очень 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 HttpContext-ом.
- **Microsoft ASP.NET MVC 4** – собственно, классы паттерна проектирования MVC (то что есть предметом изучения)
- **Microsoft ASP.NET Razor 2** – view-движок. Есть еще ASPX и Spark – мы о них также будем говорить позже
- **Microsoft ASP.NET Universal Providers Core Libraries (Microsoft ASP.NET Universal Providers for LocalDB)** – предоставляет инструменты 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 так далее
- **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 ненавязчивого ajax/валидации
- **Microsoft.Web.Infrastructure** – позволяет динамически регистрировать HTTP-модули во время выполнения
- **Modernizr** – js-библиотека, которая позволяет использовать html5 и css3 в старых браузерах
- **WebGrease** – позволяет минифицировать html, css, js.

Теперь, когда мы примерно прикинули, из чего состоит наше приложение, давайте запустим его.

Для этого необходимо создать HomeController:

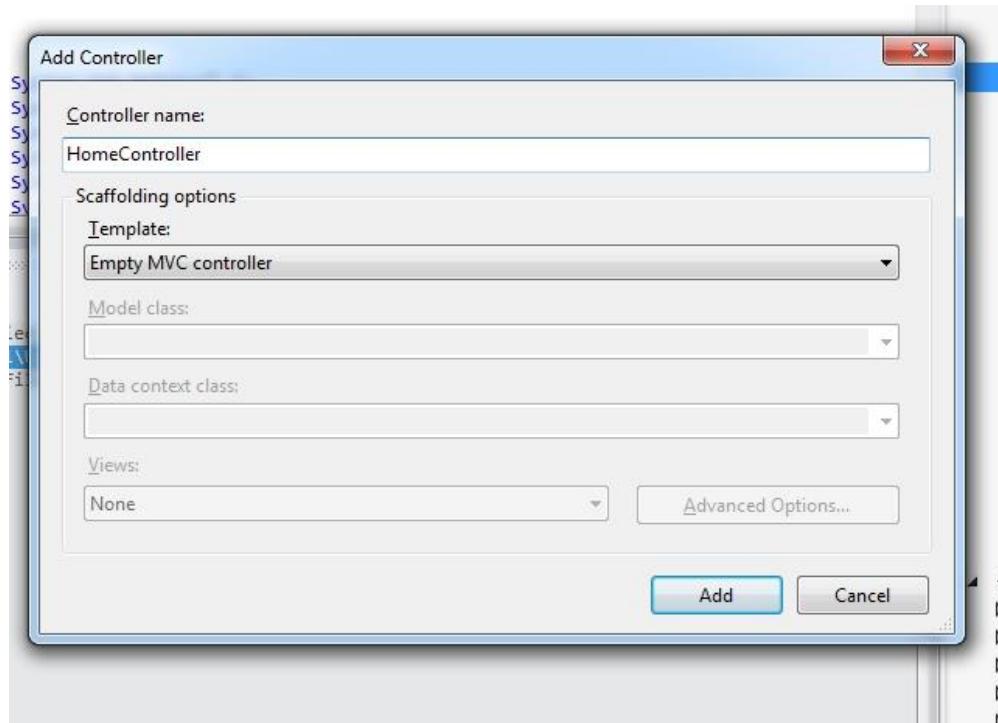
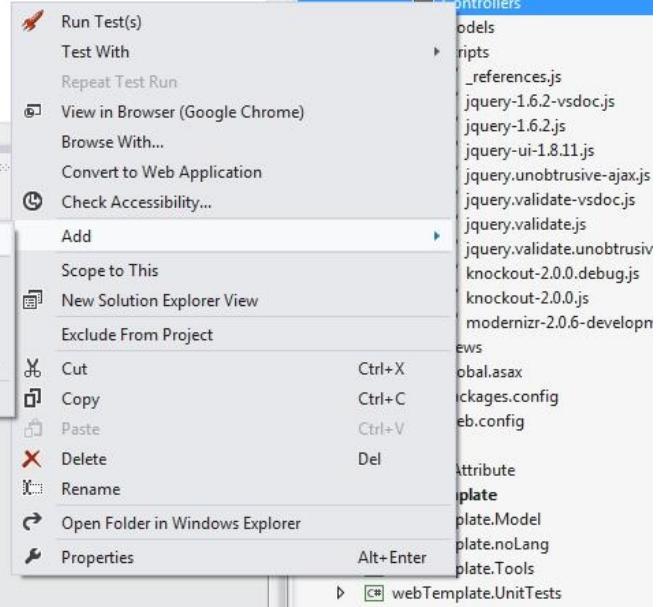
```
btrusiveJavaScriptEnabled" value="true" />

debug="true" targetFramework="4.5" />
on mode="Forms">
nUrl="~/Account/Login" timeout="2880" />
ion>

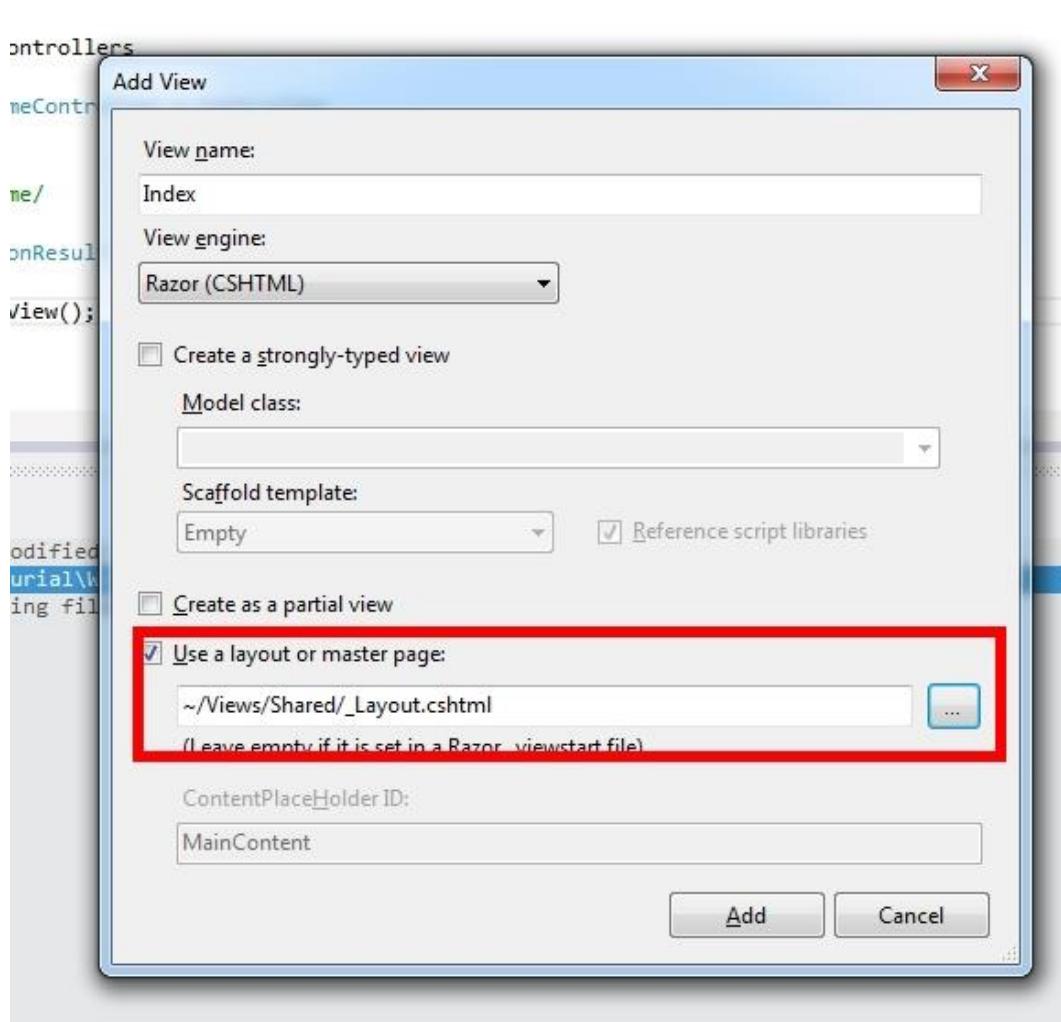
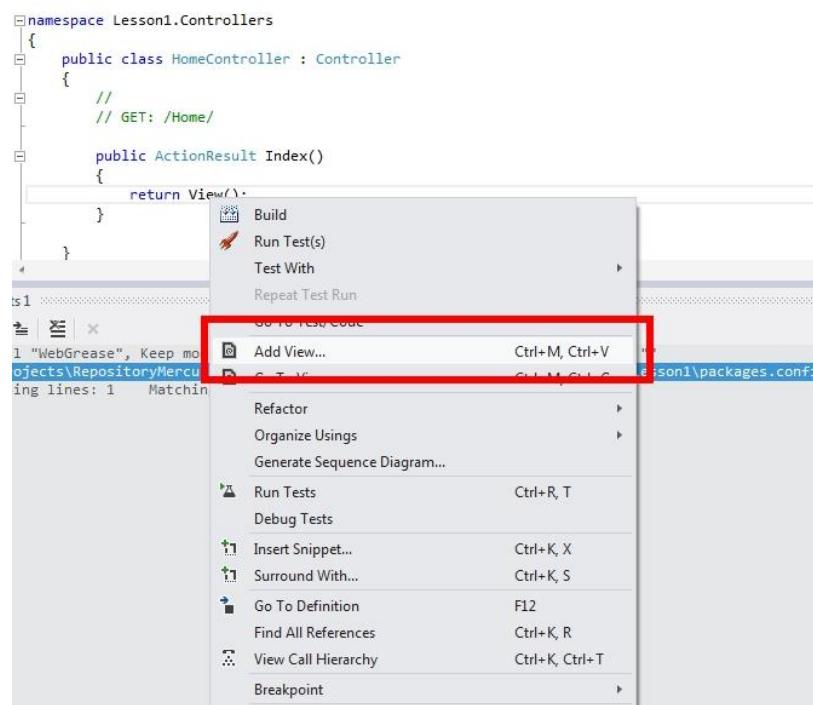
>
space="System.Web.Helpers" />
space="System.Web.Mvc" />
space="System.Web.Mvc.Ajax" />
space="System.Web.Mvc.Html" />
space="System.Web.Optimization" />
space="System.Web.Routing" />
!!!
!!!
```

modified files open, F
rcurial\WebTemplate\Sou
ching files: 1 Total

- Controller... Ctrl+M, Ctrl+C
- New Item... Ctrl+Shift+A
- Existing Item... Shift+Alt+A
- New Folder
- Add ASP.NET Folder
- Class... Shift+Alt+C

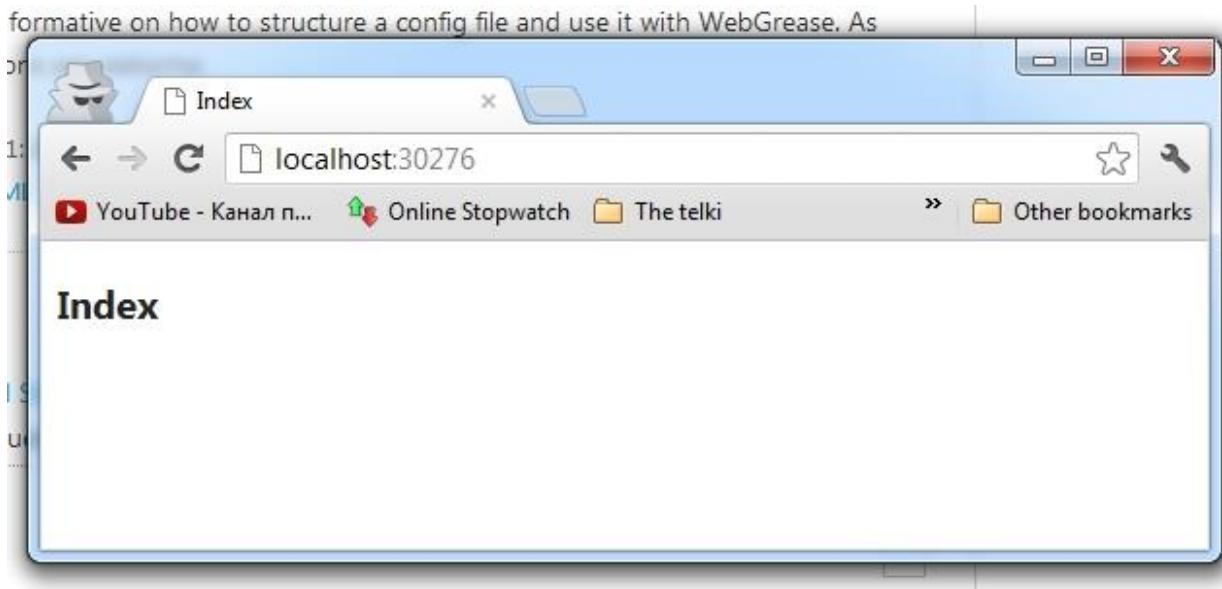


Для метода (действия) Index добавим View и выберем ~Views/Shared/_Layout.cshtml в качестве layout (типа master page):



Собственно, можем запускать.

Всё что мы увидим – это:



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

Global.asax

А сейчас 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 течение 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.
- **Application_PresendRequestHeaders** – наступает перед тем, как ASP.NET посыпает клиенту (браузеру) HTTP заголовки.
- **Application_PresendContent** – наступает перед тем, как ASP.NET посыпает 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 действительную. В текущий 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.

Хорошо. Теперь, чтобы воочию убедимся, 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 [скэфволдинга](#).

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

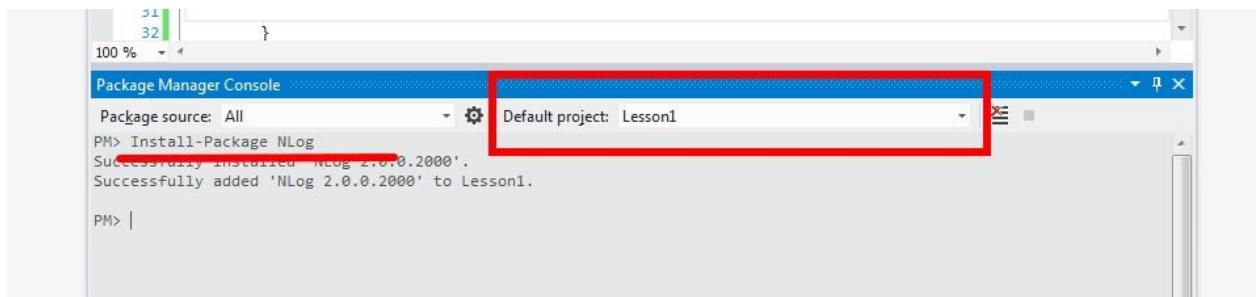
```
Get-Package -ListAvailable -Filter NLog
```

Или

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

Для установки модуля NLog необходимо вначале выбрать проект (если их в солюшнене 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. 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 завершим приложение (Stop). Откроем файл C://file.txt. Мы увидим, какие события происходили.

```

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 регистрация:

- Area (область),
- Filter (фильтров),
- Bundle (комплекты),

- Route (маршруты).

Подробности по инициализации Filter, Bundle 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, к примеру, 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
```

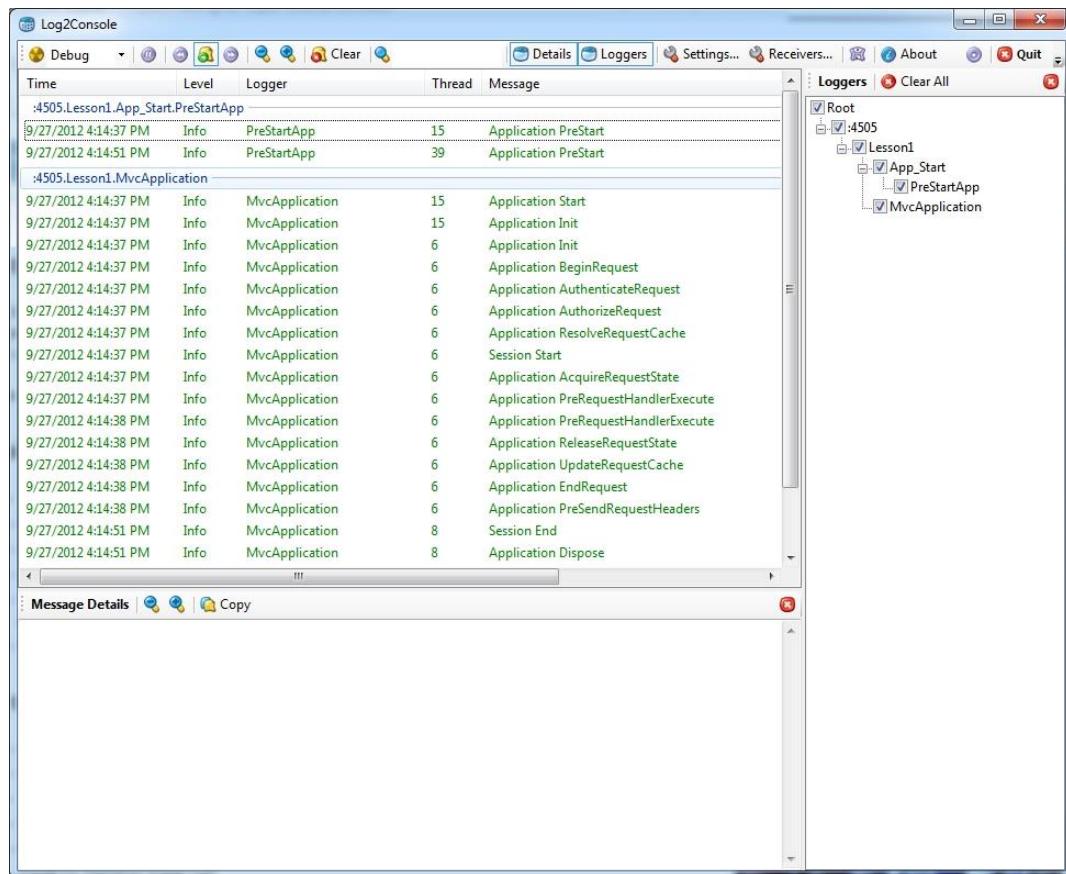
```
2012-09-19 10:29:01.6290|INFO|Lesson1.MvcApplication|Application Start
```

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

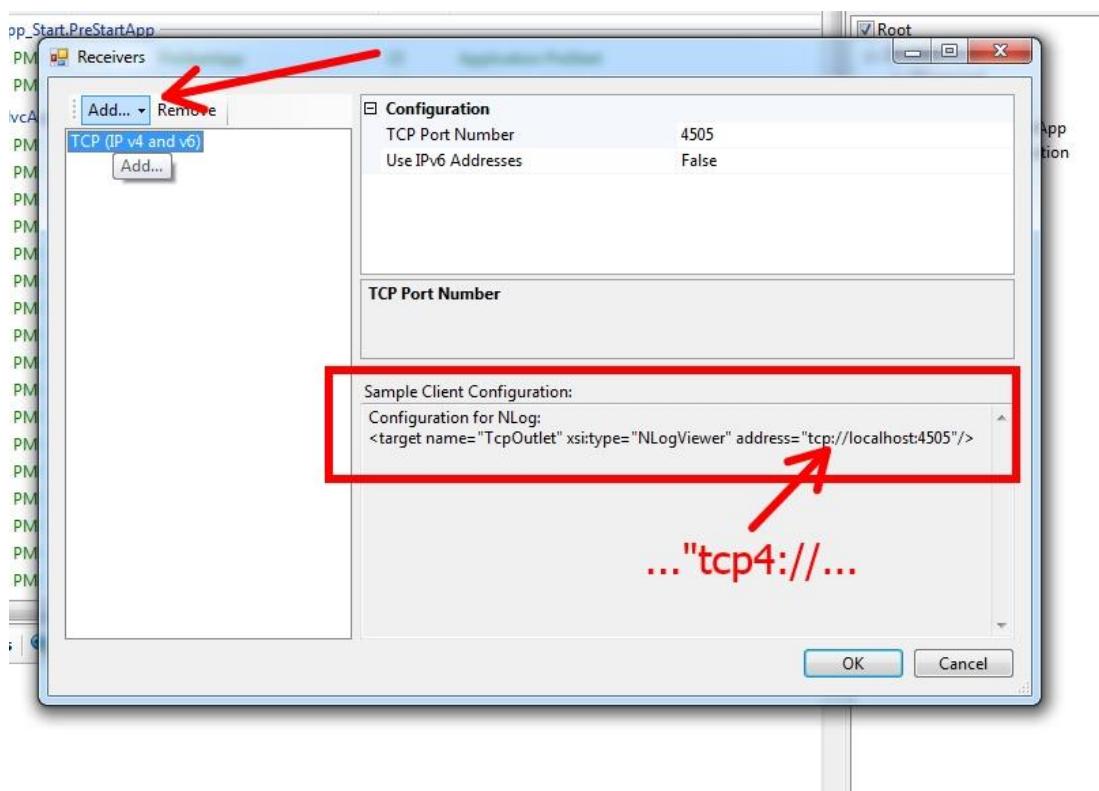
```
<nlog autoReload="true" xmlns="http://www.nlog-project.org/schemas/NLog.xsd"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <variable name="logDirectory" value="${basedir}/Content/logs/${shortdate}" />
    <targets>
        <target name="fileLogTrace" xsi:type="File" fileName="${logDirectory}/trace.txt" />
        <target name="fileLogDebug" xsi:type="File" fileName="${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 окне 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. Dependency Injection

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

Во многих случаях, один и тот же экземпляр класса используется в вашем приложении в разных модулях. Простым способом реализации является применение шаблона Одиночка (Singleton).

Но рассмотрим эту ситуацию с другой стороны. Так как данный объект создается при первом обращении к нему, мы не можем контролировать его время жизни. При модульном тестировании (unit-test) нет необходимости использовать этот объект (или это может быть невозможно). Чтобы избежать этого, мы не напрямую вызываем объект, а через интерфейс. И реальный экземпляр класса, и экземпляр-заглушка для тестирования будут реализовывать этот интерфейс. А логику создания мы поручаем DI-контейнеру.

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 нет проверки на null в строке

```
Weapon.Kill();
```

Что здесь некоректно? 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 онного 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!

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

Так 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 все необходимые классы, потому что создание экземпляров незарегистрированных классов тут не реализован.
builder.RegisterType<Bazuka>();
builder.RegisterType<Warrior>();

builder.RegisterType<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. Аналогична как и в Autofac.
container.Register(Component.For<IWeapon>().ImplementedBy<bazuka>(),
Component.For<Warrior>().ImplementedBy<Warrior>());

• Получение объекта и использование:

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

### Маленький подитог

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. 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 каждому требованию (метод по умолчанию). |
| Одиночка  | .InSingletonScope() | Объект класса будет создан один раз и будет использоваться повторно.        |
| Поток     | .InThreadScope()    | Один объект на поток.                                                       |
| Запрос    | .InRequestScope()   | Один объект будет 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 так:

```
_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. Все объекты хранятся в `HttpContext.Current.Items[_key]` и выдаются только, если уже находятся в том же контексте (`HttpContext.Current`). В ином случае, создается новый объект. Если текущий контекст (`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>

```

Запускаем – работает.

*Примечание: В дальнейшем мы будем переносить этот проект в последующие уроки.*

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

```
Install-Package Ninject.MVC3
```

Добавляем класс в папку App\_Start:

```
[assembly:
WebActivator.PreApplicationStartMethod(typeof(LessonProject.App_Start.NinjectWebCommon),
"Start")]
[assembly:
WebActivator.ApplicationShutdownMethodAttribute(typeof(LessonProject.App_Start.NinjectWeb
Common), "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 добавляем инициализацию 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:



## LessonProject

BIG BADABUM!

Ninject использует WebActivator:

- регистрирует свои модули OnePerRequestHttpModule и NinjectHttpModule
- создает StandartKernel
- инициализирует наши сервисы.

## DependencyResolver

В asp.net mvc3 появился класс DependencyResolver. Этот класс обеспечивает получение экземпляра сервиса. Наши зарегистрированные сервисы (и даже используемый DI-контейнер) мы также можем получить посредством этого класса.

```
public class HomeController : Controller
{
 private IWeapon weapon { get; set; }

 public HomeController()
 {
 weapon = DependencyResolver.Current.GetService<IWeapon>();
 }

 public ActionResult Index()
 {
 return View(weapon);
 }
}
```

## Итог

Использование DI-контейнеров в современных приложениях необходимо, чтобы избавиться от сильной связности кода, и для легкого доступа из любой его части к сервисам. Также, это необходимо для 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). Команды INSERT, UPDATE, DELETE, SELECT. Использование LinqToSql и Linq. Создание репозитария, IRepository, SqlRepository.

### Что такое БД

Реляционная база данных — база данных, основанная на реляционной модели данных. Реляционность — это отношения (связи) от англ. relation.

### Таблицы

Это таблица:

| CategoryID | ParentID | Title                    | SortOrder |
|------------|----------|--------------------------|-----------|
| 1          | 0        | Electronics              | 9835      |
| 2          | 1        | Mobile Phones            | 10000     |
| 3          | 1        | DVD Systems              | 10100     |
| 4          | 2        | Sony Ericsson            | 10000     |
| 5          | 2        | Nokia                    | 10100     |
| 6          | 2        | Motorola                 | 10200     |
| 7          | 2        | Samsung                  | 10300     |
| 8          | 0        | Apparel                  | 100       |
| 9          | 8        | John Players             | 10000     |
| 10         | 8        | Women Sarees             | 10100     |
| 11         | 9        | Shirts                   | 10000     |
| 12         | 9        | Pants                    | 10100     |
| 13         | 10       | Banarasi Sarees          | 10000     |
| 14         | 10       | Kurta Salwar             | 10100     |
| 16         | 0        | Beverages                | 9975      |
| 17         | 0        | Computer and Accessories | 55555     |
| 18         | 0        | Baby Items               | 10400.2   |
| 19         | 0        | Health and Beauty        | 10400.4   |
| 20         | 0        | Jewelry                  | 10193.85  |

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

Таблицы должны 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, однозначно определяющих уникальность строки. Обычно это одно поле, называется ID. Оно является автоинкрементным, 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.

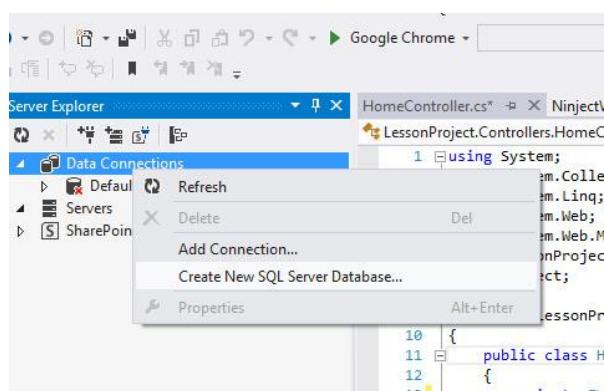
Связи бывают трех типов:

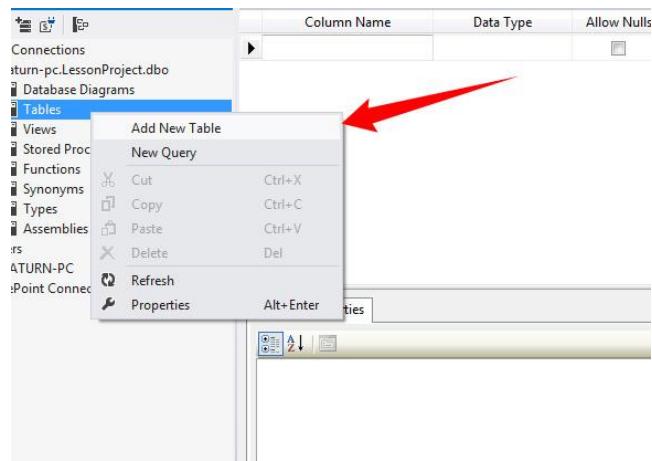
- Один-к-одному. Т.е. одной строке в таблице соответствует одна строка в другой таблице. Это редко используется, но используется. Например, в одной таблице данные о пользователе, а в другой — дополнительные данные о том же пользователе. Такой вариант необходим, чтобы манипулировать, по необходимости,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

Сейчас разберемся, как это делать.

## Создание простой схемы в БД

## Создадим БД в VS 2012:





| Column Name | Data Type    | Allow Nulls                         |
|-------------|--------------|-------------------------------------|
| ID          | int          | <input type="checkbox"/>            |
| Code        | nvarchar(50) | <input type="checkbox"/>            |
| Name        | nvarchar(50) | <input checked="" type="checkbox"/> |

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:

| Column Name | Data Type | Allow Nulls                         |
|-------------|-----------|-------------------------------------|
| ID          | int       | <input checked="" type="checkbox"/> |

The context menu for the 'ID' column is open, showing options: Set Primary Key (highlighted with a red box) and Insert Column.

Задаем для ID автоинкремент:

| Column Name | Data Type    | Allow Nulls              |
|-------------|--------------|--------------------------|
| ID          | int          | <input type="checkbox"/> |
| Code        | nvarchar(50) | <input type="checkbox"/> |
| Name        | nvarchar(50) | <input type="checkbox"/> |

**Column Properties**

Collation: <database default>

Computed Column Specification:

Condensed Data Type: int

Description:

Deterministic: Yes

DTS-published: No

Full-text Specification:

Has Non-SQL Server Subscriber: No

Identity Specification:

- (Is Identity): Yes (highlighted with a blue background)
- Identity Increment: 1
- Identity Seed: 1
- Indexable: Yes
- Is Columnset: No

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:

The screenshot shows the 'dbo.UserRole' table in Object Explorer. A context menu is open over the 'UserID' column, with a red arrow pointing to the 'Relationships...' option in the list.

| Column Name | Data Type | Allow Nulls                         |
|-------------|-----------|-------------------------------------|
| ID          | int       | <input checked="" type="checkbox"/> |
| UserID      | int       | <input checked="" type="checkbox"/> |
| RoleId      |           |                                     |

Column Properties | Properties | Alt+Enter

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

The screenshot shows the 'Tables and Columns' dialog box for creating a foreign key relationship. The 'Relationship name:' field is set to 'FK\_UserRole\_Role'. The 'Primary key table:' dropdown is set to 'Role', and the 'Foreign key table:' dropdown is set to 'UserRole'. The 'Primary key column:' dropdown has 'ID' selected, and the 'Foreign key column:' dropdown has 'RoleId' selected. The 'OK' button is highlighted.

Foreign Key

Selected: FK\_UserRole\_Role

Tables and Columns

Relationship name: FK\_UserRole\_Role

Primary key table: Role

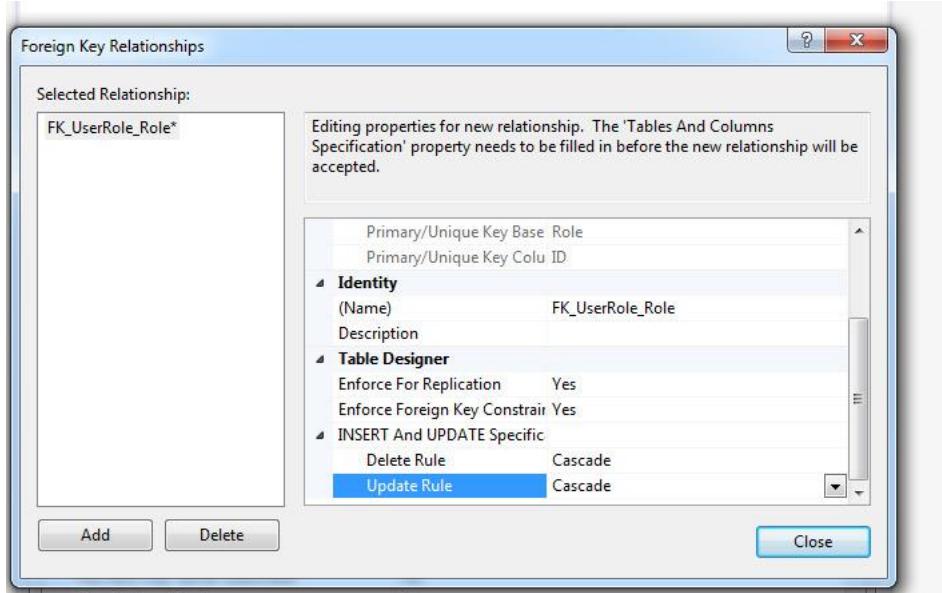
Foreign key table: UserRole

Primary key column: ID

Foreign key column: RoleID

OK Cancel

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



Это необходимо для того, чтобы при изменении/удалении столбца из таблицы Role все связанные с этой строкой строки таблицы UserRole должны быть изменены или удалены.

Аналогичную связь мы устанавливаем с таблицей User.

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

### **SELECT, INSERT, UPDATE, DELETE.**

В реляционных базах данных используется язык запросов SQL.

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

SELECT – для выбора данных и 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:

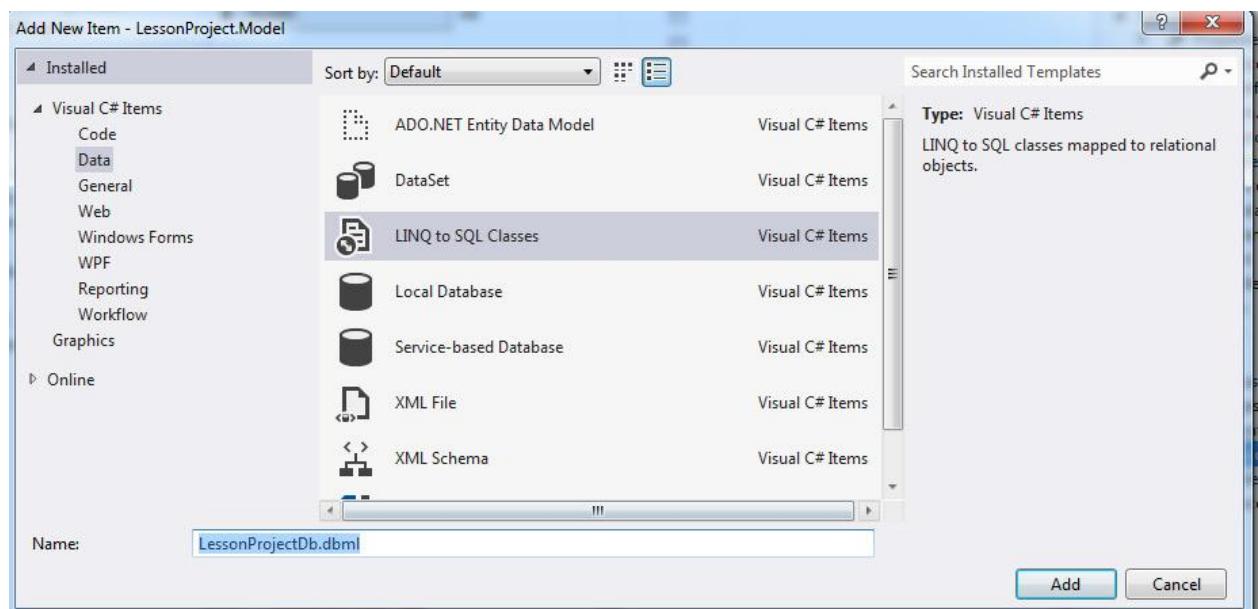
<http://www.w3schools.com/sql/>

[http://codingcraft.ru/sql\\_queries.php](http://codingcraft.ru/sql_queries.php)

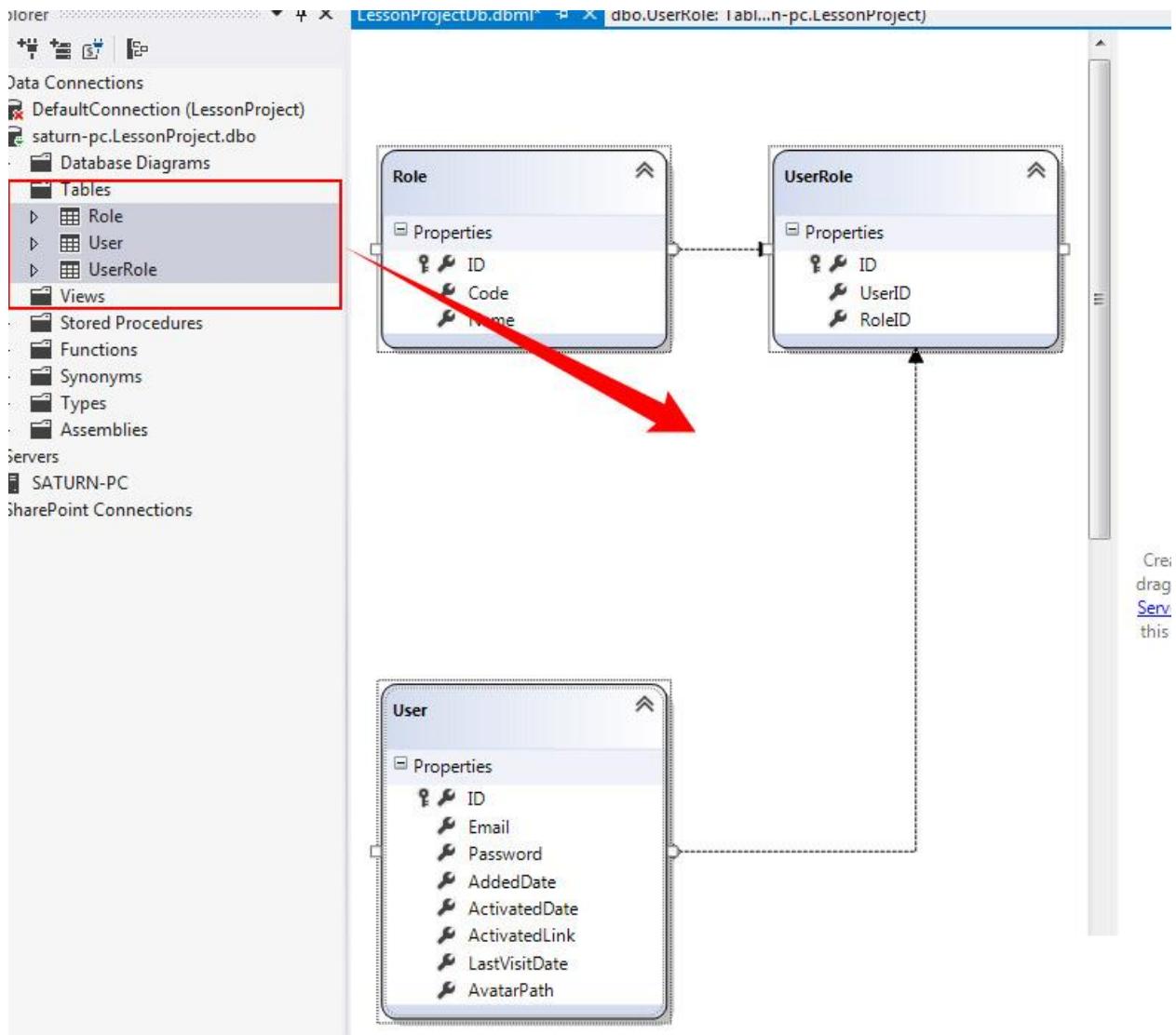
### LinqToSQL и Linq.

Создадим 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 подключим 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 IQueryble<>

.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 [default(int)], например). 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(). Лямбда-выражение должно возвращать тип 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 произвольного типа, может быть даже dynamic:

```
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")
```

*Больше примеров и вариантов использования Linq вы сможете найти: <http://code.msdn.microsoft.com/101-LINQ-Samples-3fb9811b>*

## Создание репозитория IRepository, SqlRepository.

Собственно с БД мы уже можем работать, только теперь нужно 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 к данным, а также выведены методы для создания, изменения и удаления этих данных.

```
public interface IRepository
{
 IQueryable<Role> Roles { get; }

 bool CreateRole(Role instance);

 bool UpdateRole(Role instance);

 bool RemoveRole(int idRole);

 ...
}
```

Реализацию назовем SqlRepository. Так как мы с данным контектом 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 о том, что файл этот будет выглядеть громадным и неуклюжим. Таким кодом будет управлять тяжело физически. Так что сделаем отдельную папку SqlRepository и SqlRepository класс сделаем partial, а в папке создадим реализации интерфейса IRepository, разбитые по каждой таблице.

Назовем файл Role:

```

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;
 }
}

```

Небольшой проект содержит от 10 до 40 таблиц, большой проект от 40, и всё это хотелось бы как-то автоматизировать. Создадим несколько синипетов, для IRepository и для SqlRepository.

Синипеты – это уже готовые шаблоны кода, которые вызываются с помощью intelliSense, и позволяют быстро создавать код.

## Синипеты

Для IRepository таблиц, создадим 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 снippet 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;
 }

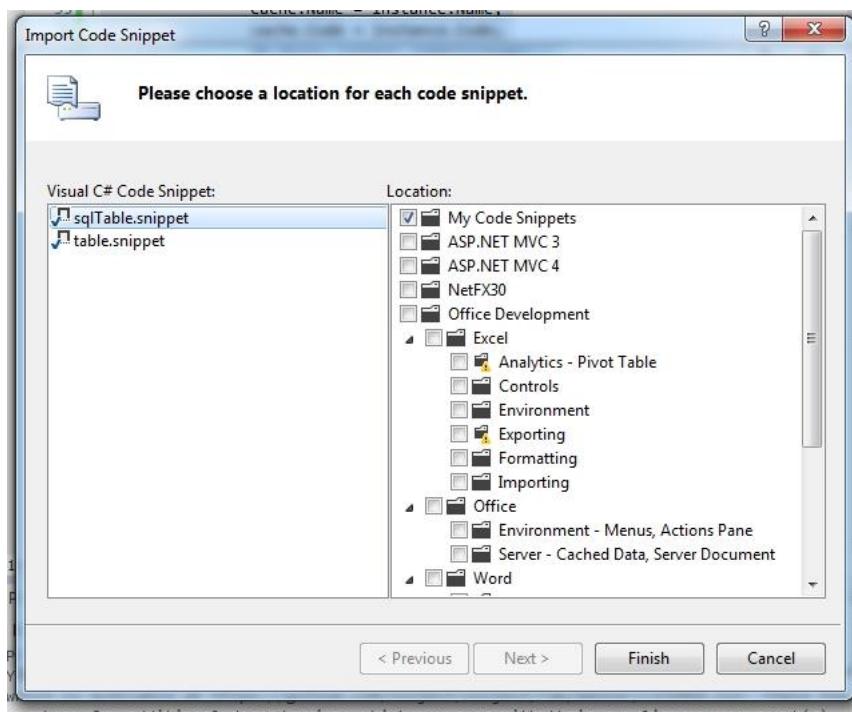
 return false;
}

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 окне нажимаем Import и импортируем оба сниппета в My Code snippet:



Finish, OK.

Используем для таблиц User и UserRole.

```

namespace LessonProject.Model
{
 public partial class SqlRepository
 {
 } } LinqToSqlShared
 SqlRepository
 sqltable
 }

public partial class SqlRepository
{

 public IQueryable<Table> Tables
 {
 get
 {
 return Db.Tables;
 }
 }

 public bool CreateTable(Table instance)
 {
 if (instance.ID == 0)
 {
 Db.Tables.InsertOnSubmit(instance);
 Db.Context.SubmitChanges();
 return true;
 }

 return false;
 }

 public bool UpdateTable(Table instance)
 {
 Table cache = Db.Tables.Where(n => n.ID == instance.ID).First()
 !!!
}

```

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

## Proxy

Как видим, 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 БД 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<LessonProjectDbContext>().ToMethod(c => new
LessonProjectDbContext(ConfigurationManager.ConnectionStrings["ConnectionString"].ConnectionString));
 kernel.Bind< IRepository>().To<SqlRepository>().InRequestScope();
}

```

Применяем InRequestScope(). Т.е. каждый запрос будет использовать отдельный объект SqlRepository. Это позволит избежать коллизий при исполнении. 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">

 @role.ID

 @role.Name

 @role.Code

 </div>
 }

```

```

 </div>
}
</p>
```

Получаем хороший результат:



## LessonProject

- 1 Админ admin
- 2 Пользователь customer
- 3 Менеджер manager

## 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 хотим иметь.

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

```
routes.MapRoute(
 name: "Default",
 url: "{controller}/{action}/{id}",
 defaults: new { controller = "Home", action = "Index", id =
UrlParameter.Optional }
);
```

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: 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, первоначально 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">
 Роли
 @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">

 @role.ID

 @role.Name

 </div>
 }

```

```


 @role.Code

 </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, третий – controller, четвертым(или пятым) параметром идут другие атрибуты тега, если мы хотим добавить другие параметры для маршрутизации – мы должны явно указать пятым 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:

```
Пользователь под номером 1
```

### Порядок объявления 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 }
);
```

Стока “roli/{action}/{id}” однозначно задает имя контроллера в секции defaults. А action является параметром.

Результат. Ссылка стала:

```
Роли
```

Уберем из defaults action="Index":

```
Роли
```

Поместим после объявления “Defaults”:

```
Роли
```

Такая ссылка получилась, потому что вышестоящим правилом маршрута “default” можно задать путь к Role/Index:

```
context.MapRoute(
 name : "default",
 url : "{controller}/{action}/{id}",
 defaults : new { controller = "Home", action = "Index", id =
UrlParameter.Optional },
);
```

Еще надо рассмотреть один метод, который 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 (Constraints)

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:

```
Роли
```

Будет:

```
Роли
```

А для:

```
Роли
```

Будет:

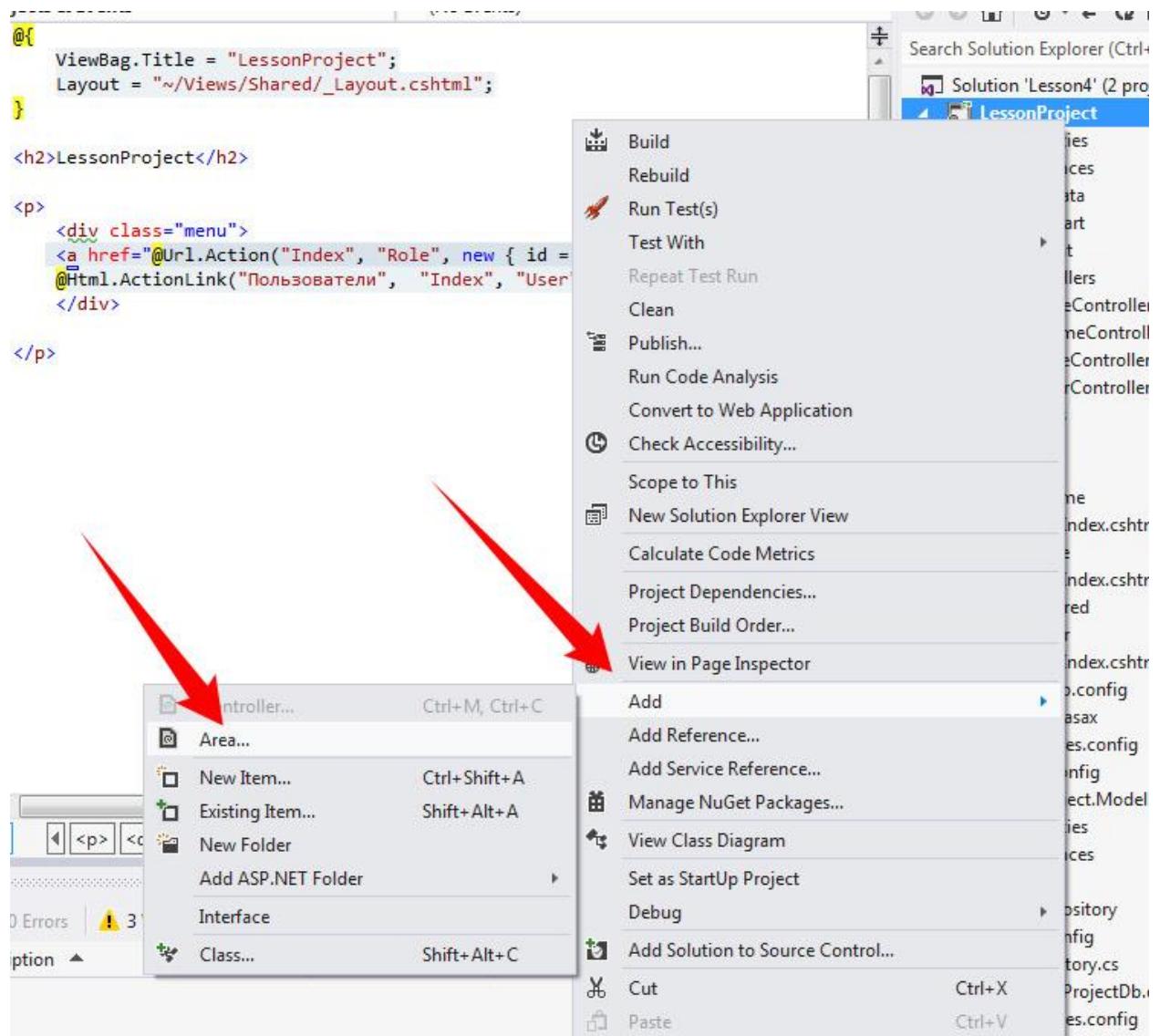
```
Роли
```

*Примечание: Более подробно об ограничениях можно узнать тут:*

<http://stephenwalther.com/archive/2008/08/07/asp-net-mvc-tip-30-create-custom-route-constraints.aspx>

## Areas

Чтобы разделить различные 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:

Здесь 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 раньше 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.

А где итог????

## 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, переходим к одному из самых важных уроков, в котором будет рассказано про создание записей. Любое действие на сайте, от сложных, когда мы заполняем регистрационную анкету, до простых, когда ставим лайк, – происходит следующим образом:

- 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 показала, что эти данные верны, то они сохраняются в БД и выводится страница с подтверждением.

### Регистрация

Сделаем форму для регистрации пользователя. При регистрации, 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 два, 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>
 </div>
 </fieldset>
}
```

```

 @Html.ActionLink("Cancel", "Index", null, null, new { @class = "btn" })
 </div>
</fieldset>
}

```

Все эти дивы дивные, fieldset и button'ы 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 такая есть

```
@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:

```

var newUser = new User();
return View(newUser);
}

[HttpPost]
public ActionResult Register(User user)
{
 return View(user);
}

```

The screenshot shows the Visual Studio IDE. On the left is the code editor with the following C# code:

```

var newUser = new User();
return View(newUser);
}

[HttpPost]
public ActionResult Register(User user)
{
 return View(user);
}

```

On the right is the Object Browser window titled "user (LessonProject.Model.User)". It lists the properties of the User class. The properties "Email" and "Password" are highlighted with red boxes.

Property	Value
_ActivatedDate	null
_ActivatedLink	null
_AddedDate	{1/1/0001 12:00:00 AM}
_AvatarPath	null
_Email	Q "chernikov@gmail.com"
_ID	0
_LastVisitDate	null
_Password	Q "123456"
_UserRoles	{System.Data.Linq.EntitySet<LessonProject.Model.UserRole>}
ActivatedDate	null
ActivatedLink	null
AddedDate	{1/1/0001 12:00:00 AM}
AvatarPath	null
Email	Q "chernikov@gmail.com"
ID	0

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

Так как мы должны принять еще 2 поля (повтор пароля и капчу), то добавим эти 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>

```

```
</div>
</div>
```

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

## Валидация

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 - partial, то мы можем реализовать для него IValidatableObject интерфейс, для этого, правда, придется добавить в проект System.ComponentModel.DataAnnotations. 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.

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 4 из 6 правил валидации, но оставим пока так, а остальные добавим непосредственно в контроллер.

Выполняем форму, получаем:

## Register

The screenshot shows a registration form with the following fields and their states:

- Email:** Labeled "Введите email". The input field is highlighted with a red border, indicating it is invalid.
- Password:** Labeled "Введите пароль". The input field is highlighted with a red border, indicating it is invalid.
- Confirm Password:** Labeled "Confirm Password". The input field is empty.
- Captcha:** Labeled "Тут картинка 1234". Below it is a text input field containing "1234".
- Buttons:** Two buttons at the bottom: "Register" (disabled) and "Cancel".

Видим, что обе наши ошибки были отловлены.

Есть два стандартных метода вывести ошибку: это `Html.ValidationMessage("ErrorField")` и `Html.ValidationSummary()`. Первый выводит ошибку, связанную с конкретным неверно 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:

## Register

- Пользователь с таким email уже зарегистрирован
- Введите пароль
- Текст с картинки введен неверно

Email  
Пользователь с таким email уже зарегистрирован

Password  
Введите пароль

Confirm Password

Captcha  
Тут картинка 1234  
Текст с картинки введен неверно

Что ж, с задачей мы справились, но в дальнейшем, используя такой способ, мы получим несколько проблем:

- Класс User всегда будет содержать проверку на необходимость введения 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.

Но есть решение, мы создаем класс, который является представлением класса User, организующим 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, стоит заметить, 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 количестве объектов и полей – это рутинно, к тому же, подобное у нас уже есть в функции Update[Table] в репозитории. Для решения этой задачи 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 при разработке программы мы избегаем сильную связность, то организуем интерфейс + реализацию и зарегистрируем это в Ninject, после чего 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 объектодиночка):

```
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; }
}
```

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:

## Register

- Введите email
- Введите пароль
- Текст с картинки введен неверно

Email

Password

Confirm Password

Captcha  
Тут картинка 1234

Мы смогли описать тут 5 из 6 правил валидации. Правила, касающиеся верного введенного email – нет. Напишем 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);
 }
}
```

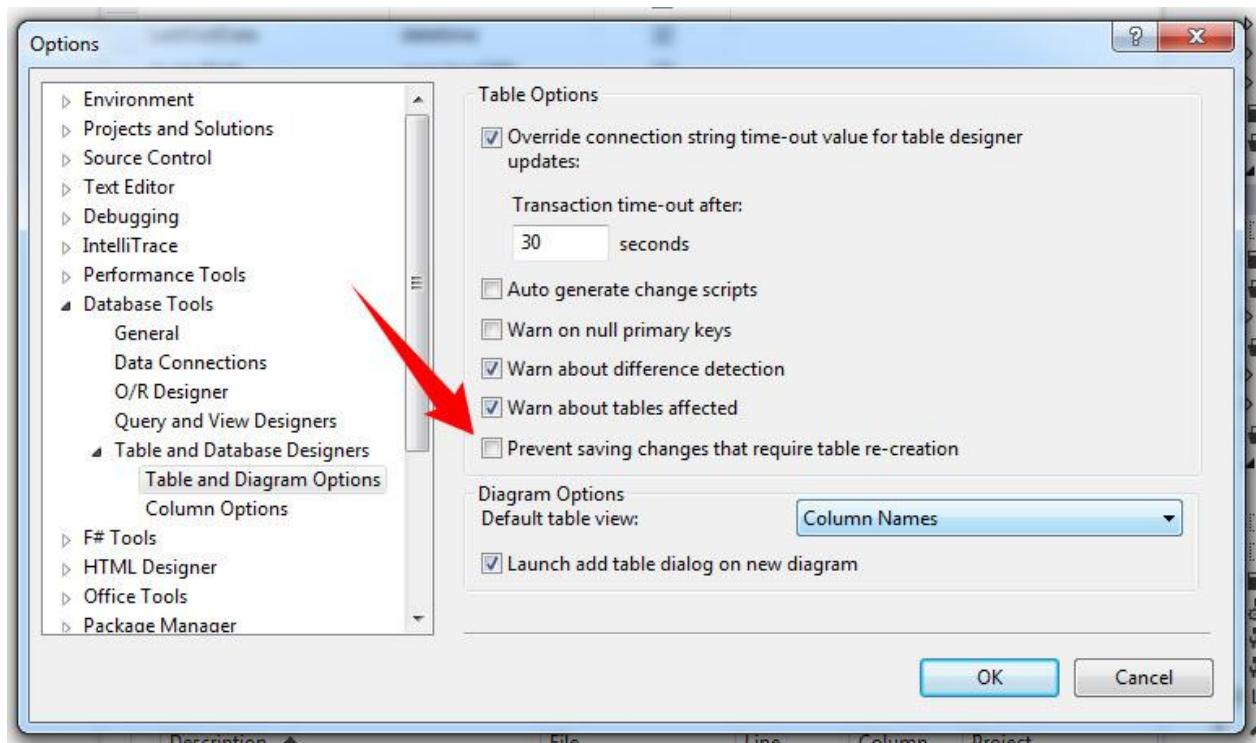
Вначале проверяем, что полученный объект есть строка, и строка не пустая, иначе возвращаем значение «истина» в проверке. Тут срабатывает правило, что «мы у инопланетян документы не проверяем», т.е. пока нет достаточных условий для проверки – мы не проверяем, а проверять будут другие атрибуты. Потом же, 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/>)*

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

- Добавляем поле в БД. Birthdate datetime null.

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



- В данных выставим всем записям значения 2012-1-1
- Изменим поле Birthdate на datetime not null
- Удаляем из LessonProjectDb.dbml таблицу User и заново переносим из Server Explorer
- 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;
}
```

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

Выбор дня рождения у нас будет таким:

Тут надо решить несколько задач. Первая из них – создание и организация выпадающего списка. В Html (который мы еще позже рассмотрим подробнее) 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; }
}
```

Для выбора, например, из 1 - apple, 2 – orange (выбран), 3 - banana мы должны написать следующий код:

```
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, которому присвоится значение Value при подтверждении (сабмите) формы.

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 брейк-поинт point на приеме данных. Проверим, как мы получаем данные для полей даты рождения для объекта UserView:

## Register

Email  
Password  
Confirm Password  
Birth date  
12 September 1983  
Captcha  
Тут картинка 1234  
Register Cancel

```
 n UserView();
 iev);

ster(UserView userView)
 l= "1234")
 delError("Captcha", "Te
tory.Users.Any(p => str
 delError("Email", "Пол
 id)
)ModelMapper.Map(userView, typeof(userView), typeof(user));
};
};
```

Теперь осталось правильно передать их в объект User. Опишем эту передачу в 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 дальше еще поговорим. 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;
}
}

```

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 здесь происходит:

- В сессии создаем случайное число от 1111 до 9999.
- Создаем в ci объект **CaptchaImage**
- Очищаем поток вывода
- Задаем header для mime-типа 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">

</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 она 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 объясно это с точки зрения «нажимай и не понимай, что там внутри». При первом знакомстве с asp.net mvc меня это смущило. Далее, в этой статье <http://habrahabr.ru/post/143024/> - сказано, что пользоваться этим провайдером – нельзя. Здесь 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, 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
```

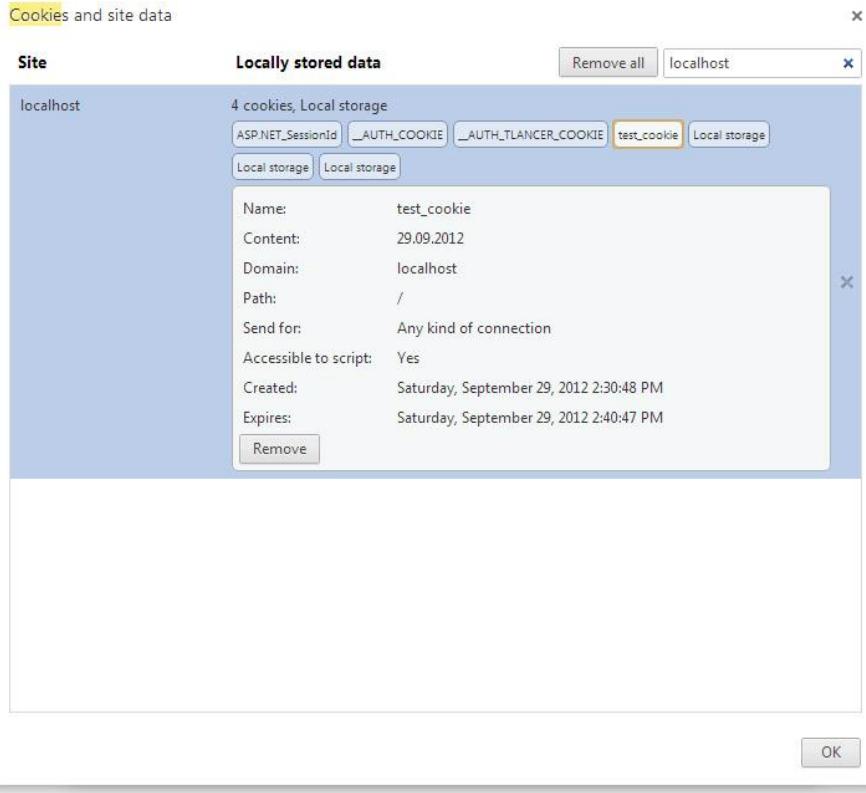
Браузер (если не истекло время действия кукиса) при каждом запросе:

```
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 остановки и проверяем:

```
public ActionResult Index()
{
 /*var cookie = new HttpCookie("test_cookie")
 {
 Value = DateTime.Now.ToString("dd.MM.yyyy"),
 Expires = DateTime.Now.AddMinutes(10),
 };
 Response.SetCookie(cookie);
*/
 var cookie = Request.Cookies["test_cookie"];

 var value = cookie.Value;
 return View();
}
```

*Примечание: подробнее можно изучить кукисы по следующим ссылкам:*

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

## Авторизация

В нашем случае авторизация будет основана на использовании кукисов. Для этого изучим следующие положения:

- **FormsAuthenticationTicket** – мы воспользуемся этим классом, чтобы хранить данные авторизации в зашифрованном 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 куникам)
 /// </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 UserIdentity 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 UserIdentity();
 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 значение «истина» (доступ есть). Сравниваем по полю Role.Code, а не Role.Name.

Рассмотрим класс UserIdentity:

```
public class UserIdentity : 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);
}

```

Почти все готово. Выведем CurrentUser в BaseController:

```

[Inject]
public IAuthentication Auth { get; set; }

public User CurrentUser
{
 get
 {
 return ((UserIdentity)Auth.CurrentUser.Identity).User;
 }
}

```

Да, это не очень правильно, так как здесь присутствует сильное связывание. Поэтому сделаем так, введем еще один интерфейс IUserProvider, из которого мы будем требовать вернуть нам авторизованного User:

```

public interface IUserProvider
{
 User User { get; set; }
}

...

public class UserIdentity : IIdentity, IUserProvider
{
 /// <summary>
 /// Текущий пользователь
 /// </summary>
 public User User { get; set; }

 ...

 [Inject]
 public IAuthentication Auth { get; set; }

 public User CurrentUser
 {
 get
 {
 return ((IUserProvider)Auth.CurrentUser.Identity).User;
 }
 }
}

```

А 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 на событие AuthenticateRequest совершать действие авторизации:

```
public class AuthHttpModule : IHttpModule
{
 public void Init(HttpApplication context)
 {
 context.AuthenticateRequest += new EventHandler(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 кукихах, ту же моментально получает доступ к имени, по нему он в репозитории получает данные пользователя и возвращает в BaseController. Но не сразу всё, а по требованию.

Подключаем модуль в Web.config:

```
<system.web>
...
<httpModules>
 <add name="AuthHttpModule" type="LessonProject.Global.Auth.HttpModule"/>
</httpModules>
</system.web>
```

План таков:

- Наверху показываем, авторизован пользователь или нет. 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")

 </div>
 </div>
</div>

@RenderBody()
```

HomeController.cs:

```
public ActionResult UserLogin()
{
 return View(CurrentUser);
}
```

UserLogin.cshtml:

```
@model LessonProject.Model.User

@if (Model != null)
{
 @Model.Email
 @Html.ActionLink("Выход", "Logout", "Login")
}
else
{
 @Html.ActionLink("Вход", "Index", "Login")
 @Html.ActionLink("Регистрация", "Register", "User")
}
```

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";
}

Вход

@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:

- [Вход](#)
- [Регистрация](#)

### Вход

**Вход**

Email

Введите Email

Пароль

После авторизации:

- chernikov@gmail.com
- [Выход](#)

### LessonProject

[Роли](#) [Пользователи](#)

## Урок 7. Bootstrap, jQuery, Ajax

Цель: Bootstrap и дополнительный css. Определить правила работы с html, js и css файлами..

Структура js-файлов. Использование jQuery, основные моменты, изучение селекторов, событий и др. addClass, removeClass, attr, data, динамическое создание dom-объекта, аjax.

Наконец мы 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 и css

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

Подробнее тут:

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

Установим bootstrap:

```
Install-Package Twitter.Bootstrap
```

Удалим Jquery.UI:

```
Uninstall-Package Jquery.UI.Combined
```

Добавим в BundleConfig bootstrap 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-*"));

 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
- Следующие 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 { }
- Атрибуты [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, равно 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().

Теперь давайте 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 оформления 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, такие как .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 необходимо, делайте это в общем описании типа: .error {color : red }, .left { float : left }.
- 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
 - междусторочное 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")

 </div>
 </div>
 </div>

 @RenderBody()
 @Scripts.Render("~/bundles/jquery")
 @Scripts.Render("~/bundles/bootstrap")
 @RenderSection("scripts", required: false)
</body>
</html>

```

Что здесь происходит:

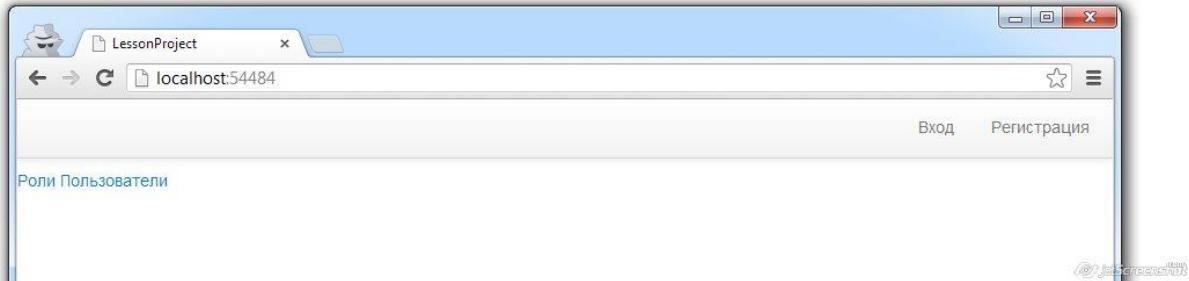
1. Мы получаем request и по запросу мы определяем маршрут /Default/Home/Index.

2. У данного контроллера/метода есть стандартный View – /Home/Index.cshtml

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

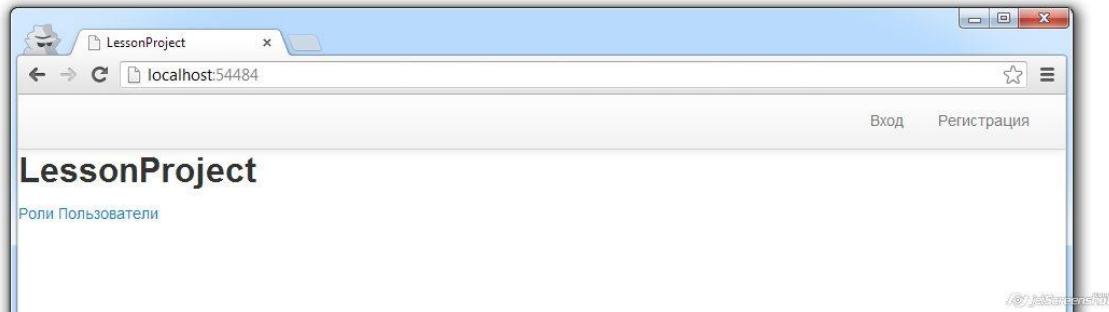
```
@{
 ViewBag.Title = "LessonProject";
 Layout = "~/Areas/Default.Views.Shared/_Layout.cshtml";
}
```

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



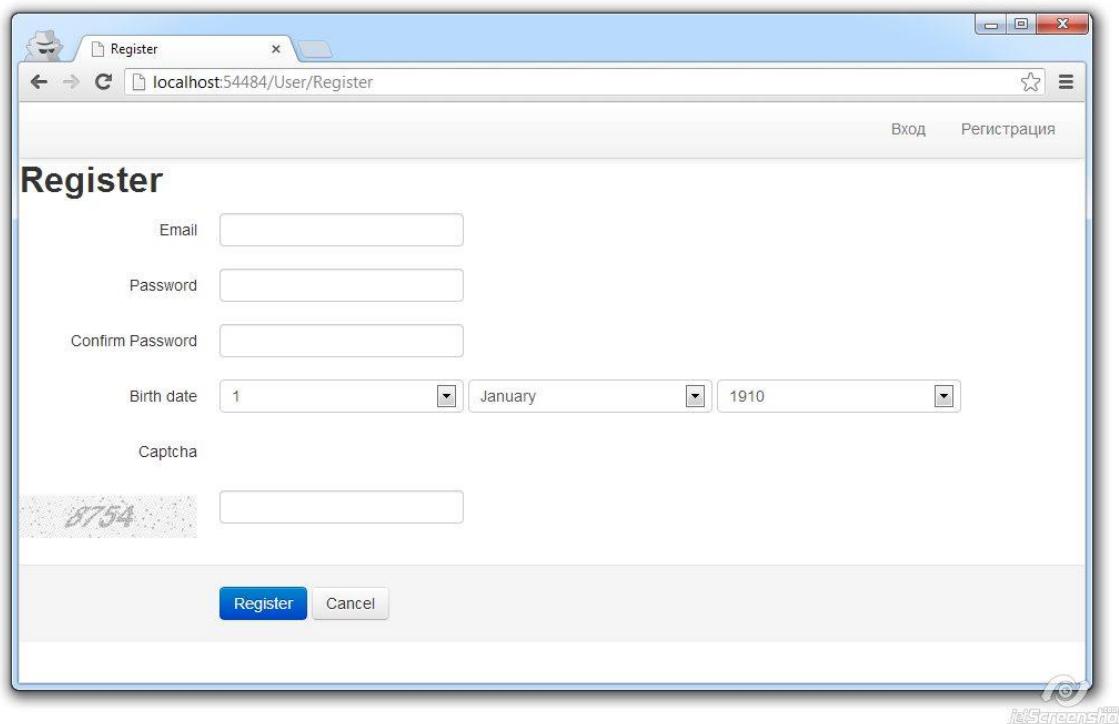
Видно, что body заехал под панель навигации. В нашем файле переопределяем body:

```
body
{
 padding-top : 40px !important;
}
```



Гораздо лучше.

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



jetScreenshot

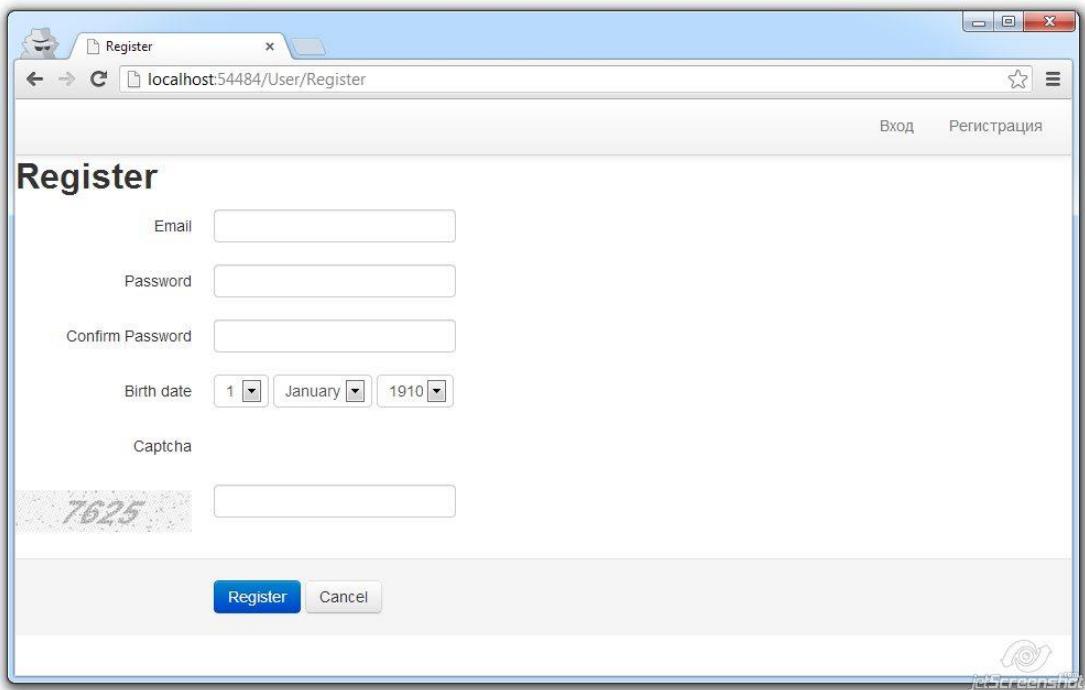
Поправим выбор даты рождения, добавим классы в @Html.DropDownList()  
(/Areas/Default/Views/User/Register.cshtml):

```
...
<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

Переходим 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 выводится в том 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("Пышь-пышь! : " + id);
 }

 /* другие приватные методы */
 function saySomething (id) {
 alert("Пышь-пышь! Но тсcc!: " + id);
 }
}

var functionName = null;
$(document).ready(function () {
 functionName = new FunctionName();
 functionName.init();
});

```

Рассмотрим подробнее:

1. function FunctionName имеет имя в стиле UpperCamelCase по имени файла, в котором находится (Common и UserRegister в файлах common.js и user-register.js соответственно)
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 использование из других файлов. (Можно использовать и из других файлов)
5. `$(document).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 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 количества запросов к ресурсным файлам
- Уменьшение ресурсных файлов.

Уменьшение количества запросов к изображениям, особенно маленьким. Это делается двумя способами:

- 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;
}

```

2. Использовать для очень маленьких изображений (иконок) объявления напрямую в css с помощью кодирования его в 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,iVBORw0KGgoAAAANSUhEUgAAAAEAAAAACAYAAQAxvaqAAAAGXRFWHRTb2Z0d2FyZQBBZG9iZSBjbWFnZVJ1YWR5c1lPAAAyJpVFh0WE1M0mNvbS5hZG9iZS54bXAAAAAAADw/eHBlY2t1dCBiZWdpbj0i77u/IiBpZD0iVzVNME1wQ2VoaUh6cmVTek5UY3prYzlkIj8+IDx40nhtcG1ldGegeG1sbnM6eD0iYWRvYmU6bnM6bWV0YS8iIHg6eG1wdGs9IkFkb2J1IFhNUCBDb3J1IDUuMC1jMDYxIDY0LjE0MDk0OSwgMjAxMC8xMi8wNy0xMD01NzowMSAgICAgICAgIj4gPHJkZjpSREYgeG1sbnM6cmRmPSJodHRwOi8vd3d3LnczLm9yZy8xOTk5LzAyLzIyLXJkZi1zew50YXgtbnMjIj4gPHJkZjpEZXXNjcmlwdGlvbibYZGY6YWJvdXQ9IiIgeG1sbnM6eG1wPSJodHRw0i8vbnMuYWRvYmUuY29tL3hhcC8xLjAvIiB4bWxuczp4bXBNTT0iaHR0cDovL25zLmFkb2J1LmNvbS94YXAvmS4wL21tLyIgeG1sbnM6c3RSZY9Imh0dHA6Ly9ucy5hZG9iZS5jb20veGFwLzEuMC9zVHlwZS9SZXNvdXJjZVJ1ZiMiIHtcE1NOKluc3RhbmN1SUQ9InhtcC5paWQ6RDhENzk1MDI1QkREMTFFMkE2QUNDMTBGODg2MjVENjAiIHtcE1NOKrVY3VtzW50SUQ9InhtcC5kaWQ6RDhENzk1MDM1QkREMTFFFMe2QUNDMTBGODg2MjVENjAiPiA8eG1wTU06RGVyaXZ1ZEZyb20gc3RSZY6aW5zdGFuY2VJRD0ieG1wLmRp1pZDpEOEQ3OTUwMDVCREQxMUUyQTZBQ0MxMEY40DYyNUQ2MCICg3RSZY6ZG9jdw11bnRJRD0ieG1wLmRpZDpEOEQ3OTUwMTVCREQxMUUyQTZBQ0MxMEY40DYyNUQ2MCIVPiA8L3JkZjpEZXXNjcmlwdGlvbj4gPC9yZGY6UkRGPiA8L3g6eG1wbWV0YT4gPD94cGFja2V0IGVuZD0iciI/P1Bqi0kAAAAwSURBVHjaYmBgYGBnghFcCIIbRPCiEvwggg8Lixf0BbN44AaACU4gZvgJIv4DBBgARTIDD2TeBRAAAAASUVORK5CYII=);
```

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.

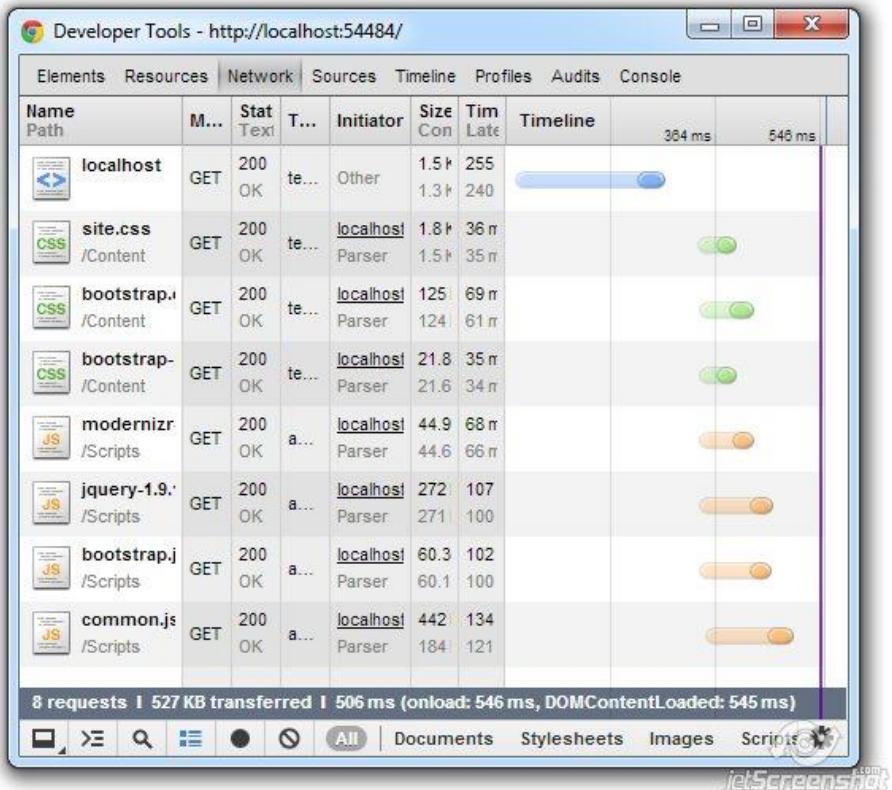
Для включения минификации необходимо в Web.config файле изменить директиву compilation:

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

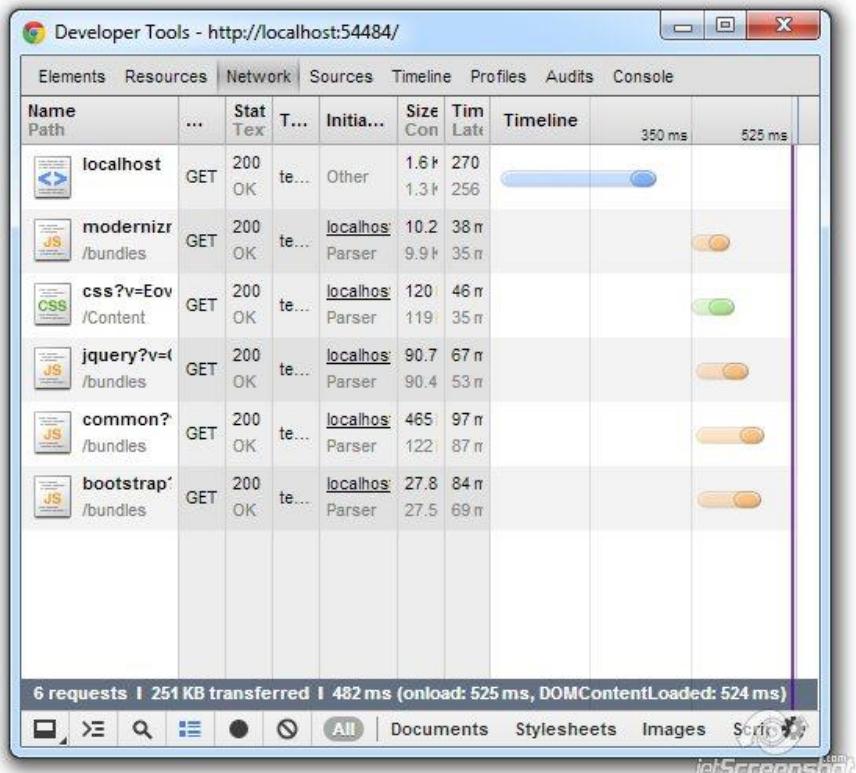
Или напрямую включить в /App\_Start/BundleConfig.cs:

```
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 от того, 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 это делается часто, то очевидно, что необходимо обновить его:

```
Install-Package JQuery
```

Далее, мы до этого убрали JQueryUI (<http://jqueryui.com/>), так как собираемся функции datepicker, modal использовать из того, что предлагает bootstrap. Но в JQueryUI есть необходимый нам функционал взаимодействия, т.е. Draggable, Droppable, Resizable, Selectable и Sortable. Установим их выбороно:

1. Так как для Install-Package JQuery.UI.Interactions необходима версия jquery < 1.6, то установим вручную.
2. Выберем custom скачивание с сайта <http://jqueryui.com/download/>
3. Выбираем только Core 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.*"));

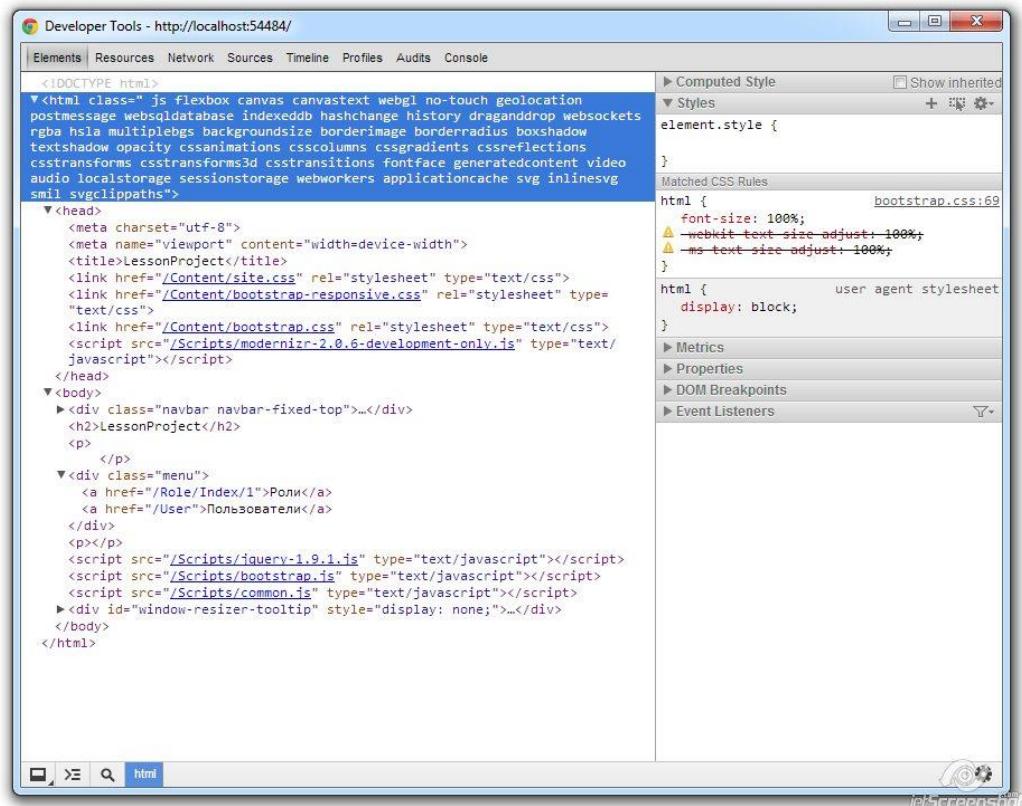
bundles.Add(new StyleBundle("~/Content/css/jqueryui")
 .Include("~/Content/jquery-ui-1*"));
```
8. При необходимости, объявляем на страницах.
9. Готово!

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

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

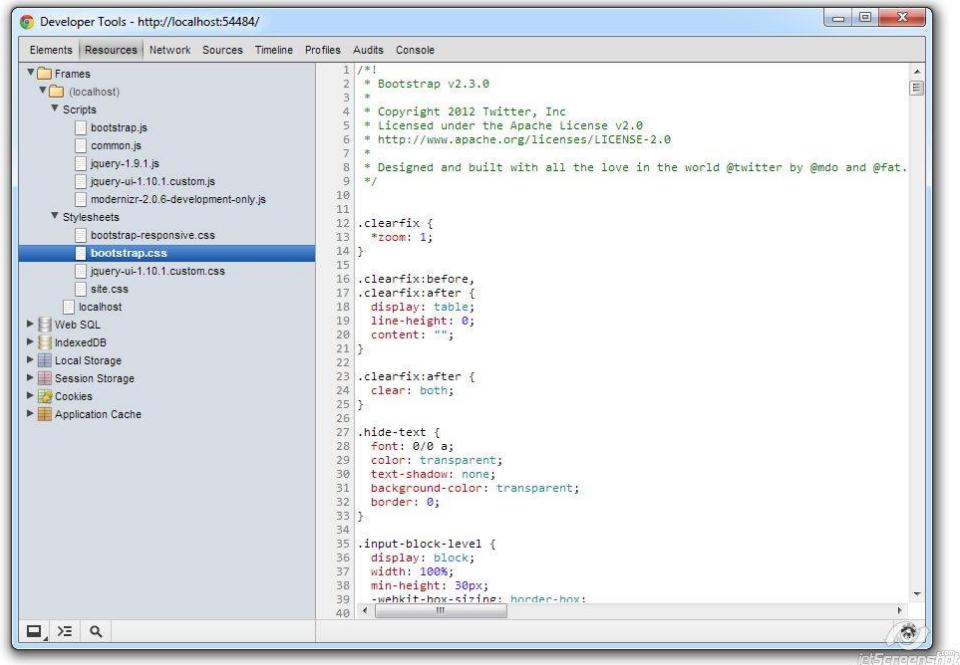
Давайте изучим его:

- Вкладка Elements.



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

- Вкладка Resources



The screenshot shows the Chrome Developer Tools Resources tab. On the left, a tree view lists resources under 'localhost' including Scripts (bootstrap.js, common.js, jquery-1.9.1.js, jquery-ui-1.10.1.custom.js, modernizr-2.0.6-development-only.js) and Stylesheets (bootstrap-responsive.css, bootstrap.css). The 'bootstrap.css' file is selected and its content is displayed on the right. The content includes Bootstrap's license notice and various CSS rules for clearing floats and styling input blocks.

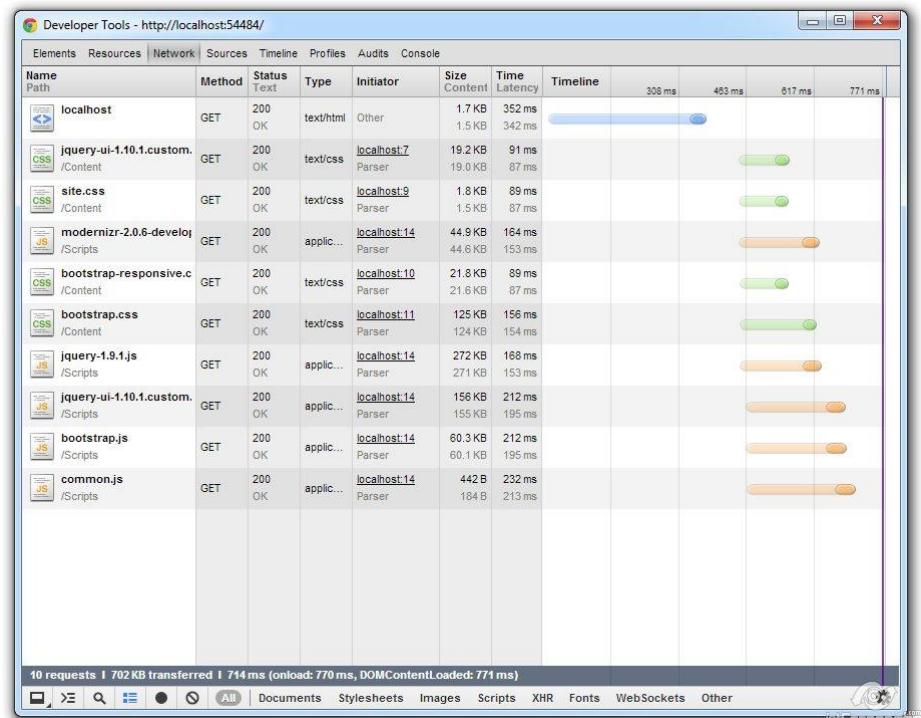
```

/*
 * Bootstrap v2.3.0
 * Copyright 2012 Twitter, Inc
 * Licensed under the Apache License v2.0
 * http://www.apache.org/licenses/LICENSE-2.0
 * Designed and built with all the love in the world @twitter by @mdo and @fat.
 */
.clearfix {
 *zoom: 1;
}
.clearfix:before,
.clearfix:after {
 display: table;
 line-height: 0;
 content: "";
}
.clearfix:after {
 clear: both;
}
.hide-text {
 font: 0/0 a;
 color: transparent;
 text-shadow: none;
 background-color: transparent;
 border: 0;
}
.input-block-level {
 display: block;
 width: 100%;
 min-height: 30px;
 -webkit-hyphens: none;
}

```

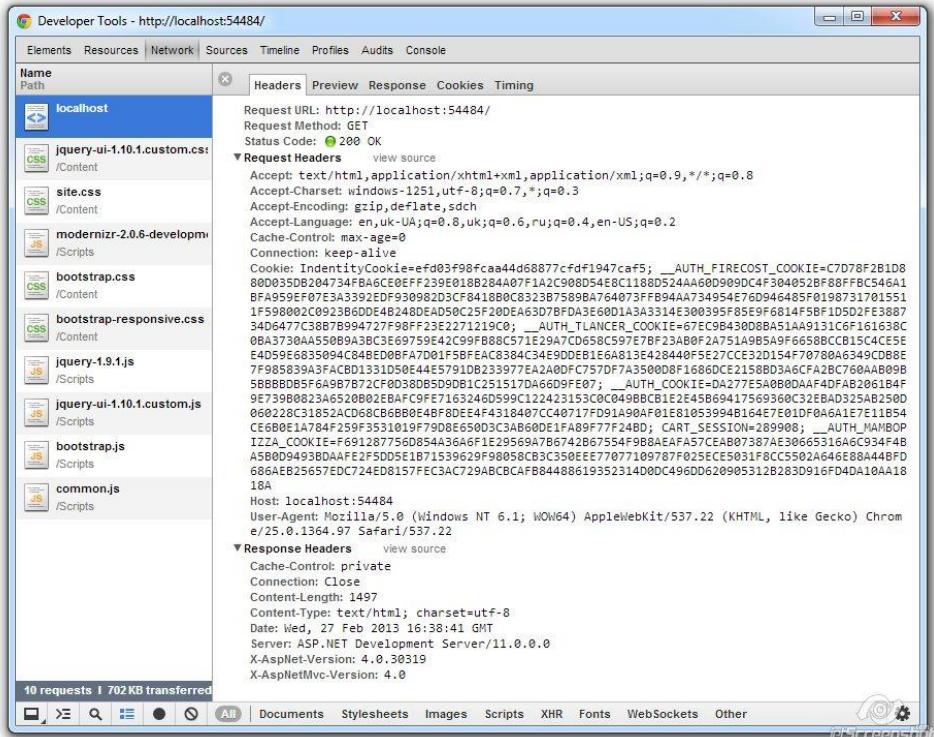
В левой части располагаются все ресурсы, которые были загружены, а в правой можно просмотреть на них

- Вкладка Network

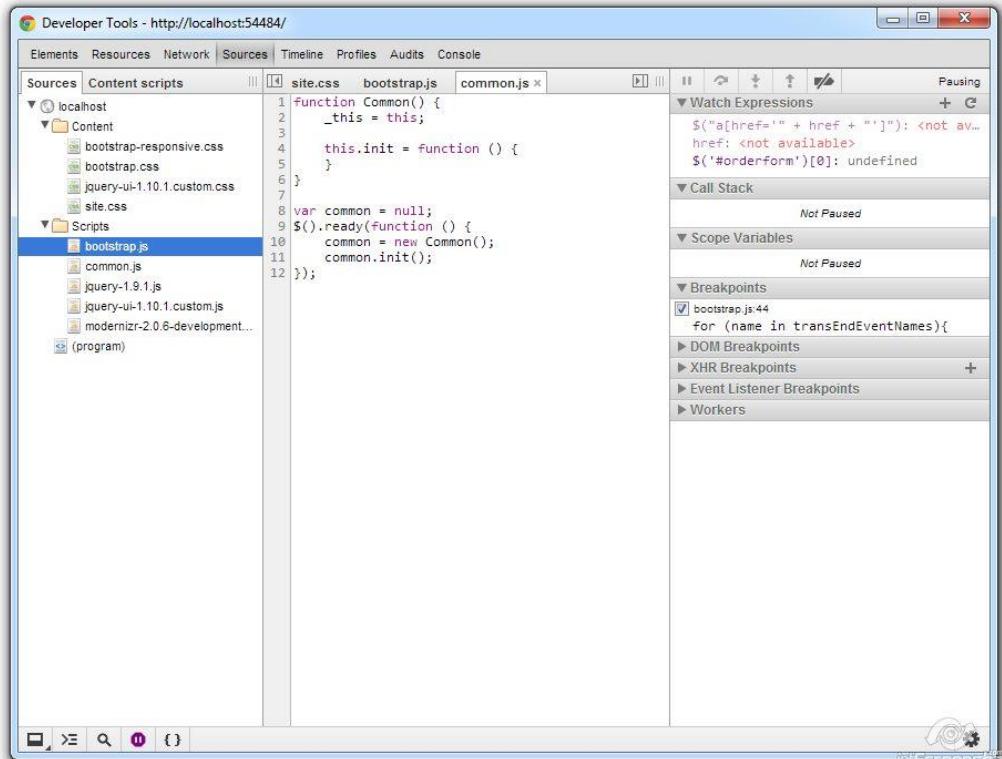


Показывает запросы и тайминг этих запросов. Обозначает разными цветами document, css, js файлы, изображения. Показывает размеры файлов.

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

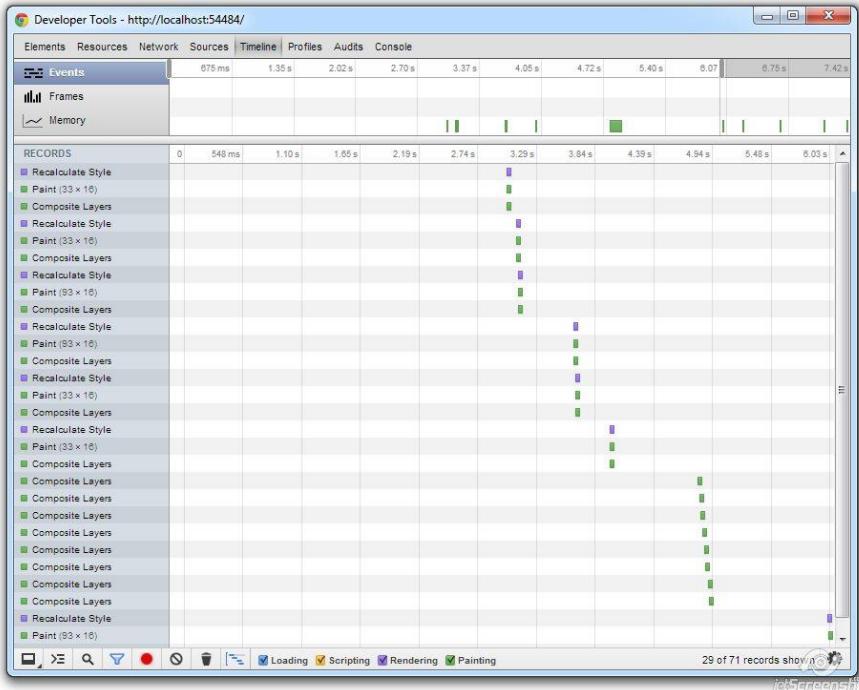


- Вкладка Sources



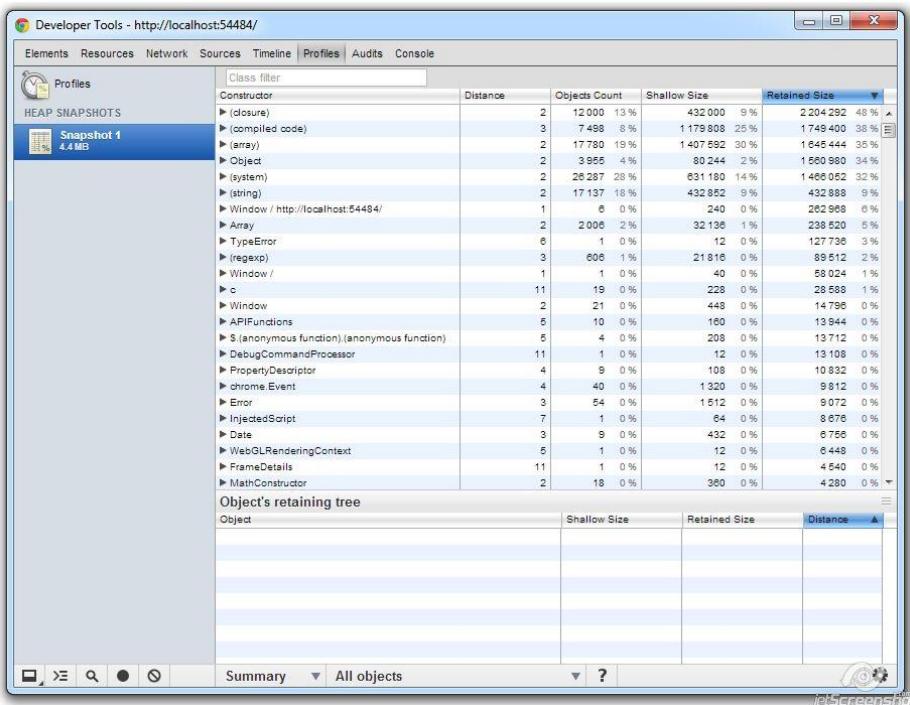
Одна из основных вкладок. Тут можно просматривать js-файлы, устанавливать точки останова, и выполнять отладку приложения. Горячие клавиши F9, F10, F11 как в VisualStudio для отладки.

- Вкладка Timeline



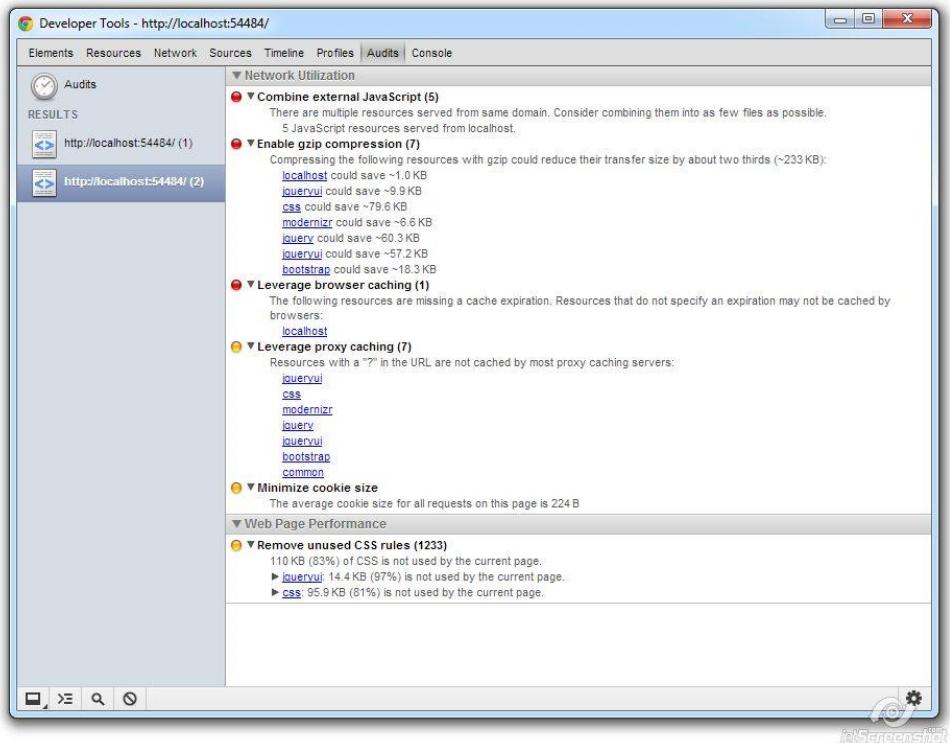
Показывает происходящие события в браузере. Я ее никогда не использовал.

- Вкладка Profiles



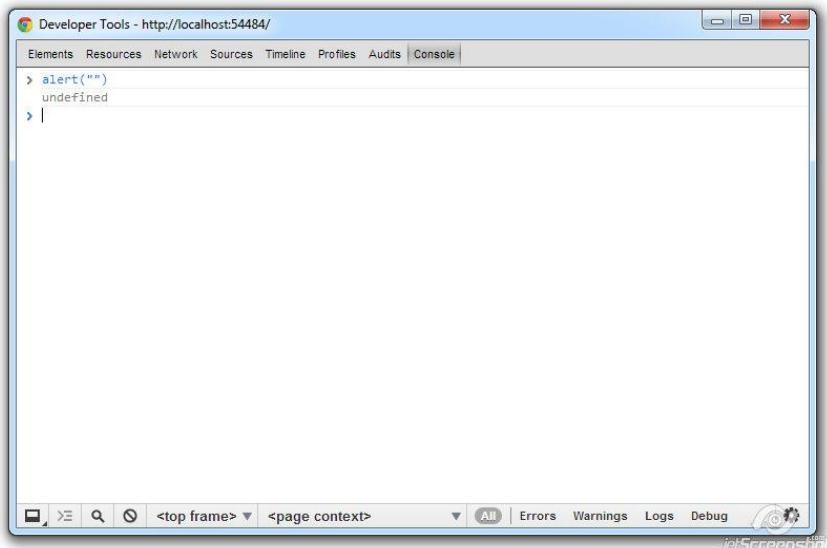
Для профилирования, т.е. нахождения нетривиальных ошибок. Также, никогда не использовал.

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

В основе всего лежит 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 ближайший элемент, который соответствует селектору, причем сам элемент может быть ним же.

## 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.
  - .unload() – вызывается, когда пользователь хочет закрыть 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 отпусканье 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 соответствующие 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, это 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 ею).

```
$.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 некоторых случаях.

Рассмотрим подробнее параметры:

- **type**. GET или POST запрос.
- **url**. Если его не задать, или будет по умолчанию, то аякс-запрос будет отправлен на текущую страницу
- **data**. В формате json или get - подобная строка, типа «id=1&value=2». Можно использовать сериализацию формы:  
`data: $("form").serialize()`  
При этом надо помнить, что при передаче множества одинаковых значений чекбоксов нужно устанавливать параметр  
`traditional : true`
- **beforeSend**. Событие, генерирующееся 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, вместо нее выскакивает попапоконшко 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 следующую функциональность добавляем тоже в common.js:

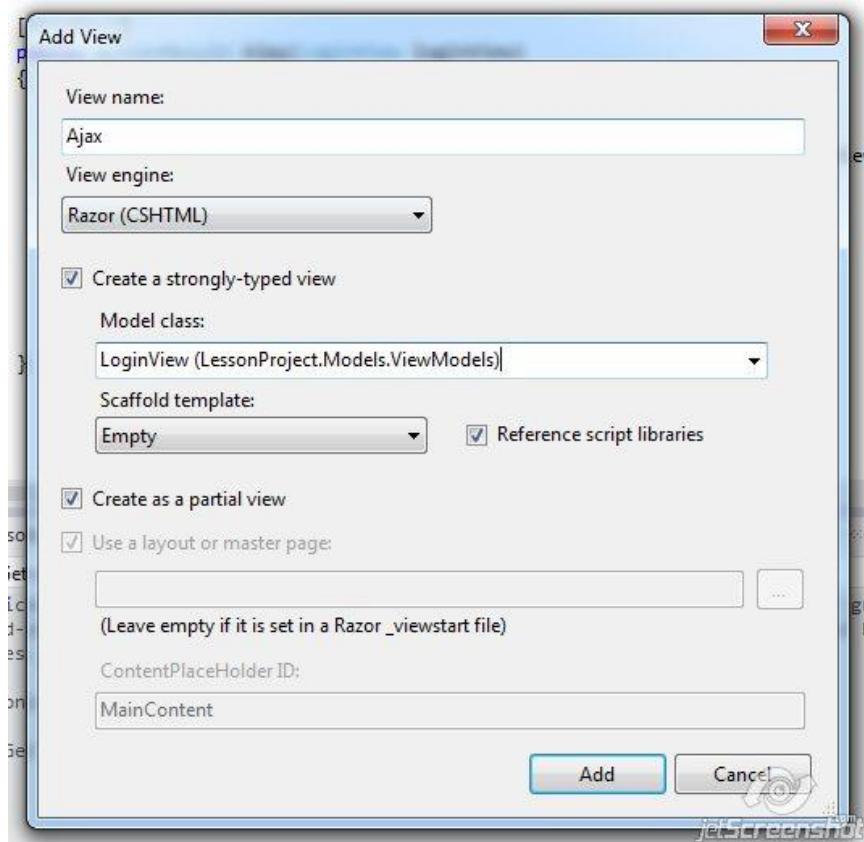
```
this.init = function () {
 $("#LoginPopup").click(function () {
 this.showPopup("/Login/Ajax", function (modal)
 {
 });
 });
}
```

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-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" })
 </div>
 <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" })
 </div>
 @Html.ValidationMessage("Password")
 </div>
 }
 </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):

```
Вход
```

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 - 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, но это лишь малая толика того, что вообще можно знать о верстке, стилях и программировании в клиентской части.

Мы научились пользоваться отладчиком в Chrome и создавать аjax 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, рассмотрим 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/

приступим к изучению.

### Razor

При создании View есть выбор между двумя движками: ASPX и Razor. Первый мы не будем использовать в дальнейшем, поэтому поговорим о Razor.



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

Razor использует конструкцию @Model.Name. Т.е. 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">

 @role.ID

 @role.Name

 @role.Code

 </div>
}
```

Внутри {} находятся теги – это маркер того, 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" : ""))>

 @role.ID

 @role.Name

 @role.Code

 </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 получить в параметрах значение страницы, которую мы будем выводить
2. По умолчанию это будет первая страница
3. При выводе, мы должны знать:
  - a. Список элементов БД, которые выводим
  - b. Количество страниц
  - c. 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 количество 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">

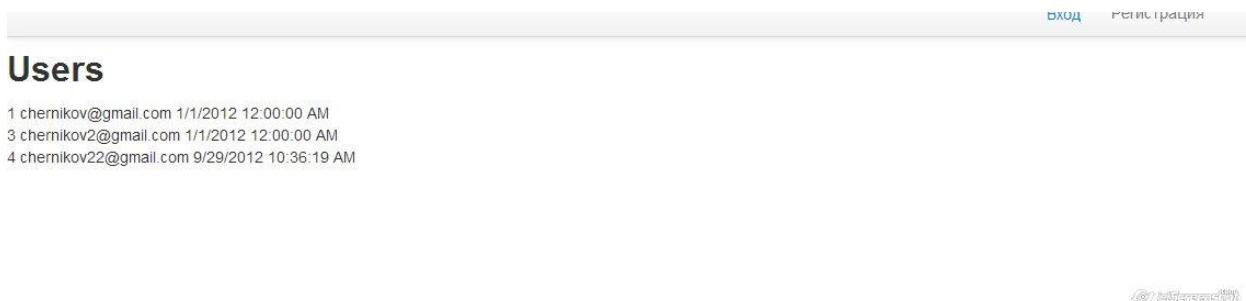
 @user.ID

 @user.Email

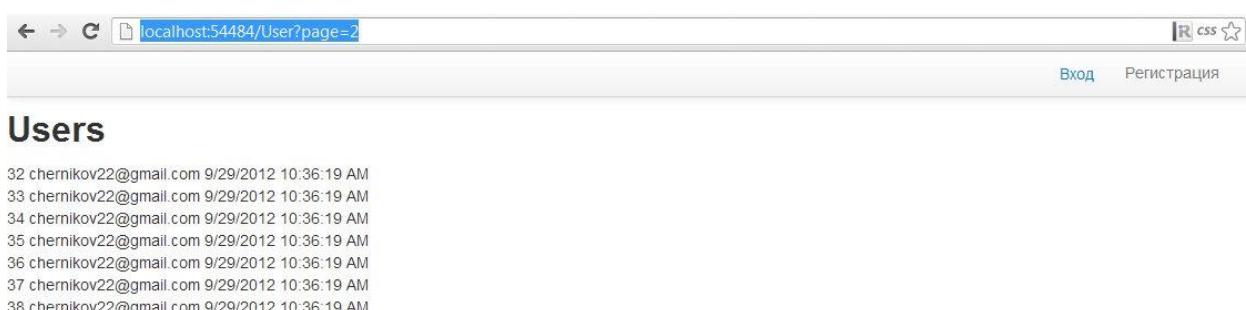
 @user.AddedDate

 </div>
 }
</p>
```

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



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



Перейдем к созданию Helper'а пагинатора, который даст нам возможность пролистывать этот список

## Helper (PagerHelper)

Так как мы используем bootstrap, то и на базе него будем делать пагинатор. В коде он выглядит так:

```
<div class="pagination">

 Prev
 1
 2
 3
 4
 5
 Next

</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 = "«";
 if (currentPage == 1)
 {
 prevBuilder.MergeAttribute("href", "#");
 builder.AppendLine("<li class=\"active>" + prevBuilder.ToString() +
 "");
 }
 else
 {
 prevBuilder.MergeAttribute("href", pageUrl.Invoke(currentPage - 1));
 builder.AppendLine("" + prevBuilder.ToString() + "");
 }
 //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() + "");
 }
 else
 {
 subBuilder.MergeAttribute("href", pageUrl.Invoke(i));
 builder.AppendLine("" + subBuilder.ToString() + "");
 }
 }
 else if ((i == 4) && (currentPage > 5))
 {
 //Трееточие первое
 builder.AppendLine("<li class=\"disabled\"> ...
");
 }
 else if ((i == (totalPages - 3)) && (currentPage < (totalPages - 4)))
 {
 //Трееточие второе
 builder.AppendLine("<li class=\"disabled\"> ...
");
 }
 //Next
 var nextBuilder = new TagBuilder("a");
 nextBuilder.InnerHtml = "»";
 if (currentPage == totalPages)
 {
 nextBuilder.MergeAttribute("href", "#");
 builder.AppendLine("<li class=\"active\">" + nextBuilder.ToString() +
"");
 }
 else
 {
 nextBuilder.MergeAttribute("href", pageUrl.Invoke(currentPage + 1));
 builder.AppendLine("" + nextBuilder.ToString() + "");
 }
 return new MvcHtmlString("" + builder.ToString() + "");
}

```

Добавим namespace LessonProject.Helper в объявления 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. А Url.Action() – формирует ссылку на страницу /User/Index с параметром page = x.

Вот что получилось (уменьшил количество вывода на странице до 5, чтобы образовалось больше страниц):

The screenshot shows a web page titled "Users". The page lists several user entries, each consisting of a name and a timestamp. Below the list is a navigation menu with links labeled from « to ».

«	1	2	3	4	...	6	7	8	»
---	---	---	---	---	-----	---	---	---	---

© jetScreenshot.com

## 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}))
```

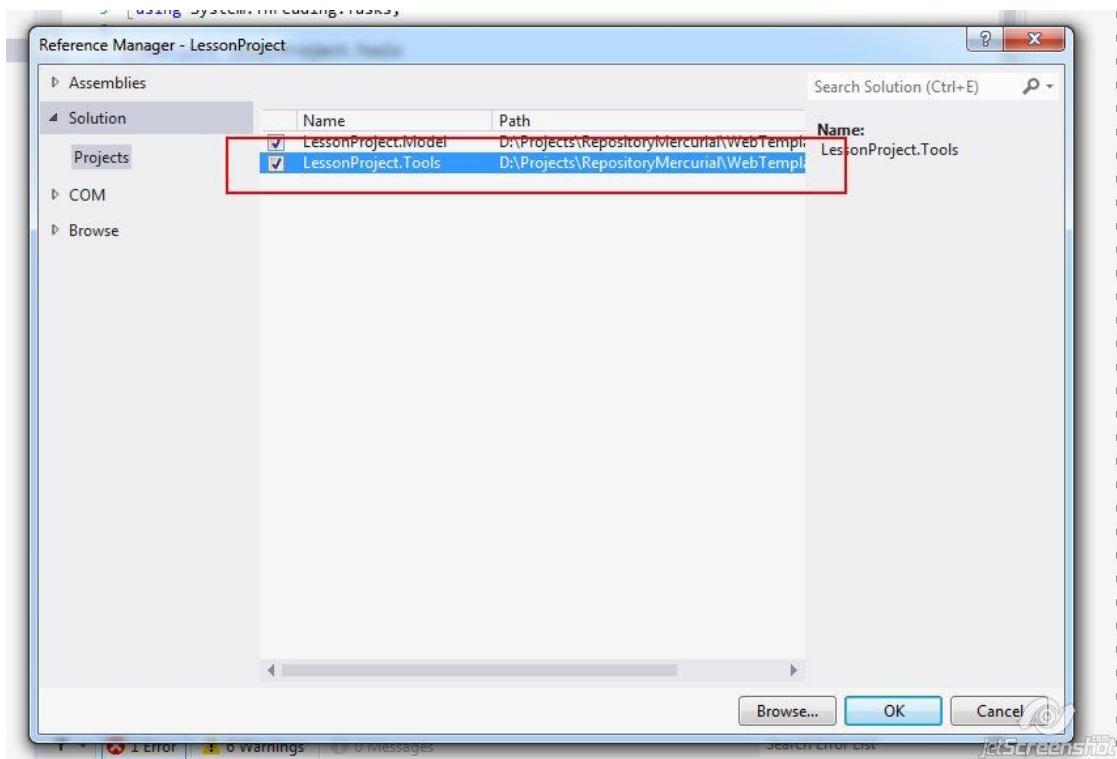
## Extension

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 NIToBr(). Создаем:

```
public static MvcHtmlString NIToBr(this string source)
{
 if (string.IsNullOrWhiteSpace(source))
 {
 return new MvcHtmlString(string.Empty);
 }
 return new MvcHtmlString(source.Replace(Environment.NewLine, "
"));
}
```

Добавляем объявление в Web.config:

```
<add namespace="LessonProject.Tools" />
```

Можно использовать:

```
@Model.Description.NIToBr()
```

Так же создадим расширение Teaser, которое урезает строку до максимального допустимого 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](#) на эту тему. Но то был asp.net mvc первый, а сейчас можно сделать все на много проще.

Значит, проблема заключается в следующем. Мы на post-action в контроллере принимаем объект, у которого, заранее не известно количество 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, поэтому здесь 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">
 Ok</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 состоит из обработки кнопок 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;
$(document).ready(function () {
 customerEdit = new CustomerEdit();
 customerEdit.init();
});

```

При нажатии на кнопку «добавить», мы получаем по аjax-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()));
}

```

## Edit

Name

Добавить Удалить

Имя

Цена  The value 'sdfas' is not valid for Price.

Удалить

Имя

Цена  0

Удалить

Имя

Цена  0

Удалить

Имя

Цена  0

Ok



### Правила перенаправления

Есть две ситуации, которые похожи, но по-разному обрабатываются. Речь идет о 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 имеем прав на это, нас переправят 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 при обработке 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.

Эта ситуация может возникнуть, когда проверка роли выполнена, но пользователь берется за управление чужого ресурса, при обращении к которому, естественно, в доступе ему должно быть отказано.

Или второй, и более частый случай, когда побитой 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 свойства RedirectToNotFoundPage и RedirectToLoginPage UserController (/Areas/Default/Controllers/UserController.cs) так:

```
[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, изначально необходимо быть авторизованным в системе, но 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-нибудь 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

 Вернуться на главную
 </div>
</body>
```

```
</html>
```

Аналогичная страница 404 (/Areas/Default.Views/Error/NotFoundPage.cshtml).

## Другие ActionResult

ActionResult не обязательно может возвращать View. Рассмотрим другие примеры:

- Content(result) – возвращает простой текст в http-ответе
- JsonResult(object) – возвращает объект в формате json (далее рассмотрим)
- FileContentResult() – возвращает ресурс для скачивания Отправляет в ответ содержимое файла. (источник - мсдн)
- 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 описания лент новостей, статей, блогов. У нас пока нет постов, так что мы просто создадим RssActionResult. В System.ServiceModel.Syndication – это набор инструментов для работы с RSS. Нам необходимы SyndicationFeed и SyndicationItem.

Подключим System.ServiceModel в reference, 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);
 }
}
}

```

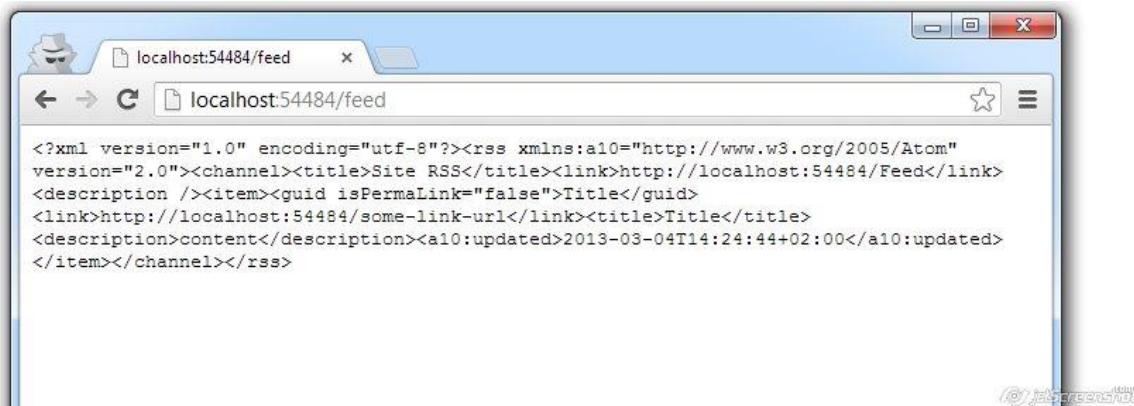
Основным параметром является Feed, куда передается уже сформированный Feed. Сам ActionResult вызывает ExecuteResult(). Мы используем Rss20FeedFormatted для записи xml в Response.Output. Response.Output – это stream, куда записывая данные обрабатываются браузером как ответ. Для того чтобы браузер понимал, 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 \(by Google\)](#), то наш RSS-канал отобразится в браузере, и мы сможем на него подписаться.:



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. Научиться 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 в нем хранятся настройки программы. Рассмотрим подробнее, из чего состоит этот файл:

- **configSection**. Это секция отвечает за то, какие классы будут обрабатывать далее объявленные секции. Состоит из атрибута name - это тег, далее объявленной секции, и type – к какому классу относится.
- **connectionStrings**. Это секция отвечает за работу с указанием строк инициализаций соединений с базами данных. (собраться подумать сходить купить выпить ;))
- **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, объявленными в configSection.

### IConfig (и реализация).

Аналогично 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; }
```

Теперь сделаем в инициализации контроллера переопределение 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:

## LessonProject

5 марта 2013 г.

Роли Пользователи



И 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

В этой части мы рассмотрим создание своих собственных ConfigSection. В этой главе мы реализуем загрузку файлов и создание превью. Нам понадобятся следующие данные: во-первых, зависимость mime-type от расширения, и иконка файлов (для скачивания, например):

- расширение
- mime-type
- большая иконка
- маленькая иконка

и во-вторых, данные для создания превью:

- наименование превью (например, UserAvatarSize)
- ширина
- высота

Оба типа делаются одинаково, так что я распишу только создание одного из них. Пусть 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 ключом (как первичный 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 понадобятся:

- SmtpServer. Имя сервера.
- SmtpPort. Порт, обычно 25й.
- SmtpUserName. Логин.
- SmtpPassword. Пароль.
- SmtpReply. Обратный адрес в строке Reply-to.
- SmtpUser. Имя пользователя в строке From.
- EnableSsl. Да/нет, использовать ли работу 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 все это теперь в IConfig.cs и Config.cs (/Global/Config/IConfig.cs):

```

public interface IConfig
{
 string Lang { get; }

 IQueryable<IconSize> IconSizes { get; }

 IQueryable<MimeType> MimeType { 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> MimeType
{
 get
 {
 MimeTypeConfigSection configInfo =
(MimeTypeConfigSection)ConfigurationManager.GetSection("mimeConfig");
 return configInfo.MimeType.OfType<MimeType>().AsQueryable<MimeType>();
 }
}

public MailSetting MailSetting
{
 get
 {
 return (MailSetting)ConfigurationManager.GetSection("mailConfig");
 }
}
```

Мы еще добавим MailTemplates - шаблоны которые нам понадобятся для рассылки email при регистрации, или при напоминании пароля.

### Простая загрузка файлов

Сейчас рассмотрим 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";
}

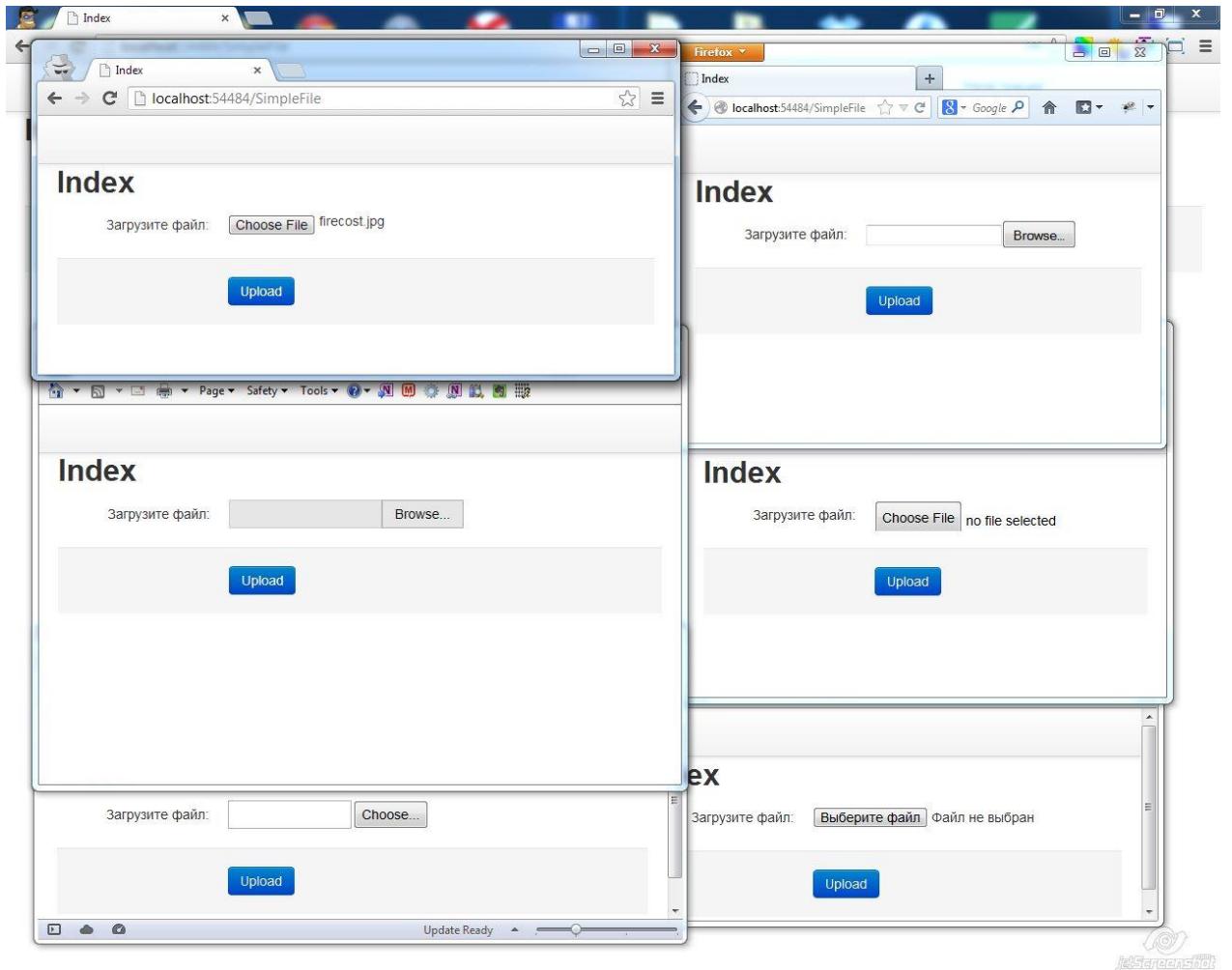
<h2>Index</h2>

@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 них есть соответствующие методы в @Html-классе). Enctype необходимо установить в “multipart/form-data”, чтобы была возможность загрузить большой объём информации.

Загружаем и проверяем. Наш файл благополучно загружен, только необходимо сохранить InputStream в некий файл. Но оставим пока так и рассмотрим недостатки.

Первый недостаток – это то, что во всех браузерах форма выбора файла выглядит по-разному:



Конечно, ведь дизайнер представляет себе, что загрузка файлов выполняется как в Safari, а заказчик проверяет в Chrome и IE, и начинает спрашивать у разработчиков: «Что за самодеятельность?»

Второй недостаток –если форма не прошла валидацию, то эти поля необходимо выбрать заново. Т.е. есть такая форма:

- Имя
- Фамилия
- Электронная почта
- Дата рождения
- Фотография
- Фотография первого разворота паспорта
- Фотография второго разворота паспорта
- Фотография паспорта с пропиской
- Пароль
- Пароль еще раз
- Капча

И вдруг вы набрали пароль неверно, или капчу не так ввели, или фотография второго разворота паспорта слишком большая, или вы забыли перегнать из raw-формата в jpeg.

В итоге фотографии, прописку и капчу надо вводить заново. Естественно, это совсем не user friendly, и раздражает заказчика (к тому же дизайнер нарисовал красиво, а выглядит убого).

## Загрузка файла (ов) с помощью Ajax

Определим как должна вести себя загрузка файла:

- Пользователь кликает на «загрузить».
- Открывается форма выбора файла
- Пользователь выбирает файл
- Файл загружается, или выдается ошибка о том, что что-то не так
- Если даже форма и не проходит валидацию, то файл остается загруженным и его не нужно загружать заново.

Это называется аяч-загрузка и для нее 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 изображения 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 ниже рассмотрю, почему так. А сейчас создадим View для Index. Для этого:

- 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
- Верификация формы и запись в БД не нужны.

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")
}

Index

 <div class="control-group">
 <label class="control-label" for="Text">
 Image
 </label>
 <div class="controls">
 <div id="UploadImage">
 Upload
 </div>
 </div>
 </div>
 <div>

 </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;

 $(document).ready(function () {
 fileIndex = new FileIndex();
 fileIndex.init();
 });
}
```

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 заранее создали) с помощью Server.MapPath.
- Далее сохраняем файл с помощью SaveAs.
- Возвращаем имя файла в json data.filePath.

Проверяем, всё ли 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 есть много функций для создания превью, так что я перечислю разные варианты создания изображения и подробно разберу один из них. Стоит учесть, что все превью создаются в формате jpg. 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 ориентировано 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 ориентировано 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.

создадим 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 создастся превью.

Теперь сделаем обработку в 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);
});
```

теперь наш файл загружается вместе с превью. Путь большого файла также можно передавать отдельно, и записывать, например, в hidden поле и сохранять в БД как строку.

Что плохого в такой конструкции, так это две следующие проблемы:

- файлы могут быть перезаписаны, но это решается тем, что можно брать только расширение, а имя файлу присваивать отдельно, или [добавлять немного соли](#)
- 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

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 по ссылке сохраняем этот файл.

Это делается так:

```
var webClient = new WebClient();
var bytes = webClient.DownloadData(url);
var ms = new MemoryStream(bytes);
```

Где url – путь к файлу. Можно сложнее, 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.freelance.ru/users/chernikov/upload/sm\\_f\\_81850beffd0d0c89.jpg](http://localhost/File/Export?uri=https://st.freelance.ru/users/chernikov/upload/sm_f_81850beffd0d0c89.jpg)

Файл загрузился и обработан. Всё супер!

Рекомендую пройтись дебаггером по работе PreviewCreator, чтобы понять, как там всё устроено.

## Урок А. Уведомление и рассылка

Цель: Разобраться в отправлении писем и подтверждающих смс. MailNotify, использование конфигурационного файла. Рассылка через создание отдельного потока.

### SmtpClient и MailNotify

При разработке сайта мы рано или поздно сталкиваемся с взаимодействием с электронной почтой, будь то активация пользователя, напоминание или сброс пароль, или создание рассылки .

Определимся, что нам для этого нужно:

- Класс, который будет рассылать письма
- Конфигурация smtp берется из IConfig
- Ошибки отправки письма протоколируются
- Наличие параметра, выключающего работу почты, дабы при работе с боевой базой клиентов не разослать какой-то треш.

Создадим статический класс, назовем его MailSender (/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, статически инициализируется IConfig из DependencyResolver
  - Если установлен флаг EnableMain, то начинаем работу с почтой, иначе просто письмо запишем в лог-файл
  - Если MailAddress не указан, то он инициализируется по данным из конфига
  - SmtpClient инициализируется по данным из конфига
  - Тело письма – html
  - Кодировка – UTF8
  - Если при отправке произошла ошибка, то запишем Exception.Message в лог (тут можно и больше информации собирать, но пока нет необходимости).

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 = new Configuration();
 }
 return _config;
 }
 }
}
```

```

 _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));
 }
}

```

Аналогично получаем конфиг. При рассылке мы указываем для неё , и дальше используем Func<string,string> для формирования темы и тела письма.

Уведомим пользователя о регистрации, используя шаблон Register из Web.config:

```

<add name="Register" subject="Регистрация на {0}" template="Здравствуйте!

 Перейдите по ссылке http://{1}/User/Activate/{0}, чтобы
подтвердить свой почтовый ящик.

С уважением, команда {1}" />
```

Заметим, как необходимо экранировать html-теги, чтобы правильно сделать шаблон. 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):

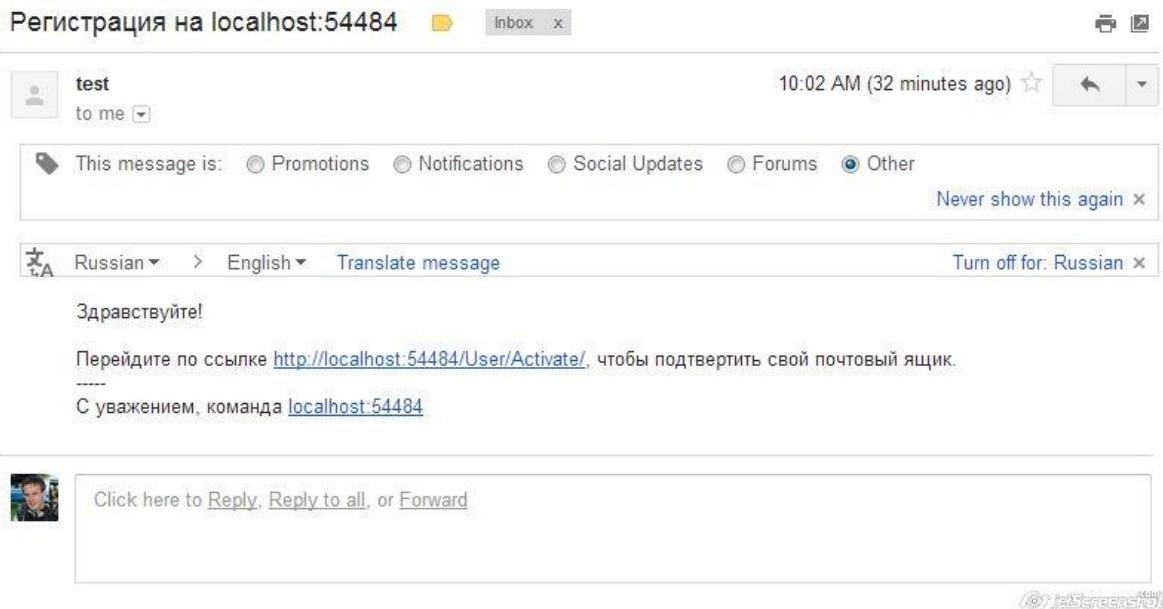
```

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 рассылку 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 получения текста 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 есть сформированная страница. Запускаем:

<http://localhost/User/SubscriptionShow>

## SmsNotify

В этой главе рассмотрим взаимодействие с помощью смс, а не только почты. Но есть нюанс – доступ к рассылке предоставляется отдельными сервисами, и тут мы рассмотрим только основные принципы написания модуля 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 deserialise 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 responseStream = 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 количеству 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
- Письма извлекаются из БД последовательно (письмо может удалиться, может только обнулить содержимое письма (чтоб сэкономить 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 В. Json

Цель урока: 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

Json – это текстовый формат данных, основанный на Javascript.

Пример данных в Json:

```
{
 "firstName": "Иван",
 "lastName": "Иванов",
 "address": {
 "streetAddress": "Московское ш., 101, кв.101",
 "city": "Ленинград",
 "postalCode": 101101
 },
 "phoneNumbers": [
 "812 123-1234",
 "916 123-4567"
]
}
```

В своей практике я работал с json с такими приложениями как yandex.maps api, facebook api, vk api, bronni api (это такой туристический портал), и даже при работе с биткоин-кошельком. Для этого используется JSON.net библиотека от <http://james.newtonking.com/pages/json-net.aspx>.

Изучим ее подробнее:

- Установим
- Изучим преобразования из json в объекты и назад
- Десериализация из сложных форматов
- Работа с facebook API (пример) - авторизация

#### Установим

```
PM> Get-Package Json.net
```

Id	Version	Description/Release Notes
--	-----	-----
Newtonsoft.Json	4.5.1	Json.NET is a popular high-performance...

```
PM> Install-Package Newtonsoft.Json
Successfully installed 'Newtonsoft.Json 4.5.11'.
Successfully added 'Newtonsoft.Json 4.5.11' to LessonProject.
```

## Документация

По этой [ссылке](#) находится документация. Мы начнем с простого преобразования объекта в json-формат и обратно. Создадим LessonProject.Console и сделаем его проектом по-умолчанию.

Добавим тип User:

```
public class User
{
 public string Id { get; set; }

 public string Name { get; set; }

 public string FirstName { get; set; }

 public string MiddleName { get; set; }

 public string LastName { get; set; }

 public string UserName { get; set; }

 public string Gender { get; set; }

 public string Email { get; set; }
}
```

Создадим объект и преобразуем в json:

```
var user = new User()
{
 Id = "404",
 Email = "chernikov@gmail.com",
 UserName = "rollinx",
 Name = "Andrey",
 FirstName = "Andrey",
 MiddleName = "Alexandrovich",
 LastName = "Chernikov",
 Gender = "M"
};

var jsonUser = JsonConvert.SerializeObject(user);
System.Console.WriteLine(jsonUser);

System.Console.ReadLine();
```

Результат:

```
{"Id": "404", "Name": "Andrey", "FirstName": "Andrey", "MiddleName": "Alexandrovich", "LastName": "Chernikov", "UserName": "rollinx", "Gender": "M", "Email": "chernikov@gmail.com"}
```

Попробуем обратное:

```
var jsonUserSource =
"{"Id": \"405\", \"Name\": \"Andrey\", \"FirstName\": \"Andrey\", \"MiddleName\": \"Alexandrovich\", \"LastName\": \"Chernikov\", \"UserName\": \"rollinx\", \"Gender\": \"M\", \"Email\": \"chernikov@gmail.com\"}";

var user2 = JsonConvert.DeserializeObject<User>(jsonUserSource);
```

И получаем результат:

```

var jsonUserSource = "{\"Id\":\"405\", \"Name\":\"Andrey\", \""
var user2 = JsonConvert.DeserializeObject<User>(jsonUserSource);

```

The screenshot shows the Visual Studio debugger's watch window. A tooltip for the 'user2' variable is displayed, showing its properties and their values. The properties listed are: Email, FirstName, Gender, Id, LastName, MiddleName, Name, and UserName. Each property has a small dropdown arrow next to it, indicating it can be expanded.

Т.е. работает в обоих направлениях. Но немного усложним. Например, зададим Gender через перечисляемый тип Male и Female, и в json должно передаваться именно Male и Female. А Id – это числовое значение:

```

public class User
{
 public enum GenderEnum
 {
 Male,
 Female
 }

 public int Id { get; set; }

 public string Name { get; set; }

 public string FirstName { get; set; }

 public string MiddleName { get; set; }

 public string LastName { get; set; }

 public string UserName { get; set; }

 public GenderEnum Gender { get; set; }

 public string Email { get; set; }
}

```

Пробуем первую часть:

```

var user = new User()
{
 Id = 404,
 Email = "chernikov@gmail.com",
 UserName = "rollinx",
 Name = "Andrey",
 FirstName = "Andrey",
 MiddleName = "Alexandrovich",
 LastName = "Chernikov",
 Gender = User.GenderEnum.Male
};

var jsonUser = JsonConvert.SerializeObject(user);

```

Результат:

```
{"Id":404,"Name":"Andrey","FirstName":"Andrey","MiddleName":"Alexandrovich","LastName":"Chernikov","UserName":"rollinx","Gender":0,"Email":"chernikov@gmail.com"}
```

Добавим:

```
[JsonConverter(typeof(StringEnumConverter))]
public enum GenderEnum
{
 Male,
 Female
}
```

Результат:

```
{"Id":404,"Name":"Andrey","FirstName":"Andrey","MiddleName":"Alexandrovich","LastName":"Chernikov","UserName":"rollinx","Gender":"Male","Email":"chernikov@gmail.com"}
```

Уже лучше. Обратно проверяем – всё ок. Изучим другие атрибуты, задающие тонкие правила настройки. Например, в json-формате будут имена в венгерской записи типа first\_name:

```
[JsonObject]
public class User
{
 [JsonConverter(typeof(StringEnumConverter))]
 public enum GenderEnum
 {
 Male,
 Female
 }

 [JsonProperty("id")]
 public int Id { get; set; }

 [JsonProperty("name")]
 public string Name { get; set; }

 [JsonProperty("first_name")]
 public string FirstName { get; set; }

 [JsonProperty("middle_name")]
 public string MiddleName { get; set; }

 [JsonProperty("last_name")]
 public string LastName { get; set; }

 [JsonProperty("user_name")]
 public string UserName { get; set; }

 [JsonProperty("gender")]
 public GenderEnum Gender { get; set; }

 [JsonProperty("email")]
 public string Email { get; set; }
}
```

Результат:

```
{"id":404,"name":"Andrey","first_name":"Andrey","middle_name":"Alexandrovich","last_name":"Chernikov","user_name":"rollinx","gender":"Male","email":"chernikov@gmail.com"}
```

Для описания списка добавим класс Photo:

```
[JsonObject]
public class Photo
{
 [JsonProperty("id")]
 public int Id { get; set; }

 [JsonProperty("name")]
 public string Name { get; set; }
}
```

И в User добавим:

```
[JsonProperty("photo_album")]
public List<Photo> PhotoAlbum { get; set; }
```

Результат:

```
{"id":404,"name":"Andrey","first_name":"Andrey","middle_name":"Alexandrovich","last_name":"Chernikov","user_name":"rollinx","gender":"Male","email":"chernikov@gmail.com","photo_album":[{"id":1,"name":"Я с инстаграммом"}, {"id":2,"name":"Я на фоне заниженного таза"}]}
```

Всё просто и предсказуемо.

Разберем сложный случай, например, когда для Gender нам надо описывать не Male\Female, а M\F. Для этого создаем класс по разбору GenderEnumConverter:

```
public class GenderEnumConverter : JsonConverter
{
 public override object ReadJson(JsonReader reader,
 Type objectType,
 object existingValue,
 JsonSerializer serializer)
 {
 var value = reader.Value.ToString();
 if (string.Compare(value, "M", true) == 0)
 {
 return User.GenderEnum.Male;
 }
 if (string.Compare(value, "F", true) == 0)
 {
 return User.GenderEnum.Female;
 }
 return User.GenderEnum.Male;
 }

 public override void WriteJson(JsonWriter writer, object value, JsonSerializer
serializer)
 {
 var obj = (User.GenderEnum)value;

 // Write associative array field name
 writer.WriteValue(value.ToString().Substring(0,1));
 }

 public override bool CanConvert(Type objectType)
 {
```

```
 return false;
 }
}
```

И устанавливаем этот конвертер для обработки

```
[JsonConverter(typeof(GenderEnumConverter))]
public enum GenderEnum
{
 Male,
 Female
}
```

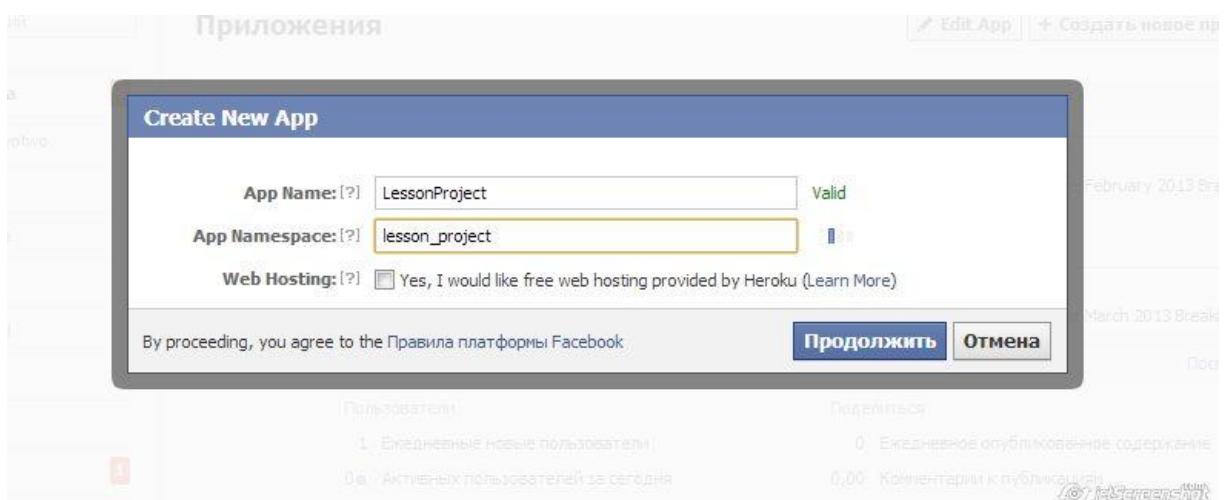
Вообще, конвертеры могут быть бесконечно сложными для разбора json-данных. Можно обрабатывать нетипичные записи дат, сложные структуры данных, формировать сериализацию своих классов.

## Работа с facebook

Всё это понятно и скучно, так что метнемся работать с facebook API. Развлечений всем! Для начала заведем проект LessonProject.FacebookAPI. Добавим туда Json.NET и свяжем с основным проектом.

На facebook надо завести ApplicationID по адресу:

<https://developers.facebook.com/apps>



Создаем, получаем:

The screenshot shows the Facebook App Dashboard for an application named 'LessonProject'. At the top, it displays the App ID (136398216534301) and App Secret (e6de78fd40596f00e225dce861b34a1a). Below this, there's a section titled 'Основная информация' (Basic Information) containing fields for Display Name (LessonProject), Namespace (lesson\_project), and Email (chernikov@gmail.com). It also includes sections for App Domains (Enter your site domains and press enter), Hosting URL (You have not generated a URL through one of our partners (Get one)), and Sandbox Mode (Enabled). A large section titled 'Как ваше приложение встроено в Facebook?' (How your app is integrated into Facebook?) lists various deployment options, all of which are checked: Website with Facebook Login, Приложение на Facebook, Мобильная версия, Приложение для iOS, Приложение для Android, and Вкладка страницы. At the bottom right of the dashboard is a watermark for 'jetScreenshot'.

Нам интересно будет AppID и AppSecret (я его потом сброшу, так что оно будет другим).

Добавляем эти данные в Config/FacebookSetting.cs (уже знаем, как это делается):

```
public class FacebookSetting : ConfigurationSection
{
 [ConfigurationProperty("AppID", IsRequired = true)]
 public string AppID
 {
 get
 {
 return this["AppID"] as string;
 }
 set
 {
 this["AppID"] = value;
 }
 }

 [ConfigurationProperty("AppSecret", IsRequired = true)]
 public string AppSecret
 {
 get
 {
 return this["AppSecret"] as string;
 }
 set
 {
 this["AppSecret"] = value;
 }
 }
}
```

Общение с фейсбуком происходит так:

- Попросим пользователя авторизоваться, так мы узнаем, какие права у нас есть
- Ответом этого будет токен доступа, по нему мы будем получать информацию
- Получаем информацию про самого пользователя

Создадим интерфейс, который будет реализовывать наш FacebookSetting (чтобы была обратная совместимость) (LessonProject.FacebookAPI/IFbAppConfig.cs):

```
public interface IFbAppConfig
{
 string AppID { get; }

 string AppSecret { get; }
}
```

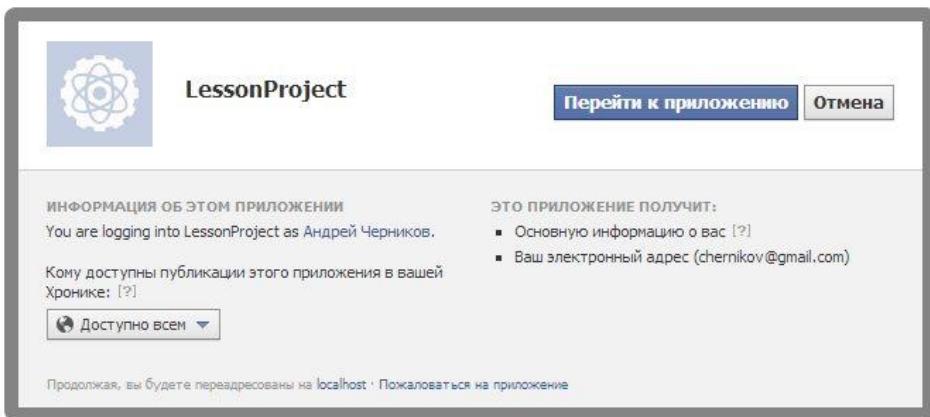
Добавляем в FacebookSetting (/Global/Config/FacebookSetting.cs):

```
public class FacebookSetting : ConfigurationSection, IFbAppConfig
```

Используя наш AppID, мы идем по строке типа:

[https://www.facebook.com/dialog/oauth?client\\_id=136398216534301&redirect\\_uri=http%3A%2F%2Flocalhost%3A54484%2FFacebook%2FToken&scope=email](https://www.facebook.com/dialog/oauth?client_id=136398216534301&redirect_uri=http%3A%2F%2Flocalhost%3A54484%2FFacebook%2FToken&scope=email)

И это выглядит так:



JetBrains

Если мы нажимаем «Перейти к приложению» - то нас переправляют на страницу <http://localhost:54484/Facebook/Token> с параметром code, по которому мы можем получить токен (который действует некоторое время). Выполняем такой запрос:

```
https://graph.facebook.com/oauth/access_token?client_id=136398216534301&redirect_uri=http://localhost:54484/Facebook/Token&client_secret=e6de78fd40596f00e225dce861b34a1a&code=AQAScKUYKGpzwijZ3Y3SHjN0d4Q5nsyrYPdJaPhX-88rwBOuMrdrimL8h82bGv3HAh7TL6oJyZ0gNgiB8BcCeH8G_Zj7h6hlft_BFbOfIJZJB9nKW6Q4iR3a0VVImxM0QYJas3eVg4qtYNkqUcWbgXDSK2JENcuomUX38haxFUFdKXrVjL1acNZSocESsx6nfx_FyF_QlbwnUO5cwogrLp
```

На что получаем ответ:

```
access_token=AAAB8Da8ZBmR0BAMCOx5293ZArYvFu5oRkmZCrwZAbvpWZB3ZCLBeiooslyYPZBVwHjxSp
e3KzJ4VLFPIxwwf0D6TIEiM5ApzU8EMoDpOxE4uAZDZD&expires=5183977
```

Нам нужен этот access\_token, сохраняем его, и с помощью него мы запрашиваем данные по ссылке:

[https://graph.facebook.com/me?access\\_token=AAAB8Da8ZBmR0BAImiTO9QwuUXbgHPLZBQWmAyZB\\_UkjR2A37aVN4vaqaFmt6h1ZBvurUpvN95EXddy5d6J1ldZA2jWTxSd3eZBHlYMzKwdxgZDZD](https://graph.facebook.com/me?access_token=AAAB8Da8ZBmR0BAImiTO9QwuUXbgHPLZBQWmAyZB_UkjR2A37aVN4vaqaFmt6h1ZBvurUpvN95EXddy5d6J1ldZA2jWTxSd3eZBHlYMzKwdxgZDZD)

На что он нам отвечает:

```
{"id":"708770020","name":"Andrey Chernikov","first_name":"Andrey","last_name":"Chernikov","link":"http://www.facebook.com/chernikov1","username":"chernikov1","gender":"male","email":"chernikov\u0040gmail.com","timezone":2,"locale":"ru_RU","verified":true,"updated_time":"2013-03-06T15:01:28+0000"}
```

И вот это мы приведем к классу FbUserInfo (LessonProject.FacebookAPI/FbUserInfo.cs):

```
[JsonObject]
public class FbUserInfo
{
 [JsonProperty("id")]
 public string Id { get; set; }

 [JsonProperty("name")]
 public string Name { get; set; }

 [JsonProperty("first_name")]
 public string FirstName { get; set; }

 [JsonProperty("last_name")]
 public string LastName { get; set; }

 [JsonProperty("link")]
 public string Link { get; set; }

 [JsonProperty("username")]
 public string UserName { get; set; }

 [JsonProperty("gender")]
 public string Gender { get; set; }

 [JsonProperty("email")]
 public string Email { get; set; }

 [JsonProperty("locale")]
 public string Locale { get; set; }

 [JsonProperty("timezone")]
 public double? Timezone { get; set; }

 [JsonProperty("verified")]
 public bool? Verified { get; set; }

 [JsonProperty("updated_time")]
 public DateTime? updatedTime { get; set; }
}
```

Всю описанную выше работу заключаем в FbProvider.cs (LessonProject.FacebookAPI/Provider.cs):

```
public class FbProvider
{
 private static string AuthorizeUri =
"https://graph.facebook.com/oauth/authorize?client_id={0}&redirect_uri={1}&scope=email";
 private static string GetAccessTokenUri =
"https://graph.facebook.com/oauth/access_token?client_id={0}&redirect_uri={1}&client_secr
et={2}&code={3}";
 private static string GetUserInfoUri =
"https://graph.facebook.com/me?access_token={0}";

 private static string GraphUri = "https://graph.facebook.com/{0}";

 public IFbAppConfig Config { get; set; }

 public string AccessToken { get; set; }

 public string Authorize(string redirectTo)
 {
 return string.Format(AuthorizeUri, Config.AppID, redirectTo);
 }

 public bool GetAccessToken(string code, string redirectTo)
 {
 var request = string.Format(GetAccessTokenUri, Config.AppID, redirectTo,
Config.AppSecret, code);
 WebClient webClient = new WebClient();
 string response = webClient.DownloadString(request);
 try
 {
 var pairResponse = response.Split('&');
 AccessToken = pairResponse[0].Split('=')[1];
 return true;
 }
 catch (Exception ex)
 {
 return false;
 }
 }

 public JObject GetUserInfo()
 {
 var request = string.Format(GetUserInfoUri, AccessToken);
 WebClient webClient = new WebClient();

 string response = webClient.DownloadString(request);
 return JObject.Parse(response);
 }
}
```

Где

- **Authorize** – это формирование ссылки для запроса прав.
- **GetAccessToken** – это запрос по получению временного токена.
- **GetUserInfo** – это запрос по получению данных пользователя.

Обратите внимание, как мы используем WebClient.DownloadString – и оп-па, мы получили нужные данные из недр интернета.

Едем дальше. Создадим контроллер FacebookController (/Areas/Default/Controllers/FacebookController.cs):

```
public class FacebookController : DefaultController
{
 private FbProvider fbProvider;

 protected override void Initialize(System.Web.Routing.RequestContext
requestContext)
 {
 fbProvider = new FbProvider();
 fbProvider.Config = Config.FacebookSetting;
 base.Initialize(requestContext);
 }

 public ActionResult Index()
 {
 return Redirect(fbProvider.Authorize("http://" + HostName +
"/Facebook/Token"));
 }

 public ActionResult Token()
 {
 if (Request.Params.AllKeys.Contains("code"))
 {
 var code = Request.Params["code"];
 if (fbProvider.GetAccessToken(code, "http://" + HostName +
"/Facebook/Token"))
 {
 var jObj = fbProvider.GetUserinfo();
 var fbUserInfo =
JsonConvert.DeserializeObject<FbUserInfo>(jObj.ToString());

 return View(fbUserInfo);
 }
 }
 return View("CantInitialize");
 }
}
```

В Initialize передаем в FbProvider AppID и AppSecret. После захода – делаем редирект на facebook с запросом прав у пользователя (окошко разрешения). Если пользователь уже нам это когда-то разрешил, чтобы не спрашивать по 100 раз, facebook нас переправит на страницу /Facebook/Token. Если код для получения токена не удалось получить – возвращаем View CantInitialize (/Areas/Default/Views/Facebook/CantInitialize.cshtml):

```
@{
 ViewBag.Title = "CantInitialize";
 Layout = "~/Areas/Default/Views/Shared/_Layout.cshtml";
}

<h2>CantInitialize</h2>

<h3> Ну нет - так нет</h3>
```

Иначе, когда всё хорошо, то получаем наш токен (он сохраняется в fbProvider) и запрашиваем данные о пользователе. Получив – преобразовываем в объект класса FbUserInfo и выводим во View (/Areas/Default/Views/Facebook/Token.cshtml):

```

@model LessonProject.FacebookAPI.FbUserInfo

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

<h2>Данные</h2>
<p> Вот что я про тебя знаю:</p>

<dl class="dl-horizontal">
 <dt>ID</dt>
 <dd>@Model.Id</dd>

 <dt>FirstName</dt>
 <dd>@Model.FirstName</dd>
 <dt>LastName</dt>
 <dd>@Model.LastName</dd>
 <dt>Link</dt>
 <dd>@Model.Link</dd>
</dl>

```

### Клиентский код/Серверный код (Access-Control-Allow-Origin)

Рассмотрим еще ситуацию, когда всё это взаимодействие заключено в js-файлах, мы выполняем только аjax-запросы. Изменим код метода Token. Получаем данные пользователя не серверным кодом от facebook, а передаем во View токен  
(/Areas/Default/Controllers/FacebookController.cs:Token):

```

public ActionResult Token()
{
 if (Request.Params.AllKeys.Contains("code"))
 {
 var code = Request.Params["code"];
 if (fbProvider.GetAccessToken(code, "http://" + HostName +
"/Facebook/Token"))
 {

 /* var jObj = fbProvider.GetUserInfo();
 var fbUserInfo =
JsonConvert.DeserializeObject<FbUserInfo>(jObj.ToString());
 */
 ViewBag.Token = fbProvider.AccessToken;
 return View();
 }
 }
 return View("CantInitialize");
}

```

Изменим Token.cshtml (/Areas/Default/Views/Facebook/Token.cshtml):

```

 @{
 ViewBag.Title = "Token";
 Layout = "~/Areas/Default/Views/Shared/_Layout.cshtml";
}
@section scripts {
 @Scripts.Render("~/Scripts/default/facebook-token.js")
}

@Html.Hidden("Token", ViewBag.Token as string)
<h2>Данные</h2>
<p> Вот что я про тебя знаю:</p>

```

```

<dl class="dl-horizontal">
 <dt>ID</dt>
 <dd id="ID"></dd>

 <dt>FirstName</dt>
 <dd id="FirstName"></dd>
 <dt>LastName</dt>
 <dd id="LastName"></dd>
 <dt>Link</dt>
 <dd id="Link"></dd>
</dl>

```

Добавляем facebook-token.js (/Scripts/default/facebook-token.js):

```

function FacebookToken() {
 _this = this;

 this.ajaxGetUserInfo = "https://graph.facebook.com/me?access_token=";

 this.init = function () {
 var token = $("#Token").val();
 $.ajax({
 type: "GET",
 dataType: 'json',
 url: _this.ajaxGetUserInfo + token,
 success: function (data)
 {
 $("#ID").text(data.id);
 $("#FirstName").text(data.first_name);
 $("#LastName").text(data.last_name);
 $("#Link").text(data.link);
 }
 })
 }

 var facebookToken = null;

 $(().ready(function () {
 facebookToken = new FacebookToken();
 facebookToken.init();
 }));
}

```

Запускаем, проверяем. Всё отлично. Но обратим внимание на такой параметр в http-ответе:

The screenshot shows the Network tab of the Google Chrome Developer Tools. A request to `graph.facebook.com/me?access_token=AAAB8Da8ZBmR0BAImiT09QwuUXbgHPLZBQwMAYZBUkjR2A37aVN4vaqaFmt6h1ZBvurUpvN95Exddy5d6J1ldZA2jWTxSd3eZBH1YmzKwdxgZDZ` is selected. The Headers section shows various HTTP headers. The Response Headers section is highlighted with a red box and contains the following entries:

- access-control-allow-origin: \*
- cache-control: private, no-cache, no-store, must-revalidate
- content-length: 358
- content-type: text/javascript; charset=UTF-8
- date: Wed, 06 Mar 2013 15:58:58 GMT
- etag: "acc50b804548327cd745ed026a3770cb53027692"
- expires: Sat, 01 Jan 2000 00:00:00 GMT
- last-modified: 2013-03-06T15:01:28+0000
- pragma: no-cache
- status: 200
- version: HTTP/1.1
- x-fb-debug: +q50LqlxzSG56od3plcBli+XnqUv2fY7PUCIrm/N66g=
- x-fb-rev: 749575

Access-control-allow-origin – это параметр, который, будучи установлен, позволяет делать ajax-запросы из браузера к сайту, размещенному на другом домене.

Т.е. если мы обращаемся по `$.ajax()` из браузера и в ответе этого заголовка нет, то выдается ошибка:

Origin http://localhost:8080 is not allowed by Access-Control-Allow-Origin

Для этого создадим атрибут, который будет добавлять этот заголовок, если мы захотим организовать обращение к нашему сайту с других сайтов (`/Attribute/AllowCrossSiteJson.cs`):

```
public class AllowCrossSiteJsonAttribute : ActionFilterAttribute
{
 public override void OnActionExecuting(ActionExecutingContext filterContext)
 {
 filterContext.RequestContext.HttpContext.Response.AddHeader("Access-Control-Allow-Origin", "*");
 base.OnActionExecuting(filterContext);
 }
}
```

Добавим использование. Например, метод-action OK, который всегда будет возвращать { "result" : "OK" } (/Areas/Default/Controllers/HomeController.cs):

```
[AllowCrossSiteJson]
public ActionResult OK()
{
 return Json(new { result = "OK" }, JsonRequestBehavior.AllowGet);
}
```

На этом всё по Json и работе с facebook. Можете потренироваться и поработать с авторизацией и взаимодействием с vk api. Документация тут: <http://vk.com/developers.php>.

## Урок С. Многоязычный сайт

Цель: Научиться создавать многоязычные сайты. Структура БД. Ресурсы сайта. Определение языка. Переключение между языками. Работа в админке.

### Проблемы многоязычного сайта

Итак, заказчик просит сделать сайт многоязычным, т.е. чтобы и по-русски, и по-французски, и по-английски. Это может быть как просто многоязычный блог, так и гостиничный сайт, сайт по работе с недвижимостью и многое другое.

Для начала определим, что же мы будем переводить:

- Написание дат, сумм в зависимости от выбранной локализации. С этим справляется класс System.Globalization
- Встроенные ресурсы сайта - выдача ошибки («Поле не может быть пустым», «The field is required») и другие сообщения.
- Не встроенные ресурсы, как то логотипы, изображения, js-локализация элементов управления. Для переключения между ними необходимо знать текущее значение языка на странице.
- Пользовательские значения.

Есть несколько вариантов решения этой задачи. Рассмотрим их.

- Самое простое, что можно сделать, - это разные сайты. Нужен русский сайт - сделали. Нужен перевод, скопировали сайт, перевели все данные и всё. Этот вариант приемлем, когда сайт небольшой, всего несколько страниц, статичный и нет админки.
- Разные БД. Сайт в зависимости от выбранной локализации подключается к одной или другой БД. Если необходимо добавить новый язык – то переводится вся БД и ресурсы сайта. Но БД могут быть разные, и статья, написанная на одном языке, не будет доступна для перевода на другой, плюс будет необходимость дублировать изображения, которые в принципе нет необходимости переводить.
- Управление локализацией при работе с базой данных.

Сразу рассмотрим третий вариант и определим, как мы это организуем в приложении:

- В адресе сайта теперь появляется параметр lang
  - Возможно, адрес будет иметь вид <http://our-site.com/{lang}/{controller}/{action}>
  - Возможно, адрес будет иметь вид <http://our-site.com/{controller}/{action}?lang=ru>
- Параметр lang – это ISO двухбуквенное определение языка (ru – русский, uk – украинский, cs – чешский)
- Первым делом мы используем System.Globalization для правильного вывода дат
- Организуем внутренние ресурсы для вывода ошибки относительно заданных языков
- Организуем таблицы в БД таким образом, чтобы для каждой записи существовал перевод необходимых полей.

Мы будем создавать 2 локализации – русскую и английскую, причем русская будет по умолчанию.

## Routing

В DefaultAreaRegistration добавим обработку lang (/Areas/Default/DefaultAreaRegistration.cs):

```
context.MapRoute(
 name: "lang",
 url: "{lang}/{controller}/{action}/{id}",
 defaults: new { controller = "Home", action = "Index", id =
UrlParameter.Optional },
 constraints : new { lang = @"ru|en" },
 namespaces: new[] { "LessonProject.Areas.Default.Controllers" }
);

context.MapRoute(
 name : "default",
 url : "{controller}/{action}/{id}",
 defaults : new { controller = "Home", action = "Index", id =
UrlParameter.Optional, lang = "ru" },
 namespaces : new [] { "LessonProject.Areas.Default.Controllers" }
);
```

Итак, если строка у нас начинается с lang, то мы используем обработку маршрута "lang". Обратите внимание на constraints (ограничения), тут задается, что язык может быть только ru или en. Если это условие не выполняется, то мы переходим к следующей обработке маршрута – "default", где по-умолчанию lang=ru.

Используем это для инициализации в DefaultController для смены культуры потока (Thread.Current.CurrentCulture) (/Areas/Default/DefaultController.cs):

```
public class DefaultController : BaseController
{
 public string CurrentLangCode { get; protected set; }

 public Language CurrentLang { get; protected set; }

 protected override void Initialize(System.Web.Routing.RequestContext
requestContext)
 {
 if (requestContext.HttpContext.Request.Url != null)
 {
 HostName = requestContext.HttpContext.Request.Url.Authority;
 }

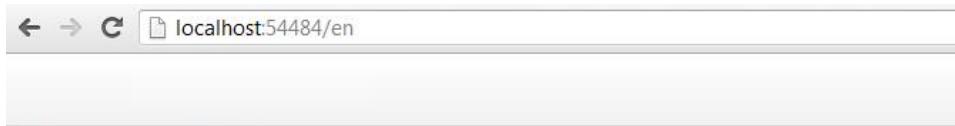
 if (requestContext.RouteData.Values["lang"] != null &&
requestContext.RouteData.Values["lang"] as string != "null")
 {
 CurrentLangCode = requestContext.RouteData.Values["lang"] as string;
 CurrentLang = Repository.Languages.FirstOrDefault(p => p.Code ==
CurrentLangCode);

 var ci = new CultureInfo(CurrentLangCode);
 Thread.CurrentCulture = ci;
 Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture(ci.Name);
 }
 base.Initialize(requestContext);
 }
}
```

Естественно, в BaseController мы убираем инициализацию культуры потока через конфигурационный файл (/Controllers/BaseController.cs):

```
protected override void Initialize(System.Web.Routing.RequestContext requestContext)
{
 if (requestContext.HttpContext.Request.Url != null)
 {
 HostName = requestContext.HttpContext.Request.Url.Authority;
 }
 base.Initialize(requestContext);
}
```

Запускаем, и проверяем, как изменяется вывод даты:



Первый этап пройден. Переходим к управлению ресурсам сайта.

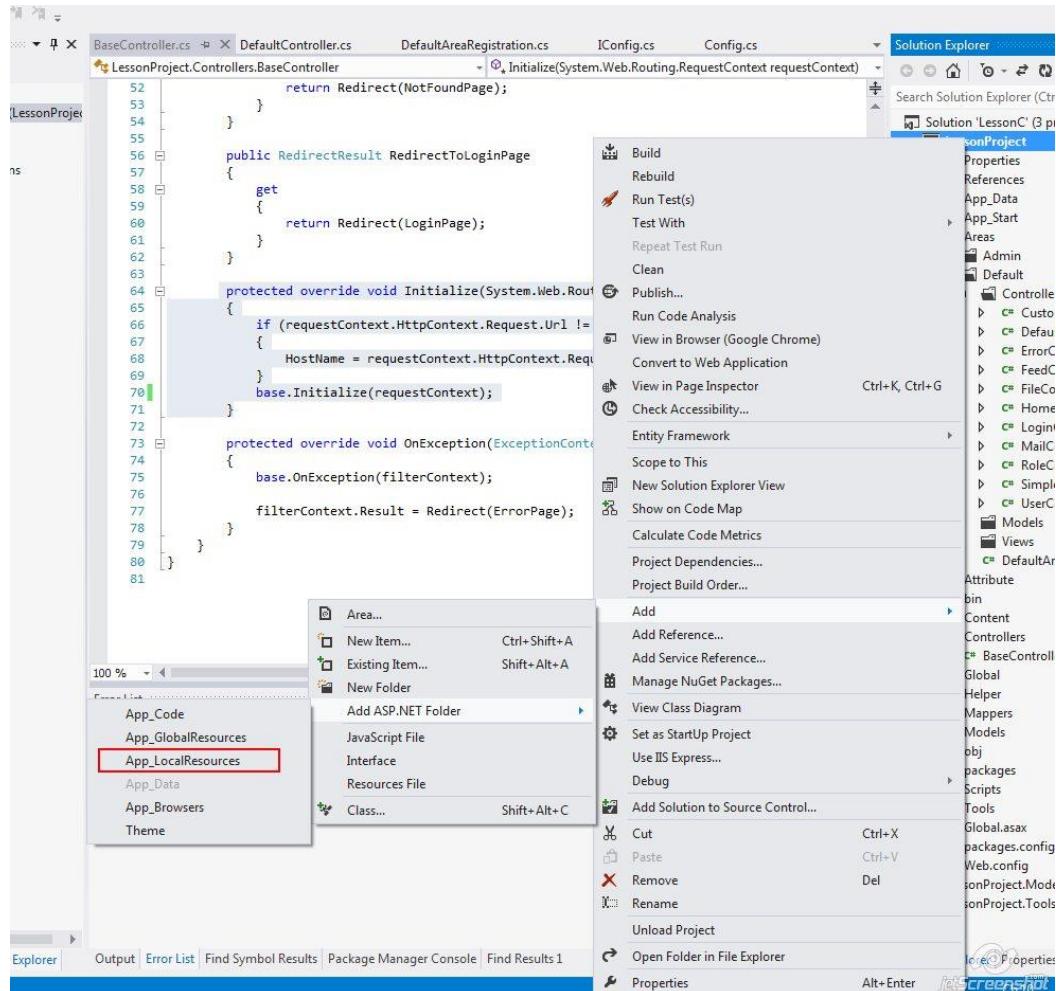
## Ресурсы сайта

Ресурсы сайта – это все статические строки, которые надо перевести:

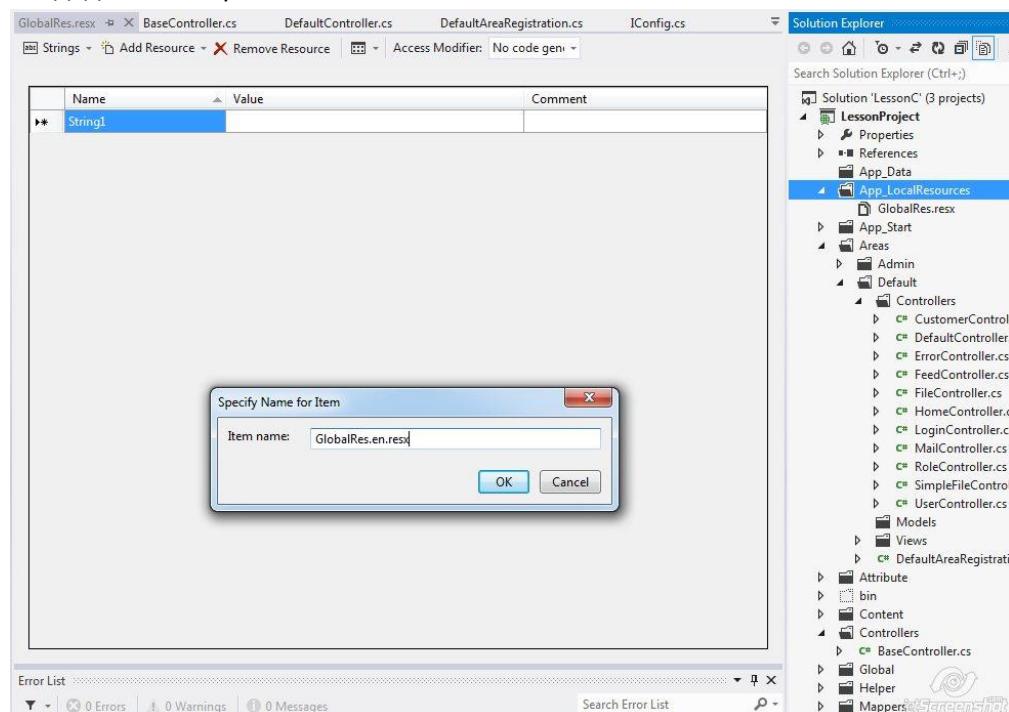
- Наименования меню
- Подсказки
- Выводы ошибок

На главной странице у нас таких строк четыре: роли, пользователи, вход и регистрация. Создадим ресурсные файлы:

- Добавим папку Asp.net папку App\_LocalResources:



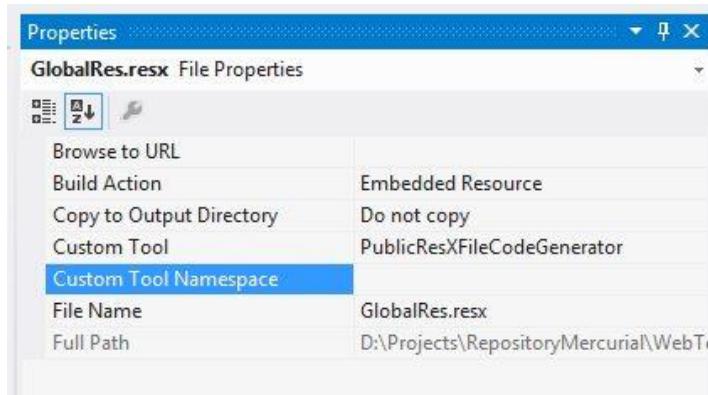
- Создадим в ней файлы GlobalRes.resx и GlobalRes.en.resx:



- Добавляем в них наши строки, в GlobalRes – русский перевод, в GlobalRes.en – английский:

Enter	Вход
Register	Регистрация
Roles	Роли
Users	Пользователи

- Открываем для GlobalRes свойства и устанавливаем следующие значения для полей
  - Build Action : Embedded Resource
  - Custom Tool : PublicResXFileCodeGenerator



- Теперь добавим namespace LessonProject.App\_LocalResources в Web.config в system.web.webPages.razor (Web.config):

```
<system.web.webPages.razor>
<pages pageBaseType="System.Web.Mvc.WebViewPage">
 <namespaces>
 <add namespace="LessonProject.Helper" />
 <add namespace="LessonProject.Tools" />
 <add namespace="LessonProject.App_LocalResources" />
 </namespaces>
</pages>
</system.web.webPages.razor>
```

- Используем в UserLogin.cshtml (/Areas/Default/Views/Home/UserLogin.cshtml) и Index.cshtml (/Areas/Default/Views/Home/Index.cshtml):

```
@model LessonProject.Model.User

@if (Model != null)
{
 @Model.Email
 @Html.ActionLink("Выход", "Logout", "Login")
}
else
{
 @GlobalRes.Enter
 @Html.ActionLink(GlobalRes.Register, "Register", "User")
}
```

...

```

@{
 ViewBag.Title = "LessonProject";
 Layout = "~/Areas/Default.Views/Shared/_Layout.cshtml";
}
<h2>LessonProject </h2>
<p>
 @DateTime.Now.ToString("D")
 <div class="menu">
 @GlobalRes.Roles
 @Html.ActionLink(GlobalRes.Users, "Index", "User")
 </div>
</p>

```

Запускаем, проверяем:

The screenshot displays two browser windows side-by-side. Both windows have the URL `localhost:54484` in the address bar.

- Top Window:** Shows the main application page titled "LessonProject". Below the title is a timestamp "11 марта 2013 г.". A navigation menu is present, with the "Roles" item highlighted in blue. The menu items include "Roles" and "Users".
- Bottom Window:** Shows the "Roles" page of the application. It has a timestamp "Monday, March 11, 2013" and a navigation menu with "Enter" and "Register" links. Below the menu, there is a table with two rows. The first row contains the header "EnterEmail" and the message "Введите email". The second row contains the header "EnterPassword" and the message "Введите пароль".

Перейдем к заданию сообщений валидации на примере LoginView:

- Выделяем ErrorMessage для полей в ресурсные файлы (`/App_LocalResources/GlobalRes.resx`):

EnterEmail	Введите email
EnterPassword	Введите пароль

- Задаем правила валидации в `LoginView.cs` (`/Models/ViewModel/LoginView.cs`):

```

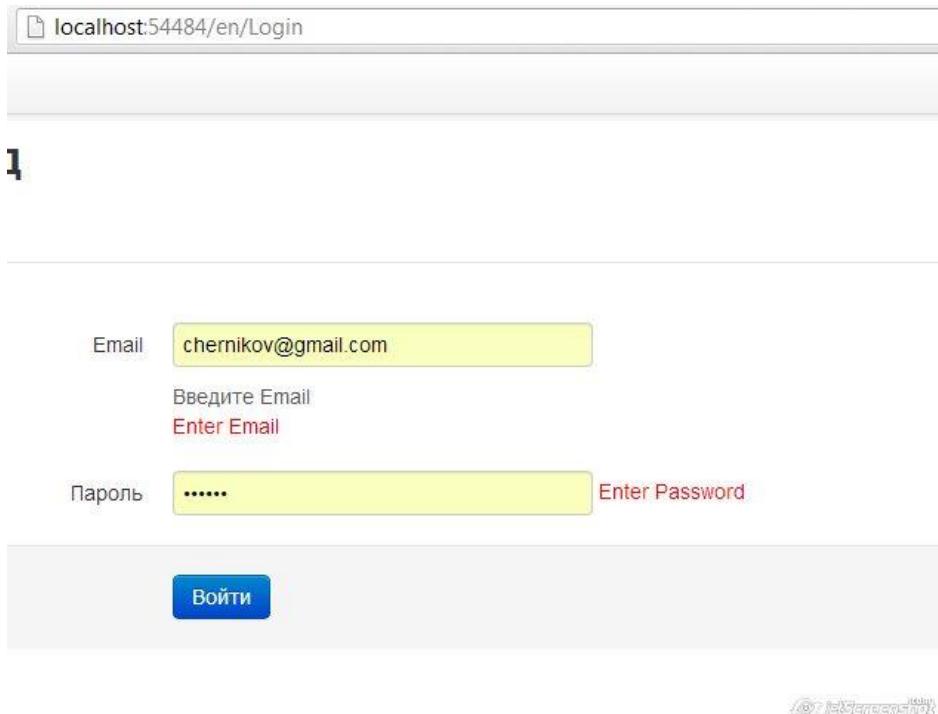
public class LoginView
{
 [Required(ErrorMessageResourceType=typeof(GlobalRes),
ErrorMessageResourceName="EnterEmail")]
 public string Email { get; set; }

 [Required(ErrorMessageResourceType = typeof(GlobalRes), ErrorMessageResourceName
= "EnterPassword")]
 public string Password { get; set; }

 public bool IsPersistent { get; set; }
}

```

Проверяем в страничной версии <http://localhost/en/Login>:



Но для рорир входа эти предупреждения так и останутся на русском языке, потому, что мы для вызова рорир-блока используем url по умолчанию. Соответственно, задав параметр lang, мы сможем изменить это:

- Добавим в ресурсы (/App\_LocalResources/GlobalRes(en).resx) CurrentLang = ru и CurrentLang = en
- Выведем это в hidden-поле в \_Layout.cshtml (/Areas/Default/Views/Shared/\_Layout.cshtml):

```
<body>
 @Html.Hidden("CurrentLang", GlobalRes.CurrentLang)
 <div class="navbar navbar-fixed-top">

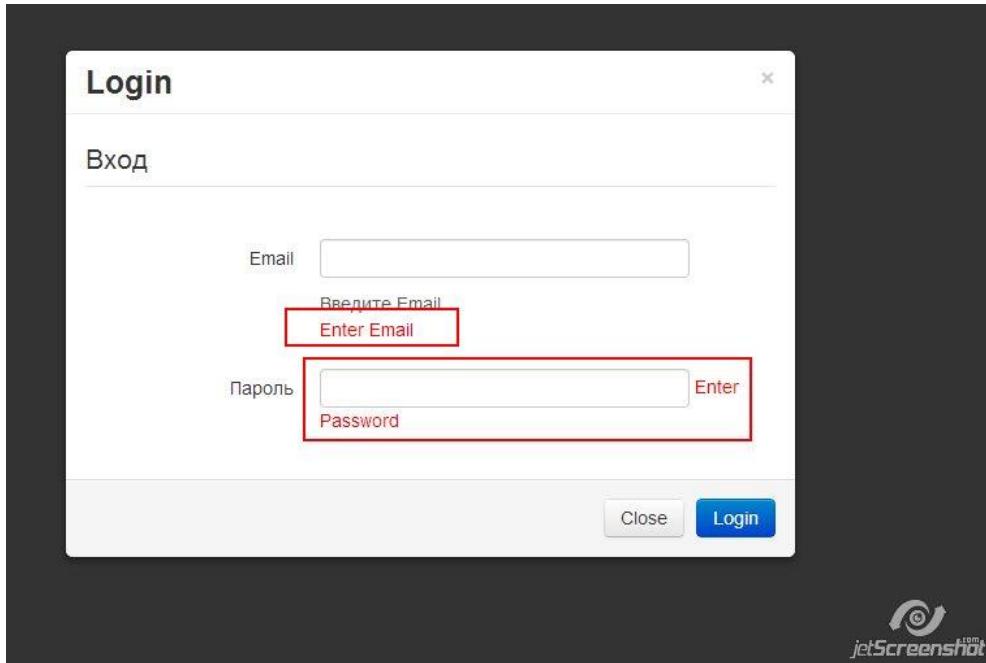
 • Добавим это в ajax-вызовы (/Scripts/common.js):
```

```
_this = this;
this.loginAjax = "/Login/Ajax";

this.init = function ()
{
 this.loginAjax = "/" + $("#CurrentLang").val() + _this.loginAjax;
 $("#LoginPopup").click(function () {
 _this.showPopup(_this.loginAjax, initLoginPopup);
 });
}
```

```
function initLoginPopup(modal) {
 $("#LoginButton").click(function () {
 $.ajax({
 type: "POST",
 url: _this.loginAjax,
 data : $("#LoginForm").serialize(),
```

Проверяем:



jetScreenshot.com

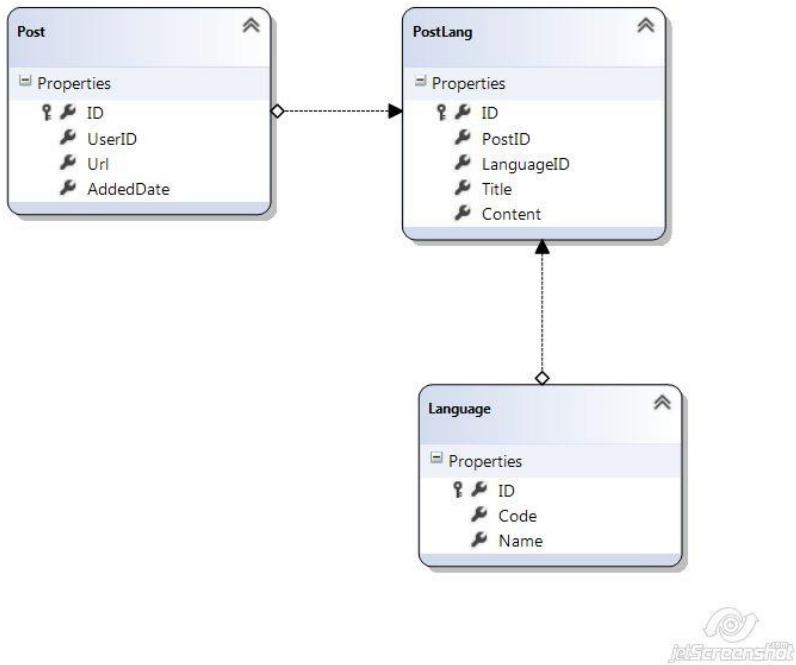
## База данных

Переходим к самому важному разделу, работе с БД. Например, у нас есть объект типа Post (блог-запись), которая, естественно, должна быть на двух языках:

ID	Уникальный номер записи	
UserID	Автор записи	
Header	Заголовок	Требует перевода
Url	Url записи	
Content	Содержимое записи	Требует перевода
AddedDate	Дата добавления	

Итак, как это всё будет организовано:

- Создадим таблицу Language, где и будут определены языки
- Создадим таблицу Post, где будут все поля, не требующие перевода
- Создадим таблицу PostLang, связанную с Post и Language, где будет перевод необходимых полей для таблицы Post и связанный с таблицей Language



Ок, теперь добавим это в LessonProject.Model (LessonProject.Model/IRepository.cs):

```

#region Language

IQueryable<Language> Languages { get; }

bool CreateLanguage(Language instance);

bool UpdateLanguage(Language instance);

bool RemoveLanguage(int idLanguage);

#endregion

#region Post

IQueryable<Post> Posts { get; }

bool CreatePost(Post instance);

bool UpdatePost(Post instance);

bool RemovePost(int idPost);

#endregion

```

Создаем модели с помощью уже созданных снippetов /Proxy/Language.cs:

```

namespace LessonProject.Model
{
 public partial class Language
 {
 }
}

```

/Proxy/Post.cs:

```
namespace LessonProject.Model
{
 public partial class Post
 {
 }
}
```

/SqlRepository/Language.cs:

```
public partial class SqlRepository
{
 public IQueryable<Language> Languages
 {
 get
 {
 return Db.Languages;
 }
 }

 public bool CreateLanguage(Language instance)
 {
 if (instance.ID == 0)
 {
 Db.Languages.InsertOnSubmit(instance);
 Db.Languages.Context.SubmitChanges();
 return true;
 }

 return false;
 }

 public bool UpdateLanguage(Language instance)
 {
 Language cache = Db.Languages.Where(p => p.ID ==
instance.ID).FirstOrDefault();
 if (cache != null)
 {
 cache.Code = instance.Code;
 cache.Name = instance.Name;
 Db.Languages.Context.SubmitChanges();
 return true;
 }

 return false;
 }

 public bool RemoveLanguage(int idLanguage)
 {
 Language instance = Db.Languages.Where(p => p.ID ==
idLanguage).FirstOrDefault();
 if (instance != null)
 {
 Db.Languages.DeleteOnSubmit(instance);
 Db.Languages.Context.SubmitChanges();
 return true;
 }

 return false;
 }
}
```

/SqlRepository/Post.cs:

```
public partial class SqlRepository
{
 public IQueryable<Post> Posts
 {
 get
 {
 return Db.Posts;
 }
 }

 public bool CreatePost(Post instance)
 {
 if (instance.ID == 0)
 {
 Db.Posts.InsertOnSubmit(instance);
 Db.Posts.Context.SubmitChanges();
 return true;
 }

 return false;
 }

 public bool UpdatePost(Post instance)
 {
 Post cache = Db.Posts.Where(p => p.ID == instance.ID).FirstOrDefault();
 if (cache != null)
 {
 //TODO : Update fields for Post
 Db.Posts.Context.SubmitChanges();
 return true;
 }

 return false;
 }

 public bool RemovePost(int idPost)
 {
 Post instance = Db.Posts.Where(p => p.ID == idPost).FirstOrDefault();
 if (instance != null)
 {
 Db.Posts.DeleteOnSubmit(instance);
 Db.Posts.Context.SubmitChanges();
 return true;
 }

 return false;
 }
}
```

Итак, у нас есть набор PostLangs в объекте класса Post, где и хранятся различные переводы. Причем, перевод на английский или русский язык может быть, так может и не быть. Но, по крайней мере, хотя бы один язык должен быть. Что необходимо сделать для этого:

- Добавим языковые поля в Post (Header, Content)
- Создадим свойство CurrentLang, при изменении которого будут инициализироваться языковые поля.
- При создании записи в БД Post автоматически создается запись в БД PostLang.

- При изменении записи в БД, проверяется, какой именно язык изменяется, и если такого языка (перевода) еще нет, то создается новая запись PostLang в БД:

Перейдем к реализации (/Proxy/Post.cs):

```
public partial class Post
{
 private int _currentLang;

 public int CurrentLang
 {
 get
 {
 return _currentLang;
 }

 set
 {
 _currentLang = value;

 var currentLang = PostLangs.FirstOrDefault(p => p.LanguageID == value);
 if (currentLang == null)
 {
 IsCorrectLang = false;
 var anyLang = PostLangs.FirstOrDefault();
 if (anyLang != null)
 {
 SetLang(anyLang);
 }
 }
 else
 {
 IsCorrectLang = true;
 SetLang(currentLang);
 }
 }
 }

 private void SetLang(PostLang postLang)
 {
 Header = postLang.Header;
 Content = postLang.Content;
 }

 public bool IsCorrectLang { get; protected set; }

 public string Header { get; set; }

 public string Content { get; set; }
}
```

Тут важно заметить, что если необходимого перевода нет, то берется первый попавшийся, и устанавливается IsCorrectLang = false. Это для того, что лучше показать пользователю хоть какую-то информацию, чем не показать ничего.

Создание/изменение объекта Post (/SqlRepository/Post.cs):

```
public bool CreatePost(Post instance)
{
 if (instance.ID == 0)
 {
 instance.AddedDate = DateTime.Now;
 Db.Posts.InsertOnSubmit(instance);
```

```

 Db.Posts.Context.SubmitChanges();
 var lang = Db.Languages.FirstOrDefault(p => p.ID ==
instance.CurrentLang);
 if (lang != null)
 {
 CreateOrChangePostLang(instance, null, lang);
 return true;
 }
 }

 return false;
}

public bool UpdatePost(Post instance)
{
 Post cache = Db.Posts.Where(p => p.ID == instance.ID).FirstOrDefault();
 if (cache != null)
 {
 cache.Url = instance.Url;
 Db.Posts.Context.SubmitChanges();

 var lang = Db.Languages.FirstOrDefault(p => p.ID ==
instance.CurrentLang);
 if (lang != null)
 {
 CreateOrChangePostLang(instance, cache, lang);
 return true;
 }
 return true;
 }

 return false;
}

private void CreateOrChangePostLang(Post instance, Post cache, Language lang)
{
 PostLang postLang = null;
 if (cache != null)
 {
 postLang = Db.PostLangs.FirstOrDefault(p => p.PostID == cache.ID &&
p.LanguageID == lang.ID);
 }
 if (postLang == null)
 {
 var newPostLang = new PostLang()
 {
 PostID = instance.ID,
 LanguageID = lang.ID,
 Header = instance.Header,
 Content = instance.Content,
 };
 Db.PostLangs.InsertOnSubmit(newPostLang);
 }
 else
 {
 postLang.Header = instance.Header;
 postLang.Content = instance.Content;
 }
 Db.PostLangs.Context.SubmitChanges();
}

```

Рассмотрим, как работает CreateOrChangePostLang функция:

- При вызове данной функции, мы ищем в Language необходимый язык. Если язык не найден, то вызова не происходит, и мы не создаем PostLang объект (т.е. перевод)
- Если мы находим необходимый язык, то вызываем CreateOrChangePostLang:
  - Если cache нулевое (объект PostLang еще точно не создан) или
  - Если cache не нулевое, но перевод не найден, то
    - Создаем перевод (запись в БД PostLang)
  - Иначе изменяем перевод, который найден.

*При удалении записи Post все PostLang удаляются по связи OnDelete = cascade (проследите за этим)*

*В БД должны быть уже добавлены записи для необходимых языков:*

1	Ru	Русский
2	En	English

### Админка

Сейчас, чтобы это всё продемонстрировать, мы создадим админку. План действий таков (мы его еще потом доработаем и озвучим):

- Создать модели
- Связать язык ввода и пользователя (для того, чтобы было понятно, в каком языке работает администратор или редактор)
- Создать переключение между языками
- Создать домашнюю страницу админки
- Создать контроллер по работе с постами
- Вывести посты в default/postController части

Добавим в таблицу User LangaugeID:

Column Name	Data Type	Allow Nulls
ID	int	<input type="checkbox"/>
Email	nvarchar(150)	<input type="checkbox"/>
Password	nvarchar(50)	<input type="checkbox"/>
Birthdate	datetime	<input type="checkbox"/>
AddedDate	datetime	<input type="checkbox"/>
ActivatedDate	datetime	<input checked="" type="checkbox"/>
ActivatedLink	nvarchar(50)	<input type="checkbox"/>
LastVisitDate	datetime	<input checked="" type="checkbox"/>
AvatarPath	nvarchar(150)	<input checked="" type="checkbox"/>
LangaugeID	int	<input checked="" type="checkbox"/>

Добавляем в IRepository.cs:

```
bool ChangeLanguage(User instance, string LangCode);
```

Реализуем в /SqlRepository/User.cs:

```
public bool ChangeLanguage(User instance, string LangCode)
{
 var cache = Db.Users.FirstOrDefault(p => p.ID == instance.ID);
 var newLang = Db.Languages.FirstOrDefault(p => p.Code == LangCode);
 if (cache != null && newLang != null)
 {
 cache.Language = newLang;
 Db.Users.Context.SubmitChanges();
 return true;
 }

 return false;
}
```

Создаем модель /Models/ViewModel/PostView.cs:

```
public class PostView
{
 public int ID { get; set; }

 public int UserID { get; set; }

 public bool IsCorrectLang { get; set; }

 public int CurrentLang { get; set; }

 [Required(ErrorMessage = "Введите заголовок")]
 public string Header { get; set; }

 [Required]
 public string Url { get; set; }

 [Required(ErrorMessage = "Введите содержимое")]
 public string Content { get; set; }
}
```

Строки валидации не надо вставлять в GlobalRes, так как тут мы работаем только в админке и это нам ни к чему (так как администраторы люди скромные). Но если есть другие требования, то мы знаем что делать.

Создаем /Areas/Admin/Controller/AdminController.cs:

```
public abstract class AdminController : BaseController
{
 public Language CurrentLang
 {
 get
 {
 return CurrentUser != null ? CurrentUser.Language : null;
 }
 }

 protected override void Initialize(RequestContext requestContext)
 {
 CultureInfo ci = new CultureInfo("ru");

 Thread.CurrentThread.CurrentCulture = ci;
 base.Initialize(requestContext);
 }
}
```

И /Areas/Admin/Controller/HomeController.cs:

```
[Authorize(Roles="admin")]
public class HomeController : AdminController
{
 public ActionResult Index()
 {
 return View();
 }

 public ActionResult AdminMenu()
 {
 return View();
 }

 public ActionResult LangMenu()
 {
 if (CurrentLang == null)
 {
 var lang = repository.Languages.FirstOrDefault();
 repository.ChangeLanguage(currentUser, lang.Code);
 }
 var langProxy = new LangAdminView(repository, CurrentLang.Code);
 return View(langProxy);
 }

 [HttpPost]
 public ActionResult ChangeLanguage(string SelectedLang)
 {
 repository.ChangeLanguage(currentUser, SelectedLang);
 return Redirect("~/admin");
 }
}
```

Итак, AdminController выбирает и устанавливает, в каком языке мы сейчас работаем. Если данный язык не установлен, то выбирается первый попавшийся, и в HomeController.cs:LangMenu устанавливается для пользователя. Создадим LangAdminView.cs (/Models/ViewModel/LangAdminView.cs):

```
public class LangAdminView
{
 private IRepository Repository
 {
 get
 {
 return DependencyResolver.Current.GetService<IRepository>();
 }
 }

 public string SelectedLang {get; set; }

 public List<SelectListItem> Langs { get; set; }

 public LangAdminView(string currentLang)
 {
 currentLang = currentLang ?? "";
 Langs = new List<SelectListItem>();

 foreach (var lang in Repository.Languages)
 {
 Langs.Add(new SelectListItem()
 {
 Selected = (string.Compare(currentLang, lang.Code, true) == 0),
 });
 }
 }
}
```

```
 Value = lang.Code,
 Text = lang.Name
 });
}
}
}
```

Опишем все View (+js-файлы):

/Areas/Admin/Views/Shared/\_Layout.cshtml:

```
@{
 var currentUser =
((LessonProject.Controllers.BaseController)ViewContext.Controller).CurrentUser;
}
<!DOCTYPE html>
<html>
<head>
 <title>@ViewBag.Title</title>
 <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
 @Styles.Render("~/Content/css/jqueryui")
 @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-fluid">
 <div class="btn-group pull-right">
 <i
class="icon-user"></i>
 @currentUser.Email

 <ul class="dropdown-menu">
 На сайт
 <li class="divider">
 <a href="@Url.Action("Logout", "Login", new { area =
"Default" })">Выход

 </div>
 LessonProject
 </div>
 </div>
 <div class="container-fluid">
 <div class="row-fluid">
 <div class="span3">
 <div class="well sidebar-nav">
 <ul class="nav nav-list">
 @Html.Action("LangMenu", "Home")
 @Html.Action("AdminMenu", "Home")

 </div>
 <div class="span9">
 @RenderBody()
 </div>
 </div>
 </div>
 @Scripts.Render("~/bundles/jquery")
 @Scripts.Render("~/bundles/jqueryui")
```

```
@Scripts.Render("~/bundles/bootstrap")
@Scripts.Render("~/bundles/common")
@Scripts.Render("~/Scripts/admin/common.js")
@RenderSection("scripts", required: false)
</body>
</html>
```

Index.cshtml (/Areas/Admin/Views/Home/Index.cshtml):

```
@{
 ViewBag.Title = "Index";
 Layout = "~/Areas/Admin/Views/Shared/_Layout.cshtml";
}

<h2>Админка</h2>
```

AdminMenu.cshtml (/Areas/Admin/Views/Home/AdminMenu.cshtml):

```

 @Html.ActionLink("Главная", "Index", "Home")

 @Html.ActionLink("Посты", "Index", "Post")

```

LangMenu.cshtml (/Areas/Admin/Views/Home/LangMenu.cshtml):

```
@model LessonProject.Models.ViewModels.LangAdminView

 @using (Html.BeginForm("ChangeLanguage", "Home", FormMethod.Post, new { id = "SelectLangForm" }))
 {
 @Html.DropDownList("SelectedLang", Model.Langs)
 }

```

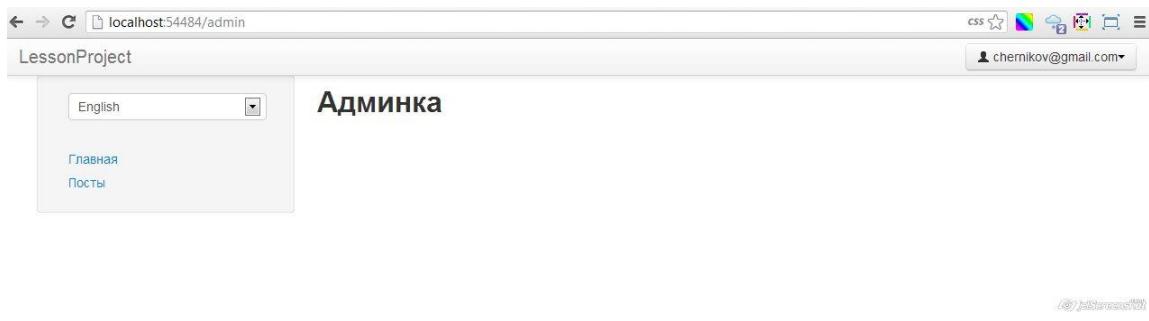
И обработчик SelectedLang (/Scripts/admin/common.js):

```
function AdminCommon()
{
 _this = this;

 this.init = function ()
 {
 $("#SelectedLang").change(function () {
 $("#SelectLangForm").submit();
 });
 }
}

var adminCommon = null;
$(().ready(function () {
 adminCommon = new AdminCommon();
 adminCommon.init();
});
```

Заходим под админом (у меня это chernikov@gmail.com) и переходим на страницу <http://localhost/admin>:



Если не удалось зайти и выкинуло на /Login, то проверьте связь UserRole в БД, чтобы текущий пользователь имел роль с кодом "admin".

Открываем выпадающий список языков. Он и показывает, в каком языке мы в данный момент работаем.

Добавляем контроллер PostController.cs (/Areas/Admin/Controllers/PostController.cs):

```
public class PostController : AdminController
{
 public ActionResult Index(int page = 1)
 {
 var list = Repository.Posts.OrderByDescending(p => p.AddedDate);
 var data = new PageableData<Post>(list, page);
 data.List.ForEach(p => p.CurrentLang = CurrentLang.ID);
 return View(data);
 }

 [HttpGet]
 public ActionResult Create()
 {
 var postView = new PostView
 {
 CurrentLang = CurrentLang.ID
 };
 return View("Edit", postView);
 }

 [HttpGet]
 public ActionResult Edit(int id)
 {
 var post = Repository.Posts.FirstOrDefault(p => p.ID == id);
 if (post != null)
 {
 post.CurrentLang = CurrentLang.ID;
 var postView = (PostView)ModelMapper.Map(post, typeof(Post),
typeof(PostView));
 return View(postView);
 }
 return RedirectToAction("NotFound");
 }

 [HttpPost]
 [ValidateInput(false)]
 public ActionResult Edit(PostView postView)
 {
 if (ModelState.IsValid)
```

```

 {
 var post = (Post)ModelMapper.Map(postView, typeof(PostView),
typeof(Post));
 post.CurrentLang = CurrentLang.ID;
 if (post.ID == 0)
 {
 post.UserID = CurrentUser.ID;
 Repository.CreatePost(post);
 }
 else
 {
 Repository.UpdatePost(post);
 }
 TempData["Message"] = "Сохранено!";
 return RedirectToAction("Index");
 }
 return View(postView);
 }

 public ActionResult Delete(int id)
 {
 Repository.RemovePost(id);
 TempData["Message"] = "Удален пост";

 return RedirectToAction("Index");
 }
}

```

Изменим PageableData, чтобы можно было сделать Foreach (/Models/Info/PageableData.cs):

```

public class PageableData<T> where T : class
{
 protected static int ItemPerPageDefault = 20;

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

 ...

 public PageableData(IQueryable<T> queryableSet, int page, int itemPerPage = 0)
 {
 ...
 List = queryableSet.Skip((PageNo - 1) * itemPerPage).Take(itemPerPage).ToList();
 }
}

```

Index.cshtml (/Areas/Admin/Views/Post/Index.cshtml):

```

@model LessonProject.Models.Info.PageableData<LessonProject.Model.Post>

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

Посты

@Html.ActionLink("Добавить", "Create", "Post", null, new { @class = "btn" })

|--|


```

```

 #
 </th>
 <th>
 перевод
 </th>
 <th>
 Наименование
 </th>

 <th>
 </th>
</tr>
</thead>
@foreach (var item in Model.List)
{
 <tr>
 <td>
 @item.ID
 </td>
 <td>
 @(item.IsCorrectLang ? "" : "нужен перевод")
 </td>
 <td>
 @item.Header
 </td>
 <td>
 @Html.ActionLink("Изменить", "Edit", "Post", new { id = item.ID }, new {
@class = "btn btn-mini" })
 @Html.ActionLink("Удалить", "Delete", "Post", new { id = item.ID }, new {
@class = "btn btn-mini btn-danger" })
 </td>
 </tr>
}
</table>

```

При инициализации в ForEach, в каждом объекте уже инициализируются языковые поля. Язык – тот, с которым в данный момент работаем в админке.

View для редактирования тривиальна, так как мы всю работу делаем в Controller, а наш PostView уже использует языковые настройки. (/Areas/Admin/Views/Post/Edit.cshtml):

```

@model LessonProject.Models.ViewModels.PostView

 @{
 ViewBag.Title = Model.ID == 0 ? "Добавить пост" : "Изменить пост";
 Layout = "~/Areas/Admin/Views/Shared/_Layout.cshtml";
}

<h2>@(Model.ID == 0 ? "Добавить пост" : "Изменить пост")</h2>
<p>
</p>
@using (Html.BeginForm("Edit", "Post", FormMethod.Post))
{
 @Html.Hidden("ID", Model.ID)
 <fieldset>
 <div class="control-group">
 <label class="control-label">
 @(!Model.IsCorrectLang && Model.ID != 0 ? "нужен перевод" : "")
 </label>
 </div>
 <div class="control-group">
 <label class="control-label">
 Заголовок</label>
 <div class="controls">

```

```

 @Html.TextBox("Header", Model.Header, new { @class = "input-xlarge" })
 @Html.ValidationMessage("Header")
 </div>
</div>
<div class="control-group">
 <label class="control-label">
 Url</label>
 <div class="controls">
 @Html.TextBox("Url", Model.Url, new { @class = "input-xlarge" })
 @Html.ValidationMessage("Url")
 </div>
</div>
<div class="control-group">
 <label class="control-label">
 Содержимое</label>
 <div class="controls">
 @Html.TextArea("Content", Model.Content, new { @class = "input-xlarge" })
 @Html.ValidationMessage("Content")
 </div>
</div>
<div class="form-actions">
 <button type="submit" class="btn btn-primary">
 Сохранить</button>
 @Html.ActionLink("Отменить", "Index", null, null, new { @class = "btn" })
</div>
</fieldset>
}

```

Обратите внимание на подсказку о необходимости перевода. В данном случае, поля уже будут заполнены, и их нужно перевести и сохранить. Таким образом, будет добавлен перевод.

Добавляем пару постов и переводим их:

#	перевод	Наименование
2	first post	<a href="#">Изменить</a> <a href="#">Удалить</a>
1	Тет	<a href="#">Изменить</a> <a href="#">Удалить</a>

Ок, посты созданы.

Создадим PostController в Default Area и выведем посты  
(/Areas/Default/Controller/PostController.cs):

```

public class PostController : DefaultController
{
 public ActionResult Index(int page = 1)
 {
 var list = Repository.Posts.OrderByDescending(p => p.AddedDate);
 var data = new PageableData<Post>(list, page);
 data.List.ForEach(p => p.CurrentLang = CurrentLang.ID);
 return View(data);
 }
}

```

Index.cshtml (/Areas/Default/Views/Post/Index.cshtml):

```
@model LessonProject.Models.Info.PageableData<LessonProject.Model.Post>

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

@foreach (var post in Model.List)
{
 <h3>@post.Header</h3>
 <p>
 @post.Content.NlToBr()
 </p>
 @post.AddedDate.ToString("d")
}
</div>
<div class="pagination">
 @Html.PageLinks(Model.PageNo, Model.CountPage, x => Url.Action("Index", new {page = x}))
</div>


```

И проверяем:



## первый пост

первый текст

11.03.2013

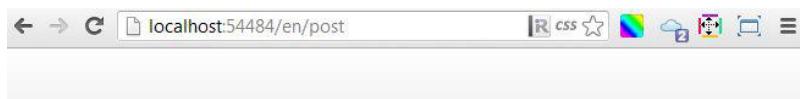
## Tet

Тет празднуется одновременно с китайским Новым годом, хотя есть расхождение во времени, возникающее в связи с разницей во времени между Ханоем и Пекином.

Празднование проходит с первого дня первого месяца по лунному календарю (примерно в конце января или начале февраля) и продолжается до недели. Тет считается первым днём весны, поэтому праздник часто ещё называют «праздник весны» — хой суан (Hội xuân).

Тет — семейный праздник, люди стараются вернуться из путешествий и встретить его с родственниками[1][2]. Детям дарят красные конверты с деньгами и одежду, взамен дети жалуют старикам долголетия[3].

11.03.2013



## first post

first text

3/11/2013

## Tet

Vietnamese New Year, more commonly known by its shortened name Tết or Tết Nguyên Đán, is the most important and popular holiday and festival in Vietnam. It is the Vietnamese New Year marking the arrival of spring based on the Chinese calendar, a lunisolar calendar.

3/11/2013



Супер!

## Переключение между языками

Создадим переключалку ru/en в клиентской части. Добавляем класс LangHelper.cs (/Helper/LangHelper.cs):

```
public static class LangHelper
{
 public static MvcHtmlString LangSwitcher(this UrlHelper url, string Name,
 RouteData routeData, string lang)
 {
 var liTagBuilder = new TagBuilder("li");
 var aTagBuilder = new TagBuilder("a");
 var routeValueDictionary = new RouteValueDictionary(routeData.Values);
 if (routeValueDictionary.ContainsKey("lang"))
 {
 if (routeData.Values["lang"] as string == lang)
 {
 liTagBuilder.AddCssClass("active");
 }
 else
 {
 routeValueDictionary["lang"] = lang;
 }
 }
 aTagBuilder.MergeAttribute("href", url.RouteUrl(routeValueDictionary));
 aTagBuilder.SetInnerText(Name);
 liTagBuilder.InnerHtml = aTagBuilder.ToString();
 return new MvcHtmlString(liTagBuilder.ToString());
 }
}
```

Добавляем Partial в \_Layout.cshtml (/Areas/Default/Views/Shared/\_Layout.cshtml):

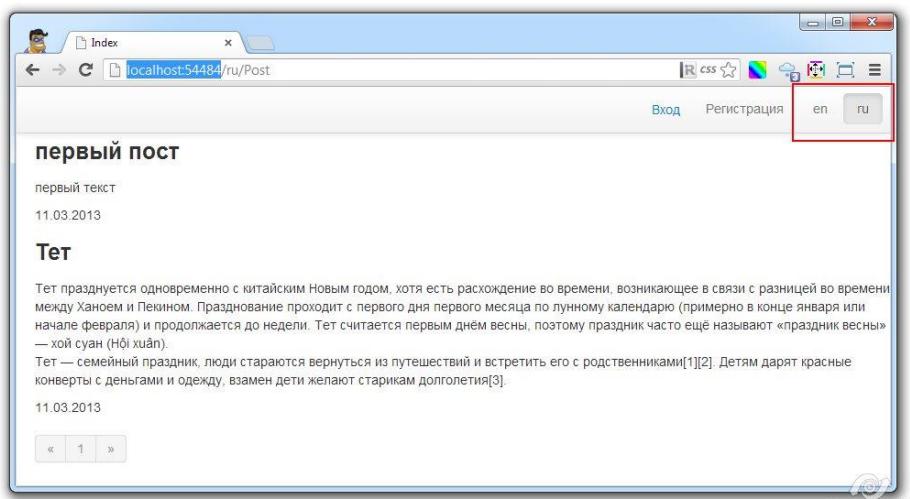
```
<div class="container">
 <ul class="nav nav-pills pull-right">
 @Html.Partial("LangMenu")

```

+ LangMenu.cshtml:

```
@Url.LangSwitcher("en", ViewContext.RouteData, "en")
@Url.LangSwitcher("ru", ViewContext.RouteData, "ru")
```

Запускаем. Вуаля! Красота.



## Неверный формат, перевод на русский

Иногда, когда мы вводим в ожидаемое числовое поле текстовое значение, то можем получить следующее сообщение:

The value 'one hundred dollars' is not valid for Price.

Но как это сообщение вывести на русском. Следующие действия помогут это сделать:

- Добавить папку App\_GlobalResources
- Добавить ресурс Messages.resx
- Добавить строку "PropertyValueInvalid: **Значение {0} недопустимо для поля {1}**"
- В App\_Start добавить строку в Application\_Start() (/Global.asax.cs)  
`DefaultModelBinder.ResourceClassKey = "Messages";`
- Для указания имени поля можно использовать атрибут Display[Name="Цена"]

Получаем результат:

Цена	<input type="text" value="45ыв"/>	Значение 45ыв недопустимо для поля Цена
------	-----------------------------------	-----------------------------------------

## Итог

Работа в многоязычном сайте заключается в следующем:

- Отделить переводные строки в ресурсы
- Определить языковые поля в таблицах БД и связать их через таблицу Language
- Использовать ajax-запросы с учетом языковой поддержки

Напоследок скажу, что не стоит начинать делать многоязычный сайт, если заказчик явно это не говорит, но, если в обозримом будущем будет использоваться такая возможность, то начинать строить сайт необходимо с использованием многоязычности, хотя бы на уровне БД.

## Урок D. 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.

В этом и следующем уроке мы изучим то, что поможет вам в разы быстрее разрабатывать приложения. Начнем издалека. Когда я делал первый сайт, я смотрел, как можно реализовать тот или иной функционал и использовал его у себя в приложении. Потом, когда у меня появился второй проект, я начал функционал улучшать. Я выделил основные моменты и инструменты, которые были описаны в предыдущих уроках. Я начал замечать, что я делаю часто много механичной работы, например:

- создать в БД новую таблицу
- прокинуть ее в класс DbContext
- добавить объявление в интерфейс репозитария
- добавить реализацию в SqlRepository
- добавить partial-часть класса в папке Proxy
- добавить модель данных
- объявить mapping
- создать контроллер в админке
- сделать типичные view для просмотра и редактирования

И так как это было поистине скучно, я часто ошибался в одном из шагов – и нужно было править банальные ошибки. И я создал сниппеты, но они решали только половину задачи, а вот модель данных, контроллер, index.cshtml, edit.cshtml – это не было решено.

И вот я прочитал статью Стивена Сандерсона «[Scaffold your ASP.NET MVC 3 project with the MvcScaffolding package](#)» и загорелся. Скаффолдинг подходил мне идеально, но он не был написан для моего решения. И я начал изучать. В основе его стоял T4 ([Text Template Transformation Toolkit](#)), в шаблонах используется именно этот синтаксис, но для работы дошаблонной логики используется [Windows PowerShell](#). Собственно, с PowerShell мы работаем в PackageManager Console (ух, как закрученко!). Я совсем немного погружусь в Windows PowerShell и T4, только для того, чтобы создать пару скаффолдеров для работы с проектом.

Итак, что нам изначально необходимо, так это установить [PowerGUI](#) для работы с PowerShell. В VS2010 есть много редакторов для PowerShell. Но мы работаем с VS2012 и нам пока так не повезло.

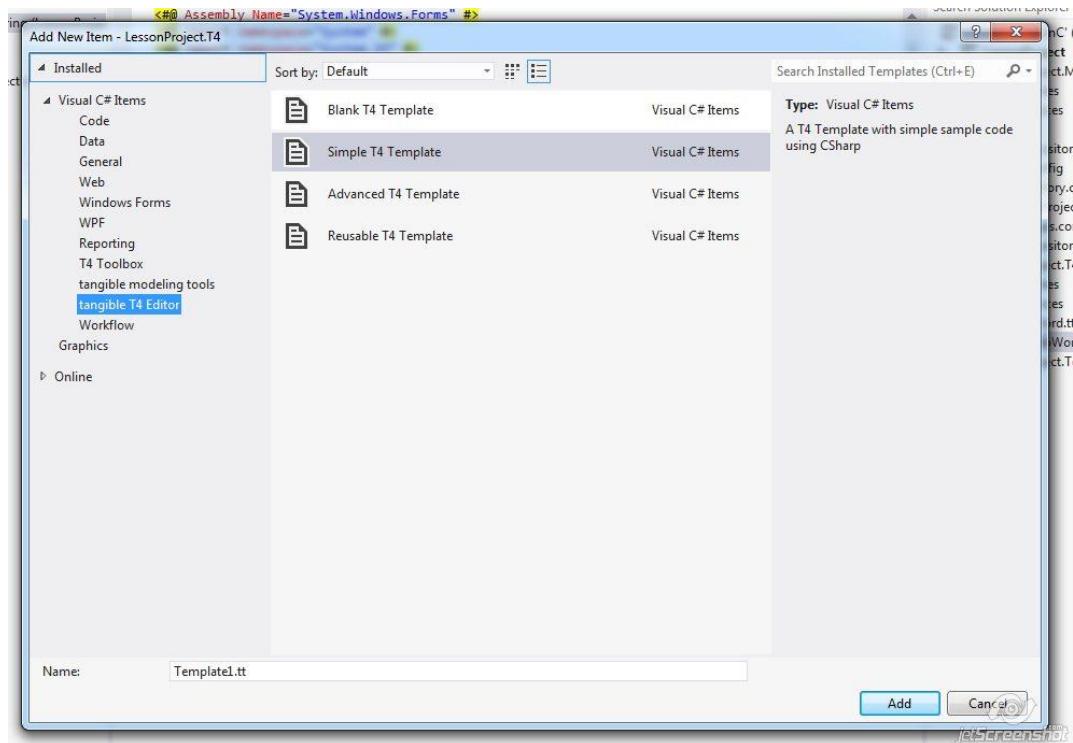
Ок, установили. Переходим к установке редактора для t4 - <http://t4-editor.tangible-engineering.com>. Тоже пока что единственный редактор для VS2012. Ну что ж – подсветочка есть и ладно.

## T4

Далее изучим, что у нас есть. Начнем с T4. Я пользовался этой ссылкой:

<http://www.olegsych.com/2007/12/text-template-transformation-toolkit/>

Создадим новый проект, библиотеку классов LessonProject.T4. И добавим туда HelloWorld.tt:



Изменим немного:

```
<#@ template debug="true" hostSpecific="true" #>
<#@ output extension=".cs" #>
<#@ Assembly Name="System.Core" #>
<#@ Assembly Name="System.Windows.Forms" #>
<#@ import namespace="System" #>
<#@ import namespace="System.IO" #>
<#@ import namespace="System.Diagnostics" #>
<#@ import namespace="System.Linq" #>
<#@ import namespace="System.Collections" #>
<#@ import namespace="System.Collections.Generic" #>
<#
 var greeting = "Hello, World!";
#>
// This is the output code from your template
// you only get syntax-highlighting here - not intellisense
namespace MyNameSpace
{
 class MyGeneratedClass
 {
 static void main (string[] args)
 {
 System.Console.WriteLine("<#= greeting #>");
 }
 }
}

<#+
// Insert any template procedures here
void foo(){}
#>
```

Ок, и результатом этого будет:

```
// This is the output code from your template
// you only get syntax-highlighting here - not intellisense
namespace MyNameSpace
{
 class MyGeneratedClass
 {
 static void main (string[] args)
 {
 System.Console.WriteLine("Hello, World!");
 }
 }
}
```

На самом деле .tt файл преобразуется в код, который создает конкретный класс, наследуемый от `TextTransformation`. Этот код запускается и генерируется файл-результат. Выглядит примерно так:

```
<#@ template language="C#" #>
Hello World!
```

Преобразуется в:

```
public class GeneratedTextTransform :
Microsoft.VisualStudio.TextTemplating.TextTransformation
{
 public override string TransformText()
 {
 this.Write("Hello, World!");
 return this.GenerationEnvironment.ToString();
 }
}
```

А итогом будет файл .cs:

```
Hello World!
```

Изучим блоки и синтаксис задания шаблонов, который очень похож на aspx, только вместо скобок `<% %>` используется `<# #>`. Но, так как aspx мы не изучали, то:

- Текстовый блок – это любой не программный текст в тексте шаблона (извините за тафтологию):

```
<#@ template language="C#" #>
Hello World!
```

- Блок операторов – это любой блок, заключенный в `<# #>`. Всё что внутри этого, это языковая конструкция, которая задает логику построения текста:

```
<#
 var greeting = "Hello, World!";
#>
```

- Блок выражения – это блок, заключенный в `<#= #>`. Всё, что внутри этого блока, будет приведено к строке и добавлено в текст шаблона:

```
System.Console.WriteLine("<#= greeting #>");
```

- Блок функций – это блок, заключенный в `<#+ #>`. Все функции, объявленные в этом блоке, могут быть вызваны в шаблоне. Кроме того, сами функции могут содержать текст шаблона.

- Директива `<#@ template #>` – позволяет задать характеристики класса преобразования из шаблона:

- `<#@ template language="C#">` – задает язык класса.
- `<#@ template debug="true">` – позволяет отладить генерацию шаблона.

- <#@ template inherits="MyTextTransformation"> – указывает, какой класс должен быть использован в качестве базового для класса генерации в процедуре генерации файла.
- Директива <#@ output #> – задает расширение для генерируемого файла:  
`<#@ output extension=".cs" #>`
- Директива <#@ import #> – добавляет использование в процедуре исполнения заданных namespace. То же самое что и using добавить (но не в результат, а при выполнении генерации текста):  
`<#@ import namespace="System.Collections" #>`
- Директива <#@ assembly #> – добавляет объявление сборки. То же самое, что и в VisualStudio добавить Reference:  
`<#@ Assembly Name="System.Core" #>`
- Директива <#@ include #> – добавляет некий другой шаблон в месте объявления. Это как @Html.Partial():  
`<#@ include file="Included.tt" #>`
- Директива <#@ parameter #> - добавляет параметр при формировании шаблона. Но передача его происходит настолько сложно, что пример я приводить не буду. [Ссылка](#)
- По остальному – смотрите по [ссылке](#).

В общем, этих знаний и знаний по Reflection вполне хватит, чтобы сгенерировать нужные нам файлы, но перейдем к проекту MvcScaffolding.

## MVCscaffolding

Установим T4Scaffolding:

```
PM> Install-Package T4Scaffolding
```

Создадим папку CodeTemplates/Scaffolders/IRepository в LessonProject.Model в ней добавим файлы IRepository.ps1 (LessonProject.Model/CodeTemplates/Scaffolders/IRepository/IRepository.ps1):

```
[T4Scaffolding.Scaffolder(Description = "Create IRepository
interface")] [CmdletBinding()]
param(
 [parameter(Mandatory = $true, ValueFromPipelineByPropertyName =
$true)][string]$ModelType,
 [string]$Project,
 [string]$CodeLanguage,
 [string[]]$TemplateFolders,
 [switch]$Force = $false
)

$foundModelType = Get-ProjectType $ModelType -Project $Project -BlockUi
if (!$foundModelType) { return }

Find the IRepository interface, or create it via a template if not already
present
$foundIRepositoryType = Get-ProjectType IRepository -Project $Project -
AllowMultiple
if (!$foundIRepositoryType)
{
 #Create IRepository
 $outputPath = " IRepository"
 $defaultNamespace = (Get-Project
$Project).Properties.Item("DefaultNamespace").Value

Add-ProjectItemViaTemplate $outputPath -Template IRepositoryTemplate `-
-Model @{ Namespace = $defaultNamespace } `-
-SuccessMessage "Added IRepository at {0}" `
```

```

 -TemplateFolders $TemplateFolders -Project $Project -CodeLanguage
$CodeLanguage -Force:$Force

 $found IRepositoryType = Get-ProjectType IRepository -Project $Project
}

Add a new property on the DbContext class
if ($found IRepositoryType) {
 $propertyName = $found ModelType.Name
 $propertyNames = Get-PluralizedWord $propertyName
 # This *is* a DbContext, so we can freely add a new property if there
 isn't already one for this model
 Add-ClassMemberViaTemplate -Name $propertyName -CodeClass
$found IRepositoryType -Template IRepositoryItemTemplate -Model @{
 EntityType = $found ModelType;
 EntityTypeNamePluralized = $propertyNames;
 } -SuccessMessage "Added '$propertyName' to interface
'$($found IRepositoryType.FullName)'"
 -TemplateFolders $TemplateFolders -Project $Project -CodeLanguage $CodeLanguage

}

return @{
 DbContextType = $found DbContextType
}

```

Потом IRepositoryItemTemplate.cs.t4:

```

<#@ Template Language="C#" HostSpecific="True" Inherits="DynamicTransform" #>
#region <#= ((EnvDTE.CodeType)Model.EntityType).Name #>

IQueryable<#= ((EnvDTE.CodeType)Model.EntityType).Name #><#= Model.EntityTypeNamePluralized #> { get; }

bool Create<#= ((EnvDTE.CodeType)Model.EntityType).Name #>(<#= ((EnvDTE.CodeType)Model.EntityType).Name #> instance);

bool Update<#= ((EnvDTE.CodeType)Model.EntityType).Name #>(<#= ((EnvDTE.CodeType)Model.EntityType).Name #> instance);

bool Remove<#=((EnvDTE.CodeType)Model.EntityType).Name #>(int id<#= ((EnvDTE.CodeType)Model.EntityType).Name #>);

#endregion

```

И IRepositoryTemplate.cs.t4:

```

<#@ Template Language="C#" HostSpecific="True" Inherits="DynamicTransform" #>
<#@ Output Extension="cs" #>
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

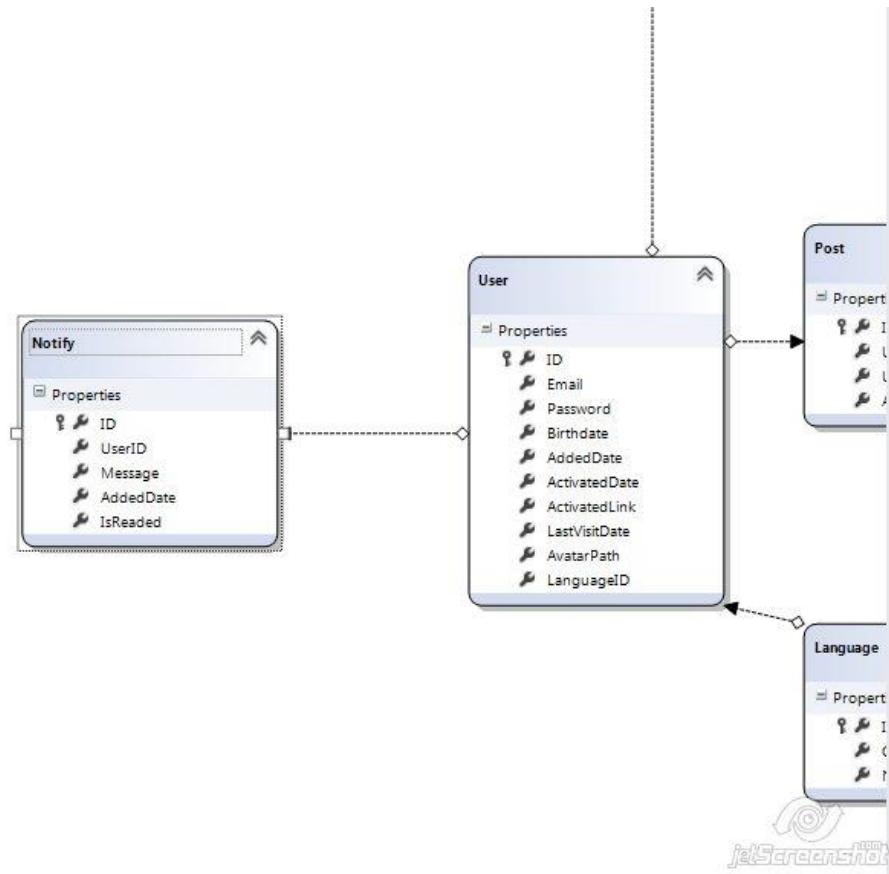
namespace <#= Model.Namespace #>
{
 public interface IRepository
 {
 IQueryable<T> GetTable<T>() where T : class;
 }
}

```

Создадим новую таблицу Notify:

Name	DataType
ID	Int autoincrement
UserID	Int (foreignKey to User)
Message	Nvarchar(140)
AddedDate	datetime
IsReaded	bit

Перенесем в DbContext (LessonProjectDb.dbml) и сохраним (ctrl-S):



В Package Manager Console пишем для проекта LessonProject.Model:

```
PM> Scaffold IRepository Notify
Added 'Notify' to interface 'LessonProject.Model.IRepository'
```

Ура! Всё работает! Просто, не правда ли? Ничего не ясно? Ок, ладно разберем IRepository.ps1 по порядку:

```
[T4Scaffolding.Scaffolder(Description = "Create IRepository
interface")] [CmdletBinding()]
param(
 [parameter(Mandatory = $true, ValueFromPipelineByPropertyName =
 $true)] [string]$ModelType,
 [string]$Project,
 [string]$CodeLanguage,
 [string[]]$TemplateFolders,
 [switch]$Force = $false
)
```

Это структура объявления кода скэффолдера. Особое внимание нужно обратить на \$ModelType – это имя класса, именно его мы и передаем в команде Scaffold IRepository **Notify**. Остальные параметры идут или по умолчанию, как Force, или по умолчанию известны, как Project, CodeLanguage.

Далее мы ищем этот класс (если мы не сохранимся, то искомый класс еще не будет записан и не будет найден):

```
$foundModelType = Get-ProjectType $ModelType -Project $Project -BlockUi
if (!$foundModelType) { return }
```

Класс найден и мы переходим к следующей части. Найти файл IRepository.cs и если его нет, то создать:

```
Find the IRepository interface, or create it via a template if not already
present
$foundIRepositoryType = Get-ProjectType IRepository -Project $Project -
AllowMultiple
if (!$foundIRepositoryType)
{
 #Create IRepository
 $outputPath = "IRepository"
 $defaultNamespace = (Get-Project
$Project).Properties.Item("DefaultNamespace").Value

 Add-ProjectItemViaTemplate $outputPath -Template IRepositoryTemplate `
 -Model @{ Namespace = $defaultNamespace } `
 -SuccessMessage "Added IRepository at {0}" `
 -TemplateFolders $TemplateFolders -Project $Project -CodeLanguage
$CodeLanguage -Force:$Force

 $foundIRepositoryType = Get-ProjectType IRepository -Project $Project
}
```

Тут как раз вызывается IRepositoryTemplate.cs.t4 при необходимости, и туда в качестве модели данных (так же, как в View) передается объект

```
-Model @{ Namespace = $defaultNamespace } `
```

A defaultNamespace был получен из свойства проекта

```
Get-Project $Project).Properties.Item("DefaultNamespace").Value
```

В шаблоне мы это используем (CodeTemplates/Scaffolders/IRepository/IRepositoryTemplate.cs.t4):

```
namespace <#= Model.Namespace #>
```

Ок, файл создан (или найден) и идем к следующему шагу. Если всё хорошо сгенерировалось (\$foundIRepositoryType), то добавим в этот класс несколько свойств по шаблону IRepositoryItemTemplate с параметрами:

```

Add a new property on the DbContext class
if ($foundIRepositoryType) {
 $propertyName = $foundModelType.Name
 $propertyNames = Get-PluralizedWord $propertyName
 # This *is* a DbContext, so we can freely add a new property if there
 isn't already one for this model
 Add-ClassMemberViaTemplate -Name $propertyName -CodeClass
 $foundIRepositoryType -Template IRepositoryItemTemplate -Model @{
 EntityType = $foundModelType;
 EntityTypeNamePluralized = $propertyNames;
 } -SuccessMessage "Added '$propertyName' to interface
'$($foundIRepositoryType.FullName)'"
 -TemplateFolders $TemplateFolders -
 Project $Project -CodeLanguage $CodeLanguage
}

}

```

Параметры:

```

-Model @{

 EntityType = $foundModelType;
 EntityTypeNamePluralized = $propertyNames;
}

```

Кстати, обратите внимание на Get-PluralizedWord и какую роль оно сыграло в созданном шаблоне:

```
IQueryable<Notify> Notifies { get; }
```

Т.е. нормально сформировал множественное число, а не просто прибавлением символа 's', как это было бы в сниппете.

Изучим еще эти [T4Scaffolding cmdlet](#):

- **Add-ClassMember** – добавляет кусочек кода в существующий класс:

```
$class = Get-ProjectType HomeController
Add-ClassMember $class "public string MyNewStringField;"
```

- **Add-ClassMemberViaTemplate** – добавляем блок кода через шаблон T4:

```
$class = Get-ProjectType HomeController
Add-ClassMemberViaTemplate -CodeClass $class -Template "YourTemplateName" -
Model @{} SomeParam = "SomeValue"; AnotherParam = $false } -TemplateFolders
$TemplateFolders
```

- **Add-ProjectItemViaTemplate** – добавляем файл (project item) посредством шаблона:

```
Add-ProjectItemViaTemplate -OutputPath "Some\Folder\MyFile" -Template
"YourTemplateName" -Model @{} SomeParam = "SomeValue"; AnotherParam = $false }
-TemplateFolders $TemplateFolders
```

- **Get-PluralizedWord / Get-SingularizedWord** – получаем множественное/единственное число слова:

```
$result = Get-PluralizedWord Person # Sets $result to "People"
$result = Get-SingularizedWord People # Sets $result to "Person"
```

- **Get-PrimaryKey** – получить первичный ключ из модели данных:

```
$pk = Get-PrimaryKey StockItem
```

- **Get-ProjectFolder** – получить класс проектной папки

```
$folder = Get-ProjectFolder "Views\Shared"
Write-Host "The shared views folder contains $($folder.Count) items"
```

- **Get-ProjectItem** – получить файл

```
$file = Get-ProjectItem "Controllers\HomeController.cs"
$file.Open()
$file.Activate()
```

- **Get-ProjectLanguage** – для C# проектов – возвращает cs, для VB проектов возвращает VB

```
$defaultProjectLanguage = Get-ProjectLanguage
$otherProjectLanguage = Get-ProjectLanguage -Project SomeOtherProjectName
```

- **Get-ProjectType** – получит модель класса (EnvDTE.CodeType)

```
$class = Get-ProjectType HomeController
Add-ClassMember $class "public string MyNewStringField;"
```

- **Get-RelatedEntities** – находит все связанные один-ко-многим объекты класса

```
Get-RelatedEntities Product
```

- **Set-IsCheckedOut** – это для работы с source-control. Т.е. если надо checkOut файл какой-то, то это можно выполнить этой командой.

```
Set-IsCheckedOut "Controllers\HomeController.cs"
```

Ну что? Чувствуете мощь, которая уже заменит копипастинг в ваших проектах? Но(!) Учтите, что все эти команды были реализованы тоже людьми. И, например, получение первичного ключа будет происходить только, если поле называется ID, а если оно называется PervichniyKlyuch – то, скорее всего, это не сработает. Также сильно не стоит уповать на переводы. Это скаффолдинг, т.е. черновое создание проекта, а не тончайшая настройка. Суть скаффолдинга – это создать, запустить и пойти пить чай, пока программа в автоматическом режиме сделает за вас самую противную механическую рутину.

## Снова шаблоны, EnvDTE.CodeType

Вернемся к шаблонам и изучим то, что такое EnvDTE.CodeType.

CodeType – это интерфейс, к которому может быть приведена информация об классе, полученная через Get-ProjectType.

Что мы знаем про этот интерфейс. Например, про свойства:

- Access – какой это тип, публичный, приватный и т.д..
- Attributes – коллекции атрибутов, связанные с этим типом.
- Bases – коллекции классов, из которых этот элемент происходит.
- Children – возвращает коллекцию объектов, содержащихся в этом CodeType.

- Comment – комментарий относящийся к классу. (Можно сделать автоматическую документацию).
- DerivedTypes – Наследуемые типы. Это свойство не поддерживается в Visual C#.
- DocComment – Документальный комментарий или атрибут, который выполняет эту роль.
- DTE – возвращает главный объект расширения
- EndPoint – строка в файле, с которой начинается описание этого класса.
- FullName – полное имя, типа System.Int32
- InfoLocation – возвращает возможности объектной модели.
- IsCodeType – можно ли CodeType получить из этого объекта.
- IsDerivedFrom – возвращает CodeType базового объекта.
- Kind – свойства типа объекта.
- Language – на каком языке это написано.
- Members – члены объекта. Вот это очень полезная функция.
- Name – имя объекта.
- Namespace – пространство имен объекта.
- Parent – непосредственный родитель объекта.
- ProjectItem – файл объекта.
- StartPoint – строка, в которой началось описание объекта.

Есть еще методы, но мы их не используем.

Кстати, обратите внимание на EnvDTEExtensions.cs в T4Scaffolding (исходники его можно скачать отсюда: <http://mvcscaffolding.codeplex.com/SourceControl/changeset/view/7cd57d172314>), какие еще вспомогательные классы вам доступны.

Фух! Ну что, попробуем разложить всё по полочкам, раскрошить программно любой код, а потом объяснить компьютеру, как мы пишем программы, и идти гонять чаи.

Создадим новый проект: LessonProject.Scaffolding, и возьмем тут пару классов из первых уроков, с мечом и воином.

IWeapon.cs:

```
public interface IWeapon
{
 void Kill();
}
```

Bazuka.cs:

```
public class Bazuka : IWeapon
{
 public void Kill()
 {
 Console.WriteLine("BIG BADABUM!");
 }
}
```

Sword.cs:

```
public class Sword : IWeapon
{
 public void Kill()
 {
 Console.WriteLine("Chuk-chuck");
 }
}
```

Warrior.cs:

```
/// <summary>
/// This is LEGENDARY WARRIOR!
/// </summary>
public class Warrior
{
 readonly IWeapon Weapon;

 public Warrior(IWeapon weapon)
 {
 this.Weapon = weapon;
 }

 public void Kill()
 {
 Weapon.Kill();
 }
}
```

Установим T4Scaffolding:

```
Install-Package T4Scaffolding
```

Создадим простейший PowerShell (/CodeTemplates/Scaffolders/Details/Details.ps1):

```
[T4Scaffolding.Scaffolder(Description = "Print Details for
class")]
[CmdletBinding()]
param(
 [parameter(Mandatory = $true, ValueFromPipelineByPropertyName =
$true)][string]$ModelType,
 [string]$Project,
 [string]$CodeLanguage,
 [string[]]$TemplateFolders,
 [switch]$Force = $false
)
$foundModelType = Get-ProjectType $ModelType -Project $Project -BlockUi
if (!$foundModelType) { return }

$outputPath = Join-Path "Details" $ModelType

Add-ProjectItemViaTemplate $outputPath -Template Details ^
 -Model @{ ModelType = $foundModelType } ^
 -SuccessMessage "Yippee-ki-yay" ^
 -TemplateFolders $TemplateFolders -Project $Project -CodeLanguage
$CodeLanguage -Force:$Force
```

Заданный тип данных передаем в Details.t4 (/CodeTemplates/Scaffolders/Details/Details.cs.t4):

```
<#@ template language="C#" HostSpecific="True" Inherits="DynamicTransform" debug="true"
#>
<#@ assembly name="System.Data.Entity" #>
<#@ import namespace="System.Linq" #>
<#@ import namespace="EnvDTE" #>
<#@ Output Extension="txt" #>
<#
 var modelType = (EnvDTE.CodeType)Model.ModelType;
#>

FullName : <#= modelType.FullName #>
Name : <#= modelType.Kind #> <#= modelType.Name #>
Access : <#= modelType.Access #>
Attributes :
<# foreach(var codeElement in modelType.Attributes) {
 var attr = (EnvDTE.CodeAttribute)codeElement;
#>
 <#= attr.Name #>
<# } #>

Bases :
<# foreach(var codeElement in modelType.Bases) {
 var @base = (EnvDTE.CodeType)codeElement;
#>
 <#= @base.Name #>
<# } #>

Comment : <#= modelType.Comment #>
DocComment : <#= modelType.DocComment #>

StartPoint : Line: <#= ((EnvDTE.TextPoint)modelType.StartPoint).Line #>
EndPoint : Line : <#= ((EnvDTE.TextPoint)modelType.EndPoint).Line #>

Members :
<# foreach(var codeElement in modelType.Members) {
 var member = (EnvDTE.CodeElement)codeElement;
#>
 <#= member.Kind #> <#= member.Name #>
<# } #>
```

Выведем для Warrior.cs

```
PM> Scaffold Details Warrior -Force:$true
Yippee-ki-yay
```

- Имя\ полное имя
- Тип модели
- Доступ
- Атрибуты модели
- Базовый класс
- Комментарий
- Комментарий для документации
- Стока файла, с которой началось объявление, и строка, на которой закончилось объявление
- Имена членов класса с типом класса

Мы можем исследовать классы, использовать направляющие атрибуты и на основе этого создавать промежуточные классы, т.е. автоматизировать процессы, которые слишком рутинны

для ручной работы. В то же время у нас появляется преимущество, ведь автоматически сгенерированный код содержит меньше ошибок, так как часть человеческого фактора мы убираем.

## Описание скраффолдеров

Итак, я не буду тут приводить код всех используемых мною скраффолдеров, только опишу здесь их параметры для запуска. Но прежде расскажу о ManageAttribute. Эти атрибуты присваиваются тем полям, которые мы хотим в дальнейшем использовать как маркеры для генерации определенного кода. Например, атрибут LangColumn – это атрибут, указывающий на то, что данное поле является «языковым». Тем самым мы можем генерировать ModelView и с учетом их тоже.

- IRepository (Model). Мы уже с ним знакомы, он создает интерфейс IRepository и вносит CRUD-методы для заданного типа:

```
Scaffold IRepository ModelName
```

- Proxy (Model). Создает Proxy partial class. Если задан параметр Lang:\$true, то скраффолдер ищет языковую модель данных ModelName+”Lang” и добавляет языковые поля в partial class.

```
Scaffold Proxy ModelName -Lang:$true
```

- SqlRepository (Model). Создает реализацию CRUD-методов класса ModelName. Также имеет параметр Lang для создания приватного метода, работающего с языковыми полями.

```
Scaffold SqlRepository ModelName -Lang:$true
```

- ProviderRepository (Model). Запускает три вышеперечисленных скраффолдинга за один раз.

```
Scaffold ProviderRepository ModelName -Lang:$true
```

- Model (Web). Создает модель ModelNameView в Models/ViewModels и создает обработчик Automapper в Mappers/MappersCollection.cs. После этого во View-классе необходимо прописать управляющие атрибуты для создания контроллера и Index/Edit view:
  - ShowIndex – это поле будет отображено в таблице Index
  - PrimaryField – поле ID
  - CheckBox – для этого поля будет создан элемент ввода CheckBox
  - DropDownField – для этого поля будет создан элемент ввода DropDownField
  - HiddenField – скрытое поле
  - HtmlTextField – элемент ввода textarea, помеченный классом htmltext
  - RadioField – поле с радио-кнопками (на практике практически не использовалось)
  - TextAreaField – элемент ввода textarea
  - TextBoxField – обычное текстовое поле ввода

```
Scaffold Model ModelName
```

- SelectReference (Web). Создает во view-классе зависимость один-ко-многим, т.е. элемент выбора. Например, если создается город (City) с принадлежностью к штату (State), то при создании города указывается выпадающий список штатов, задающий значение StateID. Для этого необходимо использовать SelectReference, который добавит необходимый код к CityView: Scaffold SelectReference City State

- Controller (Web). Создает контроллер для данного ModelName типа. Дополнительно генерирует и Index\Edit View. Параметрами являются:
  - Area (по умолчанию – нет), создает контроллер в определенном Area
  - Paging (по умолчанию – false), использует или не использует постраничный вывод
  - Lang (по умолчанию – false), генерирует код с использованием языковых настроек

```
Scaffold Controller ModelName -Area:Admin -Paging:$true -Lang:$true
```

- IndexView\EditView (Web). Создает просмотр списка или редактирование объекта. Дополнительные параметры - те же, что и у Controller:
  - Area (по умолчанию – нет), создает контроллер в определенном Area
  - Paging (по умолчанию – false), использует или не использует постраничный вывод
  - Lang (по умолчанию – false), генерирует код с использованием языковых настроек

```
Scaffold IndexView ModelName -Area:Admin -Paging:$true -Lang:$true
```

```
Scaffold EditView ModelName -Area:Admin -Paging:$true -Lang:$true
```

## Итог

Скаффолдинг – это не панацея, но это хороший инструмент, с помощью которого можно быстро создать необходимый код. Написанные классы позволяют быстро начать управлять содержимым базы данных, и избавляют от множества ручной рутинной работы.

Действия при создании новой таблицы (объекта) будут следующие:

- Описать таблицу(ы) с полями в БД
- Перенести ее в DBContext.dbml
- Запустить ProviderRepository для необходимых таблиц, убрать лишние методы
- Запустить Model для необходимых таблиц
- Прописать управляющие атрибуты во view-классах, убрать лишние поля
- Создать контроллеры в админке
- Допилить напильником сложные поля (например, загрузку файлов)

Всё это выполняется сразу на несколько таблиц, если это старт проекта или большой патч. У меня иногда генерировалось до 20-30 таблиц, это заняло около 5 минут, но без этого пришлось бы провозиться целый день.

Посмотрите на реализацию скаффолдингов, вы сможете больше понять внутренние особенности программы и ее структуру.

## Урок Е. Тестирование

Цель: Научиться создавать тесты для кода. NUnit. Принцип применения TDD. Mock. Юнит-тесты. Интегрированное тестирование. Генерация данных.

### Тестирование, принцип TDD, юнит-тестирование и прочее.

Тестирование для меня лично – это тема многих размышлений. Нужны или не нужны тесты? Но никто не будет спорить, что для написания тестов нужны ресурсы.

Рассмотрим два случая:

1. Мы делаем сайт, показываем заказчику, он высылает список неточностей и дополнительных пожеланий, мы их бодро правим и сайт отдаём заказчику, т.е. выкладываем на его сервер. На его сервер никто не ходит, заказчик понимает, что чуда не произошло и перестает платить за хостинг/домен. Сайт умирает. Нужны ли там тесты?
2. Мы делаем сайт, показываем заказчику, он высылает список правок, мы их бодро правим, запускаем сайт. Через полгода на сайте 300 уников в день и эта цифра растёт изо дня в день. Заказчик постоянно просит новые фичи, старый код начинает разрастаться, и со временем его всё сложнее поддерживать.

Видите ли, тут дилемма такова, что результат запуска сайта непредсказуемый. У меня было и так, что созданный на коленке сайт запустился весьма бодро, а бывало, что крутую работу заказчик оплатил, но даже не принял. Итак, тактика поведения может быть такой:

- **Писать тесты всегда.** Мы крутая компания, мы покрываем 90% кода всяческими тестами, и нам реально всё равно, что мы тратим на это в 100500 раз больше времени/денег, ведь результат полностью предсказуем и мы вообще красавцы.
- **Не писать тесты никогда.** Мы крутая компания, мы настолько идеально работаем, что можем в уме пересобрать весь проект, и если наш код компилируется, это значит, что он полностью рабочий. Если что-то не работает, то вероятно это хостинг, или ошибка в браузере. или фича такая.
- **Писать тесты, но не всегда.** Тут мы должны понять, что каким бы ни был сайт или проект, то он состоит из функционала. А это значит, что пользователям должны быть предоставлены всяческие возможности, и возможности важные, я бы даже сказал - критические, как-то зарегистрироваться на сайте, сделать заказ, добавить новость или комментарий. Неприятно, когда хочешь, а не можешь зарегистрироваться, ведь сайт-то нужный.

Для чего используются тесты? Это как принцип ведения двойной записи в бухгалтерии. Каждое действие, каждый функционал проверяется не только работоспособностью сайта, но и еще как минимум одним тестом. При изменении кода юнит-тесты указывают, что именно пошло не так и красным подсвечивают места, где произошло нарушение. Но так ли это?

Рассмотрим принцип TDD:

1. Прочитать задание и написать тест, который заваливается
2. Написать любой код, который позволяет проходить данный тест и остальные тесты
3. Сделать рефакторинг, т.е. убрать повторяющийся код, если надо, но чтобы все тесты проходили

Например, было дано следующее исправление:

Мы решили добавить в блог поле тегов. Так как у нас уже существует много записей в блоге, то это поле решили сделать необязательным. Так как уже есть существующий код, то скаффолдингом не пользовались. Вручную проверили создание записи – всё ок. Прогнали тесты – всё ок. Но забыли добавить изменение поля в `UpdatePost (cache.Tags = instance.Tags;)`. При изменении старой записи мы добавляем теги, которые собственно не сохраняются. При этом тесты прошли на ура. Жизнь - боль!

Что ж, как видно, мы нарушили основной принцип TDD – вначале пиши тест, который заваливается, а уже потом пиши код, который его обрабатывает. Но(!) тут есть и вторая хитрость – мы написали тест, который проверяет создание записи блога с тегом. Конечно, сразу же у нас это не скомпилировалось (т.е. тест не прошел), но мы добавили в `ModelView` что-то типа `throw new NotImplementedException()`. Всё скомпилировалось, тест горит красным, мы добавляем это поле с тегом, убирая исключение, тест проходит. Все остальные тесты тоже проходят. Принципы соблюdenы, а ошибки остались.

Что я могу сказать, на каждый принцип найдется ситуация, где он не сработает. Т.е. нет такого – отключили мозги и погнали. Одно можно сказать точно, и это главный вывод из этих рассуждений:

## **тесты должны писаться быстро**

Так какие же задачи мы решаем в основном на сайте:

- Добавление информации
- Проверка информации
- Изменение информации
- Удаление информации
- Проверка прав на действие
- Выдача информации

Это основные действия. Как, например, проходит регистрация:

- Показываем поля для заполнения
- При нажатии на «Зарегистрироваться» проверяем данные
- Если всё удачно, то выдаем страничку «Молодец», если же не всё хорошо, то выдаем предупреждение и позволяем исправить оплошность
- Если всё хорошо, то в БД у нас появляется запись
- А еще мы письмо с активацией отправляем

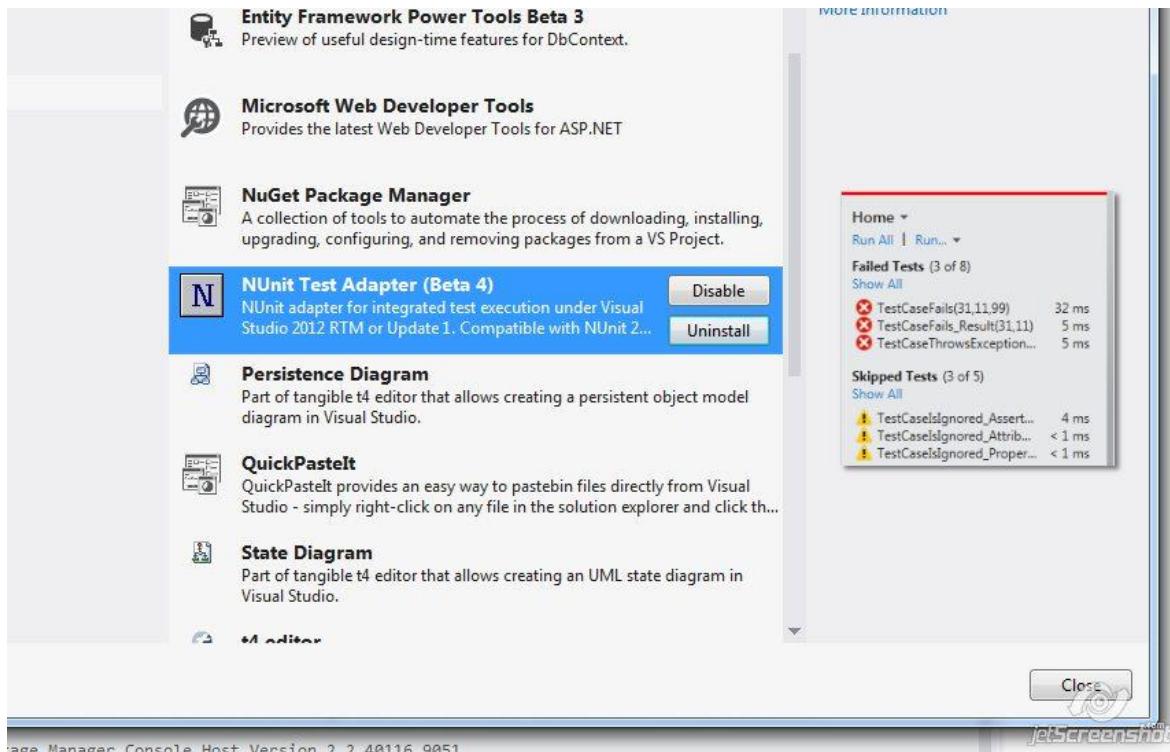
Создадим для всего этого юнит-тесты:

- Что мы показываем ему поля для заполнения (т.е. передаем пустой объект класса `RegisterUserView`)
- Что у нас стоят атрибуты и всё такое, проверяем, что действительно ли мы проверяем, что можно записать в БД
- Что выдаем именно «Молодец» страницу
- Что появляется запись, что было две записи, а стало три записи
- Что пытаемся что-то отправить, находим шаблон и вызываем `MailNotify`.

Приступим, пожалуй.

## Установить NUnit

Идем по ссылке <http://sourceforge.net/projects/nunit/> и устанавливаем NUnit. Так же в VS устанавливаем NUnit Test Adapter (ну чтобы запускать тесты прямо в VS).



Создадим папочку типа Solution Folder **Test** и в нее добавим проект LessonProject.UnitTesting и установим там NUnit:

```
Install-Package NUnit
```

Создадим класс UserControllerTest в (/Test/Default/UserController.cs):

```
[TestFixture]
public class UserControllerTest
{}
```

Итак, принцип написания наименования методов тестов Method\_Scenario\_ExpectedBehavior:

- Method – метод [или свойство], который тестируем
- Scenario – сценарий, который мы тестируем
- ExpectedBehavior – ожидаемое поведение

Например, проверяем первое, что возвращаем View с классом UserView для регистрации:

```
public void Register_GetView_ItsOkViewModelIsUserView()
{
 Console.WriteLine("=====INIT=====");
 var controller = new UserController();
 Console.WriteLine("=====ACT=====");
 var result = controller.Register();
 Console.WriteLine("=====ASSERT=====");
```

```

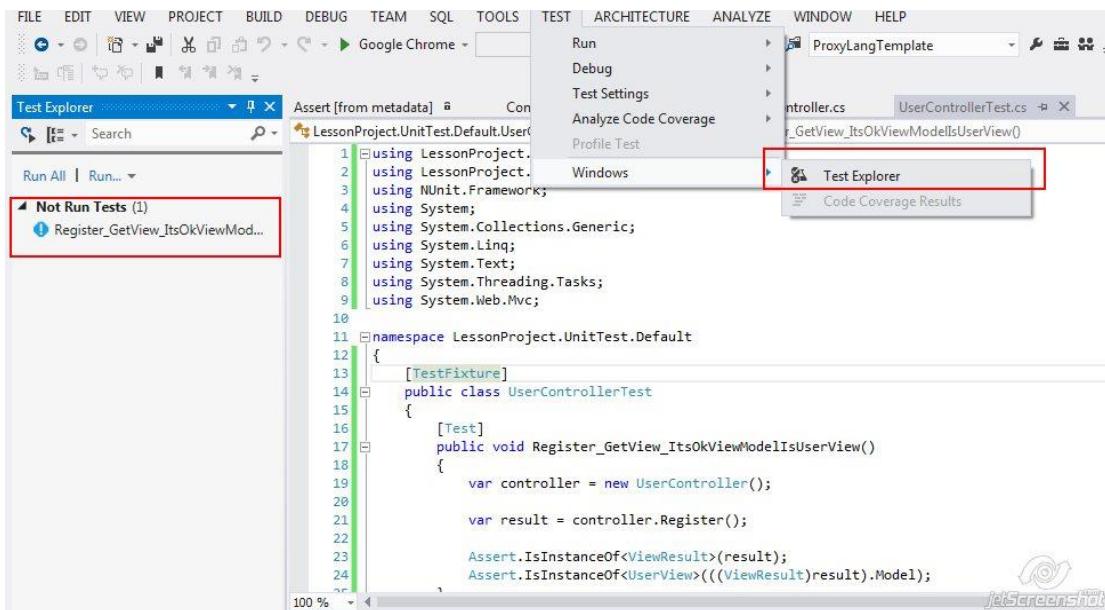
 Assert.IsInstanceOf<ViewResult>(result);
 Assert.IsInstanceOf<UserView>(((ViewResult)result).Model);
 }
}

```

Итак, все тесты делятся на 3 части Init->Act->Assert:

- Init – инициализация, мы получаем наш UserController
- Act – действие, мы запускаем наш controller.Register
- Assert – проверка, что всё действительно так.

Откроем вкладку Test Explorer:



Если адаптер NUnit правильно был установлен, то мы увидим наш тест-метод.

Запускаем. Тест пройден, можно идти открывать шампанское. Стооп. Это лишь самая легкая часть, а как быть с той частью, где мы что-то сохраняем. В данном случае мы не имеем БД, наш Repository – null, ноль, ничего.

Изучим теперь класс и методы для инициализации ([документация](#)):

- SetUpFixture – класс, помеченный этим атрибутом, означает, что в нем есть методы, которые проводят инициализацию перед тестами и зачистку после тестов. Это относится к одному и тому же пространству имен.
  - Setup – метод, помеченный этим атрибутом, вызывается до выполнения всех тестовых методов. Если находится в классе с атрибутом TestFixture, то вызывается перед выполнением методов только этого класса.
  - TearDown – метод, помеченный этим атрибутом, вызывается после выполнения всех тестов. Если находится в классе с атрибутом TestFixture, то вызывается после выполнения всех методов.

Создадим класс UnitTestSetupFixture.cs (/Setup/UnitTestSetupFixture.cs):

```

[SetUpFixture]
public class UnitTestSetupFixture
{
 [SetUp]
 public void Setup()
 {
 }
}

```

```

 {
 Console.WriteLine("=====");
 Console.WriteLine("====START====");
 Console.WriteLine("=====");
 }

 [TearDown]
 public void TearDown()
 {
 Console.WriteLine("=====");
 Console.WriteLine("====BYE!====");
 Console.WriteLine("=====");
 }
}

```

Запустим и получим:

```

=====
=====START=====
=====
=====INIT=====
=====ACT=====
=====ASSERT=====

=====
=====BYE!=====
=====

```

## Mock

Итак, Mock – это объект-пародия. Т.е. например, не база данных, а что-то похожее на базу данных. Мираж, в общем-то. Есть еще Stub – это заглушка. Пример метода заглушки:

```

public int GetRandom()
{
 return 4;
}

```

Но мы будем использовать Mock:

```
Install-Package Moq
```

Определим, какое окружение есть у нас, чтобы мы проинициализировали для него Mock-объекты. В принципе, это всё, что мы некогда вынесли в Ninject Kernel:

- IRepository
- IConfig
- IMapper
- IAuthentication

И тут я сделаю небольшое замечание. Мы не можем вынести Config в объекты-миражи. Не в плане, что это совсем невозможно, а в плане – что это плохая затея. Например, мы изменили шаблон письма так, что string.Format() выдает ошибку FormatException. А в тесте всё хорошо, тест отлично проходит. И за что он после этого отвечает? Ни за что. Так что файл конфигурации надо использовать оригинальный. Оставим это на потом.

По поводу, IMapper – в этом нет необходимости, мы совершенно спокойно можем использовать и CommonMapper.

Но для начала проинициализируем IKernel для работы в тестовом режиме. В App\_Start/NinjectWebCommon.cs мы в методе RegisterServices указываем, как должны быть реализованы интерфейсы, и вызываем это в bootstrapper.Initialize(CreateKernel). В дальнейшем мы обращаемся по поводу получения сервиса через DependencyResolver.GetService(). Так что создадим NinjectDependencyResolver (/Tools/NinjectDependencyResolver.cs):

```
public class NinjectDependencyResolver : IDependencyResolver
{
 private readonly IKernel _kernel;

 public NinjectDependencyResolver(IKernel kernel)
 {
 _kernel = kernel;
 }

 public object GetService(Type serviceType)
 {
 return _kernel.TryGet(serviceType);
 }

 public IEnumerable<object> GetServices(Type serviceType)
 {
 try
 {
 return _kernel.GetAll(serviceType);
 }
 catch (Exception)
 {
 return new List<object>();
 }
 }
}
```

Добавим в SetUp метод (/Setup/UnitTestSetupFixture.cs):

```
[SetUp]
public virtual void Setup()
{
 InitKernel();
}

protected virtual IKernel InitKernel()
{
 var kernel = new StandardKernel();
 DependencyResolver.SetResolver(new NinjectDependencyResolver(kernel));
 InitRepository(kernel); //намотаем
 return kernel;
}
```

Создадим MockRepository:

(/Mock/Repository/MockRepository.cs):

```
public partial class MockRepository : Mock< IRepository>
{
 public MockRepository(MockBehavior mockBehavior = MockBehavior.Strict)
 : base(mockBehavior)
 {
 GenerateRoles();
 }
}
```

```

 GenerateLanguages();
 GenerateUsers();

 }

}

(/Mock/Repository/Entity/Language.cs)

namespace LessonProject.UnitTesting.Mock
{
 public partial class MockRepository
 {
 public List<Language> Languages { get; set; }

 public void GenerateLanguages()
 {
 Languages = new List<Language>();
 Languages.Add(new Language()
 {
 ID = 1,
 Code = "en",
 Name = "English"
 });
 Languages.Add(new Language()
 {
 ID = 2,
 Code = "ru",
 Name = "Русский"
 });
 this.Setup(p => p.Languages).Returns(Languages.AsQueryable());
 }
 }
}

```

(/Mock/Repository/Entity/Role.cs)

```

public partial class MockRepository
{
 public List<Role> Roles { get; set; }

 public void GenerateRoles()
 {
 Roles = new List<Role>();
 Roles.Add(new Role()
 {
 ID = 1,
 Code = "admin",
 Name = "Administrator"
 });

 this.Setup(p => p.Roles).Returns(Roles.AsQueryable());
 }
}

```

(/Mock/Repository/Entity/User.cs)

```

public partial class MockRepository
{
 public List<User> Users { get; set; }

 public void GenerateUsers()

```

```

{
 Users = new List<User>();

 var admin = new User()
 {
 ID = 1,
 ActivatedDate = DateTime.Now,
 ActivatedLink = "",
 Email = "admin",
 FirstName = "",
 LastName = "",
 Password = "password",
 LastVisitDate = DateTime.Now,
 };

 var role = Roles.First(p => p.Code == "admin");
 var userRole = new UserRole()
 {
 User = admin,
 UserID = admin.ID,
 Role = role,
 RoleID = role.ID
 };

 admin.UserRoles =
 new EntitySet<UserRole>() {
 userRole
 };
 Users.Add(admin);

 Users.Add(new User())
 {
 ID = 2,
 ActivatedDate = DateTime.Now,
 ActivatedLink = "",
 Email = "chernikov@gmail.com",
 FirstName = "Andrey",
 LastName = "Chernikov",
 Password = "password2",
 LastVisitDate = DateTime.Now
 });

 this.Setup(p => p.Users).Returns(Users.AsQueryable());
 this.Setup(p => p.GetUser(It.IsAny<string>())).>Returns((string email) =>
 Users.FirstOrDefault(p => string.Compare(p.Email, email, 0) == 0));
 this.Setup(p => p.Login(It.IsAny<string>(),
 It.IsAny<string>())).>Returns((string email, string password) =>
 Users.FirstOrDefault(p => string.Compare(p.Email, email, 0) == 0));
}
}

```

Рассмотрим, как работает Mock. У него есть такой хороший метод, как Setup (опять?! сплошной сетап!), который работает таким образом:

```
this.Setup(что у нас запрашивают).Returns(что мы отвечаем на это);
```

Например:

```
this.Setup(p => p.WillYou()).>Returns(true);
```

Рассмотрим подробнее, какие еще могут быть варианты:

- Методы

```
var mock = new Mock<IFoo>();
mock.Setup(foo => foo.DoSomething("ping")).Returns(true);
```

- параметр out

```
var outString = "ack";
mock.Setup(foo => foo.TryParse("ping", out outString)).Returns(true);
```

- ссылочный параметр

```
var instance = new Bar();
mock.Setup(foo => foo.Submit(ref instance)).Returns(true);

- зависимость от входного параметра и возвращаемого значения (можно и несколько параметров)

```

```
mock.Setup(x => x.DoSomething(It.IsAny<string>()))
 .Returns((string s) => s.ToLower());
```

- кидаем исключение

```
mock.Setup(foo => foo.DoSomething("reset")).Throws<InvalidOperationException>();
mock.Setup(foo => foo.DoSomething("")).Throws(new ArgumentException("command"));
```

- возвращает различные значения для (???) и использование Callback

```
var mock = new Mock<IFoo>();
var calls = 0;
mock.Setup(foo => foo.GetCountThing())
 .Returns(() => calls)
 .Callback(() => calls++);
```

- Соответствие на аргументы

- любое значение

```
mock.Setup(foo => foo.DoSomething(It.IsAny<string>())).Returns(true);
```

- условие через Func<bool, T>

```
mock.Setup(foo => foo.Add(It.Is<int>(i => i % 2 == 0))).Returns(true);
```

- нахождение в диапазоне

```
mock.Setup(foo => foo.Add(It.IsInRange<int>(0, 10, Range.Inclusive))).Returns(true);
```

- Regex выражение

```
mock.Setup(x => x.DoSomething(It.IsRegex("[a-d]+",
 RegexOptions.IgnoreCase))).Returns("foo");
```

- Свойства

- Любое свойство

```
mock.Setup(foo => foo.Name).Returns("bar");
```

- Любой иерархии свойство

```
mock.Setup(foo => foo.Bar.Baz.Name).Returns("baz");
```

- Обратные вызовы (callback)

- Без параметров

```
mock.Setup(foo => foo.Execute("ping"))
 .Returns(true)
 .Callback(() => calls++);
```

- С параметром

```
mock.Setup(foo => foo.Execute(It.IsAny<string>()))
 .Returns(true)
 .Callback<string>((string s) => calls.Add(s));
```

- С параметром, немного другой синтаксис

```
mock.Setup(foo => foo.Execute(It.IsAny<string>()))
 .Returns(true)
 .Callback<string>(s => calls.Add(s));
```

- Несколько параметров

```
mock.Setup(foo => foo.Execute(It.IsAny<int>(), It.IsAny<string>()))
 .Returns(true)
 .Callback<int, string>((i, s) => calls.Add(s));
```

- До и после вызова

```
mock.Setup(foo => foo.Execute("ping"))
 .Callback(() => Console.WriteLine("Before returns"))
 .Returns(true)
 .Callback(() => Console.WriteLine("After returns"));
```

- Проверка (Mock объект сохраняет количество обращений к своим параметрам, тем самым мы также можем проверить правильно ли был исполнен код)

- Обычная проверка, что был вызван метод Execute с параметром "ping"

```
mock.Verify(foo => foo.Execute("ping"));
```

- // С добавлением собственного сообщения об ошибке

```
mock.Verify(foo => foo.Execute("ping"), "When doing operation X, the service
should be pinged always");
```

- Не должен был быть вызван ни разу

```
mock.Verify(foo => foo.Execute("ping"), Times.Never());
```

- Хотя бы раз должен был быть вызван

```
mock.Verify(foo => foo.Execute("ping"), Times.AtLeastOnce());
```

```
mock.VerifyGet(foo => foo.Name);
```

- Должен был быть вызван именно сеттер для свойства

```
mock.VerifySet(foo => foo.Name);
```

- Должен был быть вызван сеттер со значением "foo"

```
mock.VerifySet(foo => foo.Name = "foo");
```

- Сеттер должен был быть вызван со значением в заданном диапазоне

```
mock.VerifySet(foo => foo.Value = It.IsInRange(1, 5, Range.Inclusive));
```

Хорошо, этого нам пока хватит, остальное можно будет почитать здесь:

<https://code.google.com/p/moq/wiki/QuickStart>

Возвращаемся обратно в UnitTestSetupFixture.cs (/Setup/UnitTestSetupFixture.cs) и инициализируем конфиг:

```
protected virtual void InitRepository(StandardKernel kernel)
{
 kernel.Bind<MockRepository>().To<MockRepository>().InThreadScope();
 kernel.Bind< IRepository>().ToMethod(p =>
kernel.Get<MockRepository>().Object);
}
```

Проверим какой-то наш вывод, например класс /Default/Controllers/UserController.cs:

```
[Test]
public void Index_GetPageableDataOfUsers_CountOfUsersIsTwo()
{
 //init
 var controller =
DependencyResolver.Current.GetService<Areas.Default.Controllers.UserController>();
 //act
 var result = controller.Index();

 Assert.IsNotNull(result);
 Assert.IsInstanceOf<PageableData<User>>(((ViewResult)result).Model);
 var count = ((PageableData<User>)((ViewResult)result).Model).List.Count();

 Assert.AreEqual(2, count);
}
```

В BaseController.cs (/LessonProject/Controllers/BaseController.cs) уберем атрибуты Inject у свойств Auth и Config (иначе выделенная строка не сможет проинициализовать контроллер и вернет null). Кстати о выделенной строке. Мы делаем именно такую инициализацию, чтобы все Inject-атрибутованные свойства были проинициализированы. Запускаем, и, правда, count == 2. Отлично, MockRepository работает. Вернем назад атрибуты Inject.

Кстати, тесты не запускаются обычно в дебаг-режиме, чтобы запустить Debug надо сделать так:

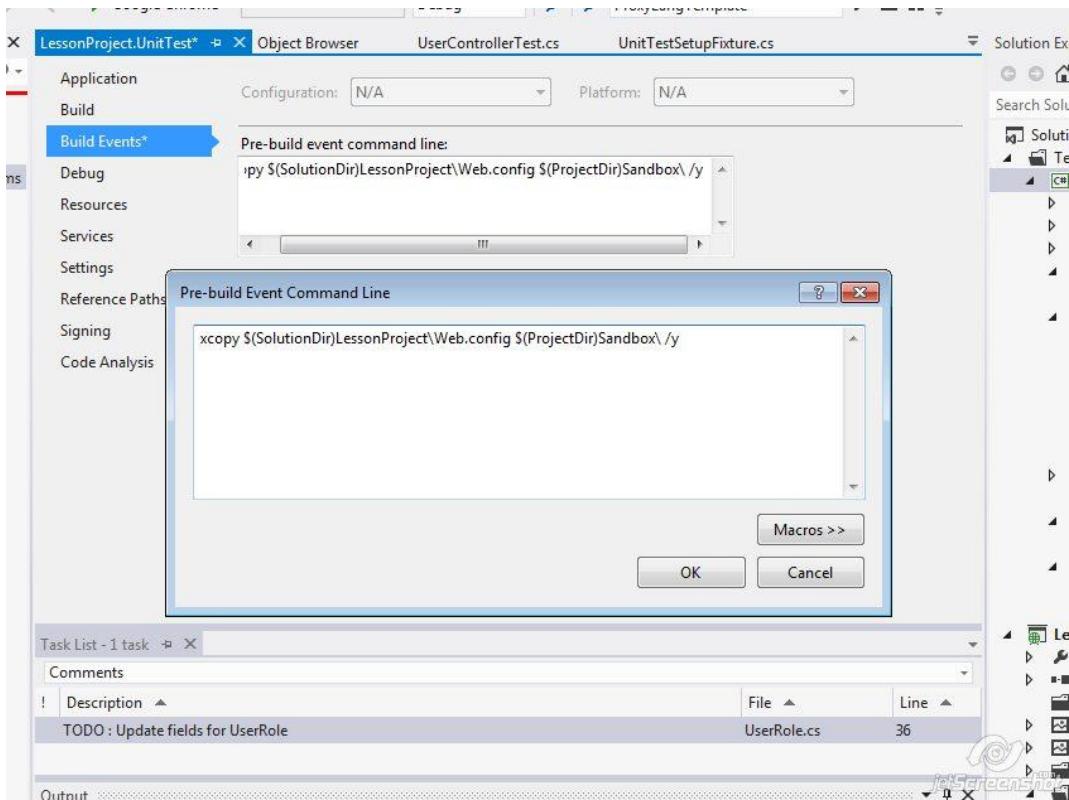
Теперь поработаем с Config. Это будет круто!

## TestConfig

Что нам нужно сделать. Нам нужно:

- Взять Web.Config с проекта LessonProject (каким-то хитрым образом)
- И на его базе создать некий класс, который будет реализовывать IConfig интерфейс
- Ну и поцепить на Ninject Kernel
- И можно использовать.

Начнем. Для того чтобы взять Web.Config – нам нужно скопировать его в свою папку. Назовем её Sandbox. Теперь скопируем, поставим на pre-build Event в Project Properties:



```
xcopy $(SolutionDir)LessonProject\Web.config $(ProjectDir)Sandbox\ /y
```

При каждом запуске билда мы копируем Web.config (и, если надо, то перезаписываем) к себе в Sandbox.

Создадим TestConfig.cs и в конструктор будем передавать наш файл (/Tools/TestConfig.cs):

```
public class TestConfig : IConfig
{
 private Configuration configuration;

 public TestConfig(string configPath)
 {
 var configFileMap = new ExeConfigurationFileMap();
 configFileMap.ExeConfigFilename = configPath;
 configuration =
ConfigurationManager.OpenMappedExeConfiguration(configFileMap,
ConfigurationUserLevel.None);
 }

 public string ConnectionStrings(string connectionString)
 {
 return
configuration.ConnectionStrings.ConnectionStrings[connectionString].ConnectionString;
 }

 public string Lang
 {
 get
 {
 return configuration.AppSettings.Settings["Lang"].Value;
 }
 }

 public bool EnableMail
```

```

 {
 get
 {
 return
bool.Parse(configuration.AppSettings.Settings["EnableMail"].Value);
 }
 }

 public IQueryable<IconSize> IconSizes
 {
 get
 {
 IconSizesConfigSection configInfo =
(IIconSizesConfigSection)configuration.GetSection("iconConfig");
 if (configInfo != null)
 {
 return
configInfo.IconSizes.OfType<IconSize>().AsQueryable<IconSize>();
 }
 return null;
 }
 }

 public IQueryable<MimeType> MimeType
 {
 get
 {
 MimeTypeConfigSection configInfo =
(MimeTypeConfigSection)configuration.GetSection("mimeConfig");
 return configInfo.MimeType.OfType<MimeType>().AsQueryable<MimeType>();
 }
 }

 public IQueryable<MailTemplate> MailTemplates
 {
 get {
 MailTemplateConfigSection configInfo =
(MailTemplateConfigSection)configuration.GetSection("mailTemplatesConfig");
 return
configInfo.MailTemplates.OfType<MailTemplate>().AsQueryable<MailTemplate>();
 }
 }

 public MailSetting MailSetting
 {
 get
 {
 return (MailSetting)configuration.GetSection("mailConfig");
 }
 }

 public SmsSetting SmsSetting
 {
 get
 {
 return (SmsSetting)configuration.GetSection("smsConfig");
 }
 }
}

```

И инициализируем в UnitTestSetupFixture.cs (/Setup/UnitTestSetupFixture.cs):

```

protected virtual void InitConfig(StandardKernel kernel)
{
 var fullPath = new FileInfo(Sandbox + "/Web.config").FullName;

```

```
 kernel.Bind<IConfig>().ToMethod(c => new TestConfig(fullPath));
 }
```

Создадим простой тест на проверку данных в конфиге:

```
[TestFixture]
public class MailTemplateTest
{
 [Test]
 public void MailTemplates_ExistRegisterTemplate_Exist()
 {
 var config = DependencyResolver.Current.GetService<IConfig>();
 var template = config.MailTemplates.FirstOrDefault(p =>
p.Name.StartsWith("Register"));
 Assert.IsNotNull(template);
 }
}
```

Запускаем, проверяем, вуаля! Переходим к реализации IAuthentication.

## Authentication

В веб-приложении, когда мы уже исполняем код в контроллере, мы уже имеем какой-то заданный контекст, окружение, сформированное http-запросом. Т.е. это и параметры, и кукисы, и данные о версии браузера, и каково разрешение экрана, и какая операционная система. В общем, это всё – HttpContext. Следует понимать, что мы при авторизации помещаем в кукисы какие-то данные, а потом достаем их и всё. Собственно, для этого мы создадим специальный интерфейс IAuthCookieProvider, который будет типа записывать кукисы

IAuthCookieProvider.cs (LessonProject/Global/Auth/IAuthCookieProvider):

```
public interface IAuthCookieProvider
{
 HttpCookie GetCookie(string cookieName);

 void SetCookie(HttpCookie cookie);
}
```

И реализуем его для HttpAuthCookieProvider.cs (/Global/Auth/HttpAuthCookieProvider.cs):

```
public class HttpContextCookieProvider : IAuthCookieProvider
{
 public HttpContextCookieProvider(HttpContext HttpContext)
 {
 this.HttpContext = HttpContext;
 }

 protected HttpContext HttpContext { get; set; }

 public HttpCookie GetCookie(string cookieName)
 {
 return HttpContext.Request.Cookies.Get(cookieName);
 }

 public void SetCookie(HttpCookie cookie)
 {
 HttpContext.Response.Cookies.Set(cookie);
 }
}
```

И теперь используем эту реализацию для работы с Cookies в CustomAuthentication (/Global/Auth/CustomAuthentication.cs):

```
public IAuthCookieProvider AuthCookieProvider { get; set; }
```

и вместо `HttpContext.Request.Cookies.Get` – используем `GetCookie()` и  
`HttpContext.Response.Cookies.Set` – соответственно `SetCookie()`.

Изменяем и в `IAuthencation.cs` (`/Global/Auth/IAuthencation.cs`):

```
public interface IAuthentication
{
 /// <summary>
 /// Конекст (тут мы получаем доступ к запросу и куписам)
 /// </summary>
 IAuthCookieProvider AuthCookieProvider { get; set; }
```

И в `AuthHttpModule.cs` (`/Global/Auth/AuthHttpModule.cs`):

```
var auth = DependencyResolver.Current.GetService<IAuthentication>();
auth.AuthCookieProvider = new HttpContextCookieProvider(context);
```

## MockHttpContext

Теперь создадим Mock-объекты для `HttpContext` в `LessonProject.UnitTesting`:

`MockHttpContext.cs` в (`/Mock/HttpContext.cs`):

```
public class MockHttpContext : Mock<HttpContextBase>
{
 [Inject]
 public HttpCookieCollection Cookies { get; set; }

 public MockHttpCachePolicy Cache { get; set; }

 public MockHttpBrowserCapabilities Browser { get; set; }

 public MockHttpSessionState SessionState { get; set; }

 public MockHttpServerUtility ServerUtility { get; set; }

 public MockHttpResponse Response { get; set; }

 public MockHttpRequest Request { get; set; }

 public MockHttpContext(MockBehavior mockBehavior = MockBehavior.Strict)
 : this(null, mockBehavior)
 {
 }

 public MockHttpContext(IAuthentication auth, MockBehavior mockBehavior =
MockBehavior.Strict)
 : base(mockBehavior)
 {
 //request
 Browser = new MockHttpBrowserCapabilities(mockBehavior);
 Browser.Setup(b => b.IsMobileDevice).Returns(false);

 Request = new MockHttpRequest(mockBehavior);
 Request.Setup(r => r.Cookies).Returns(Cookies);
 Request.Setup(r => r.ValidateInput());
 Request.Setup(r => r.UserAgent).Returns("Mozilla/5.0 (Windows NT 6.1; WOW64)
AppleWebKit/537.11 (KHTML, like Gecko) Chrome/23.0.1271.64 Safari/537.11");
 Request.Setup(r => r.Browser).Returns(Browser.Object);
 this.Setup(p => p.Request).Returns(Request.Object);

 //response
 Cache = new MockHttpCachePolicy(MockBehavior.Loose);
```

```

Response = new MockHttpResponse(mockBehavior);
Response.Setup(r => r.Cookies).Returns(Cookies);
Response.Setup(r => r.Cache).Returns(Cache.Object);
this.Setup(p => p.Response).Returns(Response.Object);

//user
if (auth != null)
{
 this.Setup(p => p.User).Returns(() => auth.CurrentUser);
}
else
{
 this.Setup(p => p.User).Returns(new UserProvider("", null));
}

//Session State
SessionState = new MockHttpSessionState();
this.Setup(p => p.Session).Returns(SessionState.Object);

//Server Utility
ServerUtility = new MockHttpServerUtility(mockBehavior);
this.Setup(p => p.Server).Returns(ServerUtility.Object);

//Items
var items = new ListDictionary();
this.Setup(p => p.Items).Returns(items);
}
}

```

Кроме этого создаем еще такие классы:

- MockHttpCachePolicy
- MockHttpBrowserCapabilities
- MockHttpSessionState
- MockHttpServerUtility
- MockHttpResponse
- MockHttpRequest

Все эти мок-объекты весьма тривиальны, кроме MockSessionState, где и хранится session-storage (/Mock/Http/MockHttpSessionState.cs):

```

public class MockHttpSessionState : Mock< HttpSessionStateBase>
{
 Dictionary<string, object> sessionStorage;

 public MockHttpSessionState(MockBehavior mockBehavior = MockBehavior.Strict)
 : base(mockBehavior)
 {
 sessionStorage = new Dictionary<string, object>();
 this.Setup(p => p[It.IsAny<string>()]).Returns((string index) =>
sessionStorage[index]);
 this.Setup(p => p.Add(It.IsAny<string>(),
It.IsAny<object>())).Callback<string, object>((name, obj) =>
{
 if (!sessionStorage.ContainsKey(name))
 {
 sessionStorage.Add(name, obj);
 }
 else
 {
 sessionStorage[name] = obj;
 }
});
 }
}

```

```
 }
}
```

Создаем FakeAuthCookieProvider.cs (/Fake/FakeAuthCookieProvider.cs):

```
public class FakeAuthCookieProvider : IAuthCookieProvider
{
 [Inject]
 public HttpCookieCollection Cookies { get; set; }

 public HttpCookie GetCookie(string cookieName)
 {
 return Cookies.Get(cookieName);
 }

 public void SetCookie(HttpCookie cookie)
 {
 if (Cookies.Get(cookie.Name) != null)
 {
 Cookies.Remove(cookie.Name);
 }
 Cookies.Add(cookie);
 }
}
```

Фух! Инициализируем это в UnitTestSetupFixture.cs (/Setup/UnitTestSetupFixture.cs):

```
protected virtual void InitAuth(StandardKernel kernel)
{
 kernel.Bind<HttpCookieCollection>().To<HttpCookieCollection>();

kernel.Bind<IAuthCookieProvider>().To<FakeAuthCookieProvider>().InSingletonScope();
 kernel.Bind<IAuthentication>().ToMethod<CustomAuthentication>(c =>
 {
 var auth = new CustomAuthentication();
 auth.AuthCookieProvider = kernel.Get<IAuthCookieProvider>();
 return auth;
 });
}
```

Заметим, что Bind происходит на SingletonScope(), т.е. единожды авторизовавшись в каком-то тесте, мы в последующих тестах будем использовать эту же авторизацию.

Компилим и пытаемся с этим всем взлететь. Сейчас начнется магия..

## Проверка валидации

Если мы просто вызовем что-то типа:

```
var registerUser = new UserView()
{
 Email = "user@sample.com",
 Password = "123456",
 ConfirmPassword = "1234567",
 AvatarPath = "/file/no-image.jpg",
 BirthdateDay = 1,
 BirthdateMonth = 12,
 BirthdateYear = 1987,
 Captcha = "1234"
};
var result = controller.Register(registerUser);
```

То, во-первых, никакая неявная валидация не выполнится, а во-вторых, у нас там есть session и мы ее не проинициализировали, она null и всё – ошибка. Так что проверку валидации (та, что в атрибутах) будем устраивать через отдельный класс. Назовем его Валидатор Валидаторович (/Tools/Validator.cs):

```

public class ValidatorException : Exception
{
 public ValidationAttribute Attribute { get; private set; }

 public ValidatorException(ValidationException ex, ValidationAttribute attribute)
 : base(attribute.GetType().Name, ex)
 {
 Attribute = attribute;
 }
}

public class Validator
{
 public static void ValidateObject<T>(T obj)
 {
 var type = typeof(T);
 var meta =
type.GetCustomAttributes(false).OfType<MetadataTypeAttribute>().FirstOrDefault();
 if (meta != null)
 {
 type = meta.MetadataClassType;
 }

 var typeAttributes = type.GetCustomAttributes(typeof(ValidationAttribute),
true).OfType<ValidationAttribute>();
 var validationContext = new ValidationContext(obj);
 foreach (var attribute in typeAttributes)
 {
 try
 {
 attribute.Validate(obj, validationContext);
 }
 catch (ValidationException ex)
 {
 throw new ValidatorException(ex, attribute);
 }
 }

 var PropertyInfo = type.GetProperties();
 foreach (var info in PropertyInfo)
 {
 var attributes = info.GetCustomAttributes(typeof(ValidationAttribute),
true).OfType<ValidationAttribute>();
 foreach (var attribute in attributes)
 {
 var objPropInfo = obj.GetType().GetProperty(info.Name);
 try
 {
 attribute.Validate(objPropInfo.GetValue(obj, null),
validationContext);
 }
 catch (ValidationException ex)
 {
 throw new ValidatorException(ex, attribute);
 }
 }
 }
 }
}

```

Итак, что тут у нас происходит. Вначале мы получаем все атрибуты класса T, которые относятся к типу ValidationAttribute:

```
var typeAttributes = type.GetCustomAttributes(typeof(ValidationAttribute),
true).OfType<ValidationAttribute>();
 var validationContext = new ValidationContext(obj);
 foreach (var attribute in typeAttributes)
 {
 try
 {
 attribute.Validate(obj, validationContext);
 }
 catch (ValidationException ex)
 {
 throw new ValidatorException(ex, attribute);
 }
 }
```

Потом аналогично для каждого свойства:

```
var propertyInfo = type.GetProperties();
 foreach (var info in propertyInfo)
 {
 var attributes = info.GetCustomAttributes(typeof(ValidationAttribute),
true).OfType<ValidationAttribute>();
 foreach (var attribute in attributes)
 {
 var objPropInfo = obj.GetType().GetProperty(info.Name);
 try
 {
 attribute.Validate(objPropInfo.GetValue(obj, null),
validationContext);
 }
 catch (ValidationException ex)
 {
 throw new ValidatorException(ex, attribute);
 }
 }
 }
```

Если валидация не проходит, то происходит исключение, и мы оборачиваем его в ValidatorException, передавая еще и атрибут, по которому произошло исключение.

Теперь по поводу капчи и Session. Мы должны контроллеру передать контекст (MockHttpContext):

```
var controller =
DependencyResolver.Current.GetService<Areas.Default.Controllers.UserController>();
 var httpContext = new MockHttpContext().Object;
 ControllerContext context = new ControllerContext(new
RequestContext(httpContext, new RouteData()), controller);
 controller.ControllerContext = context;
 controller.Session.Add(CaptchaImage.CaptchaValueKey, "1111");
```

И теперь всё вместе:

```
[Test]
 public void Index_RegisterUserWithDifferentPassword_ExceptionCompare()
 {
 //init
 var controller =
DependencyResolver.Current.GetService<Areas.Default.Controllers.UserController>();
```

```

 var httpContext = new MockHttpContext().Object;
 ControllerContext context = new ControllerContext(new
RequestContext(httpContext, new RouteData()), controller);
 controller.ControllerContext = context;

 //act
 var registerUserView = new UserView()
{
 Email = "user@sample.com",
 Password = "123456",
 ConfirmPassword = "1234567",
 AvatarPath = "/file/no-image.jpg",
 BirthdateDay = 1,
 BirthdateMonth = 12,
 BirthdateYear = 1987,
 Captcha = "1111"
};
try
{
 Validator.ValidateObject<UserView>(registerUserView);
}
catch (Exception ex)
{
 Assert.IsInstanceOf<ValidatorException>(ex);

Assert.IsInstanceOf<System.ComponentModel.DataAnnotations.CompareAttribute>(((ValidatorEx
ception)ex).Attribute);
}
}

```

Запускаем, и всё получилось. Но капча проверяется непосредственно в методе контроллера.  
Специально для капчи:

```

[Test]
public void Index_RegisterUserWithWrongCaptcha_ModelStateWithError()
{
 //init
 var controller =
DependencyResolver.Current.GetService<Areas.Default.Controllers.UserController>();
 var httpContext = new MockHttpContext().Object;
 ControllerContext context = new ControllerContext(new
RequestContext(httpContext, new RouteData()), controller);
 controller.ControllerContext = context;
 controller.Session.Add(CaptchaImage.CaptchaValueKey, "2222");
 //act
 var registerUserView = new UserView()
{
 Email = "user@sample.com",
 Password = "123456",
 ConfirmPassword = "1234567",
 AvatarPath = "/file/no-image.jpg",
 BirthdateDay = 1,
 BirthdateMonth = 12,
 BirthdateYear = 1987,
 Captcha = "1111"
};

 var result = controller.Register(registerUserView);
 Assert.AreEqual("Текст с картинки введен неверно",
controller.ModelState["Captcha"].Errors[0].ErrorMessage);
}

```

Круто?

## Проверка авторизации

Например, мы должны проверить, что, если я захожу не под админом, то в авторизованную часть (в контроллер, помеченный атрибутом [Authorize(Roles="admin")]) – обычному пользователю не дадут войти. Есть отличный способ это проверить. Обратим внимание на класс ControllerActionInvoker и отнаследуем его для вызовов (/Fake/FakeControllerActionInvoker.cs + FakeValueProvider.cs):

```
public class FakeValueProvider
{
 protected Dictionary<string, object> Values { get; set; }

 public FakeValueProvider()
 {
 Values = new Dictionary<string, object>();
 }

 public object this[string index]
 {
 get
 {
 if (Values.ContainsKey(index))
 {
 return Values[index];
 }
 return null;
 }

 set
 {
 if (Values.ContainsKey(index))
 {
 Values[index] = value;
 }
 else
 {
 Values.Add(index, value);
 }
 }
 }
}

public class FakeControllerActionInvoker<TExpectedResult> : ControllerActionInvoker
where TExpectedResult : ActionResult
{
 protected FakeValueProvider FakeValueProvider { get; set; }

 public FakeControllerActionInvoker()
 {
 FakeValueProvider = new FakeValueProvider();
 }

 public FakeControllerActionInvoker(FakeValueProvider fakeValueProvider)
 {
 FakeValueProvider = fakeValueProvider;
 }

 protected override ActionExecutedContext
InvokeActionMethodWithFilters(ControllerContext controllerContext, IList<IActionFilter>
filters, ActionDescriptor actionDescriptor, IDictionary<string, object> parameters)
 {
 return base.InvokeActionMethodWithFilters(controllerContext, filters,
actionDescriptor, parameters);
 }
}
```

```

 protected override object GetParameterValue(ControllerContext controllerContext,
ParameterDescriptor parameterDescriptor)
{
 var obj = FakeValueProvider[parameterDescriptor.ParameterName];
 if (obj != null)
 {
 return obj;
 }
 return parameterDescriptor.DefaultValue;
}

 protected override void InvokeActionResult(ControllerContext controllerContext,
ActionResult ActionResult)
{
 Assert.IsInstanceOf<TExpectedResult>(ActionResult);
}
}

```

По сути это «вызывальщик» action-методов контроллеров, где Generic класс – это ожидаемый класс результата. В случае неавторизации это будет HttpNotFoundResult. Сделаем тест (/Test/Admin/HomeControllerTest.cs):

```

[TestFixture]
public class AdminHomeControllerTest
{
 [Test]
 public void Index_NotAuthorizeGetDefaultView_RedirectToLoginPage()
 {
 var auth = DependencyResolver.Current.GetService<IAuthentication>();
 auth.Login("chernikov@gmail.com", "password2", false);

 var httpContext = new MockHttpContext(auth).Object;
 var controller =
DependencyResolver.Current.GetService<Areas.Admin.Controllers.HomeController>();
 var route = new RouteData();
 route.Values.Add("controller", "Home");
 route.Values.Add("action", "Index");
 route.Values.Add("area", "Admin");

 ControllerContext context = new ControllerContext(new
RequestContext(httpContext, route), controller);
 controller.ControllerContext = context;

 var controllerActionInvoker = new
FakeControllerActionInvoker<HttpUnauthorizedResult>();
 var result =
controllerActionInvoker.InvokeAction(controller.ControllerContext, "Index");
 }
}

```

Запускаем тест, он проходит. Сделаем, чтобы авторизация была под пользователем admin и будем ожидать получение ViewResult:

```

[Test]
 public void Index_AdminAuthorize_GetViewResult()
 {
 var auth = DependencyResolver.Current.GetService<IAuthentication>();
 auth.Login("admin", "password", false);

 var httpContext = new MockHttpContext(auth).Object;

```

```

 var controller =
DependencyResolver.Current.GetService<Areas.Admin.Controllers.HomeController>();
 var route = new RouteData();
 route.Values.Add("controller", "Home");
 route.Values.Add("action", "Index");
 route.Values.Add("area", "Admin");

 ControllerContext context = new ControllerContext(new
RequestContext(HttpContext, route), controller);
 controller.ControllerContext = context;

 var controllerActionInvoker = new FakeControllerActionInvoker<ViewResult>();
 var result =
controllerActionInvoker.InvokeAction(controller.ControllerContext, "Index");
 }
}

```

Так же прошли. Молодцом.

На этом давайте остановимся и подумаем, чего мы достигли. Мы можем оттестировать любой контроллер, проверить правильность любой валидации, проверку прав пользователя. Но это касается только контроллера. А как же работа с моделью? Да, мы можем проверить, что вызывается метод репозитория, но на этом всё. Да, мы можем написать Mock-методы для добавления, изменения, удаления, но как это поможет решить ту проблему, о которой я писал вначале главы? Как мы заметим, что что-то не так при упущении поля с тегом? В хрестоматийном примере NerdDinner тесты не покрывают эту область.

Есть IRepository, есть SqlRepository, есть MockRepository. И всё что находится в SqlRepository – это не покрытая тестами область. А там может быть реализовано очень многое. Что же делать? К чему этот TDD?

## Интегрированное тестирование

Идея будет совершенно безумной, мы будем использовать и проверять уже существующий код в SqlRepository. Для этого мы через Web.config находим базу (она должна располагаться локально), дублировать ее, подключаться к дубликату БД, проходить тесты и в конце, удалять дубликат БД.

Создаем проект LessonProject.IntegrationTest в папке Test.

Добавляем Ninject, Moq и NUnit:

```

Install-Package Ninject
Install-Package Moq
Install-Package NUnit

```

Так же создаем папку Sandbox и в Setup наследуем UnitTestSetupFixture (/Setup/IntegrationTestSetupFixture.cs) и функцию по копированию БД:

```

[SetUpFixture]
public class IntegrationTestSetupFixture : UnitTestSetupFixture
{
 public class FileListRestore
 {
 public string LogicalName { get; set; }
 public string Type { get; set; }
 }

 protected static string NameDb = "LessonProject";
}

```

```

protected static string TestDbName;

private void CopyDb(StandardKernel kernel, out FileInfo sandboxFile, out string
connectionString)
{
 var config = kernel.Get<IConfig>();
 var db = new DataContext(config.ConnectionStrings("ConnectionString"));

 TestDbName = string.Format("{0}_{1}", NameDb,
DateTime.Now.ToString("yyyyMMdd_HHmmss"));

 Console.WriteLine("Create DB = " + TestDbName);
 sandboxFile = new FileInfo(string.Format("{0}\\{1}.bak", Sandbox,
TestDbName));
 var sandboxDir = new DirectoryInfo(Sandbox);

 //backupFile
 var textBackUp = string.Format(@"-- Backup the database
BACKUP DATABASE [{0}]
TO DISK = '{1}'
WITH COPY_ONLY",
NameDb, sandboxFile.FullName);
 db.ExecuteCommand(textBackUp);

 var restoreFileList = string.Format("RESTORE FILELISTONLY FROM DISK = '{0}'",
sandboxFile.FullName);
 var fileListRestores =
db.ExecuteQuery<FileListRestore>(restoreFileList).ToList();
 var logicalDbName = fileListRestores.FirstOrDefault(p => p.Type == "D");
 var logicalLogDbName = fileListRestores.FirstOrDefault(p => p.Type == "L");

 var restoreDb = string.Format("RESTORE DATABASE [{0}] FROM DISK = '{1}' WITH
FILE = 1, MOVE N'{2}' TO N'{4}\\{0}.mdf', MOVE N'{3}' TO N'{4}\\{0}.ldf', NOUNLOAD, STATS
= 10", TestDbName, sandboxFile.FullName, logicalDbName.LogicalName,
logicalLogDbName.LogicalName, sandboxDir.FullName);
 db.ExecuteCommand(restoreDb);

 connectionString =
config.ConnectionStrings("ConnectionString").Replace(NameDb, TestDbName);
}

```

По порядку:

В строках

```

var config = kernel.Get<IConfig>();
var db = new DataContext(config.ConnectionStrings("ConnectionString"));

```

- получаем подключение к БД.

TestDbName = string.Format("{0}\_{1}", NameDb, DateTime.Now.ToString("yyyyMMdd\_HHmmss"));  
Создаем наименование тестовой БД.

```

//backupFile
var textBackUp = string.Format(@"-- Backup the database
BACKUP DATABASE [{0}]
TO DISK = '{1}'
WITH COPY_ONLY",
NameDb, sandboxFile.FullName);
db.ExecuteCommand(textBackUp);

```

- выполняем бекап БД в папку Sandbox.

```
 var restoreFileList = string.Format("RESTORE FILELISTONLY FROM DISK = '{0}'",
sandboxFile.FullName);
 var fileListRestores =
db.ExecuteQuery<FileListRestore>(restoreFileList).ToList();
 var logicalDbName = fileListRestores.FirstOrDefault(p => p.Type == "D");
 var logicalLogDbName = fileListRestores.FirstOrDefault(p => p.Type == "L");
- получаем логическое имя БД и файла логов, используя приведение к классу FileListRestore.
```

```
 var restoreDb = string.Format("RESTORE DATABASE [{0}] FROM DISK = '{1}' WITH
FILE = 1, MOVE N'{2}' TO N'{4}\{0}.mdf', MOVE N'{3}' TO N'{4}\{0}.ldf', NOUNLOAD, STATS
= 10", TestDbName, sandboxFile.FullName, logicalDbName.LogicalName,
logicalLogDbName.LogicalName, sandboxDir.FullName);
 db.ExecuteNonQuery(restoreDb);
```

- восстанавливаем БД под другим именем (TestDbName)

```
connectionString =
config.ConnectionStrings("ConnectionString").Replace(NameDb, TestDbName);
- меняем connectionString.
```

И теперь можем спокойно проинициализировать IRepository к SqlRepository:

```
protected override void InitRepository(StandardKernel kernel)
{
 FileInfo sandboxFile;
 string connectionString;
 CopyDb(kernel, out sandboxFile, out connectionString);
 kernel.Bind<webTemplateDbDataContext>().ToMethod(c => new
webTemplateDbDataContext(connectionString));
 kernel.Bind< IRepository>().To<SqlRepository>().InTransientScope();
 sandboxFile.Delete();
}
```

Итак, у нас есть sandboxFile – это файл бекапа, и connectionString – это новая строка подключения (к дубликату БД). Мы копируем БД, связываем именно с SqlRepository, но базу подсовываем не основную. И с ней можно делать всё что угодно. Файл бекапа базы в конце удаляем.

И дописываем уже удаление тестовой БД, после прогона всех тестов:

```
private void RemoveDb()
{
 var config = DependencyResolver.Current.GetService<IConfig>();
 var db = new DataContext(config.ConnectionStrings("ConnectionString"));

 var textCloseConnectionTestDb = string.Format(@"ALTER DATABASE [{0}] SET
SINGLE_USER WITH ROLLBACK IMMEDIATE", TestDbName);
 db.ExecuteNonQuery(textCloseConnectionTestDb);

 var textDropTestDb = string.Format(@"DROP DATABASE [{0}]", TestDbName);
 db.ExecuteNonQuery(textDropTestDb);
}
```

Используя TestDbName, закрываем подключение (а то оно активное), и удаляем базу данных.

Не забываем сделать копию Web.config:

```
xcopy ${SolutionDir}LessonProject\Web.config ${ProjectDir}Sandbox\ /y
```

Но кстати, иногда БД нет необходимости удалять. Например, мы хотим заполнить базу кучей данных автоматически, чтобы проверить поиск или пейджинг. Это мы рассмотрим ниже. А сейчас тест – реальное создание в БД записи:

```
[TestFixture]
public class DefaultUserControllerTest
{
 [Test]
 public void CreateUser_CreateNormalUser_CountPlusOne()
 {
 var repository = DependencyResolver.Current.GetService< IRepository>();

 var controller =
DependencyResolver.Current.GetService< LessonProject.Areas.Default.Controllers. UserController>();

 var countBefore = repository.Users.Count();
 var httpContext = new MockHttpContext().Object;

 var route = new RouteData();

 route.Values.Add("controller", "User");
 route.Values.Add("action", "Register");
 route.Values.Add("area", "Default");

 ControllerContext context = new ControllerContext(new
RequestContext(httpContext, route), controller);
 controller.ControllerContext = context;

 controller.Session.Add(CaptchaImage.CaptchaValueKey, "1111");

 var registerUserView = new UserView()
 {
 ID = 0,
 Email = "rollinx@gmail.com",
 Password = "123456",
 ConfirmPassword = "123456",
 Captcha = "1111",
 BirthdateDay = 13,
 BirthdateMonth = 9,
 BirthdateYear = 1970
 };

 Validator.ValidateObject< UserView>(registerUserView);
 controller.Register(registerUserView);

 var countAfter = repository.Users.Count();
 Assert.AreEqual(countBefore + 1, countAfter);
 }
}
```

Проверьте, что нет в БД пользователя с таким email.

Запускаем, проверяем. Работает. Кайф! Тут понятно, какие мощности открываются. И если юнит-тестирование – это как обработка минимальных кусочков кода, а тут – это целый сценарий. Но, кстати, замечу, что MailNotify всё же высыпает письма на почту. Так что перепишем его как сервис:

/LessonProject/Tools/Mail/IMailSender.cs:

```
public interface IMailSender
{
```

```

 void SendMail(string email, string subject, string body, MailAddress mailAddress
= null);
 }

/LessonProject/Tools/Mail/MailSender.cs:

public class MailSender : IMailSender
{
 [Inject]
 public IConfig Config { get; set; }

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

 public 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);
 }
 }
}

```

/LessonProject/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;
 }
 }

 private static IMailSender _mailSender;

 public static IMailSender MailSender
 {
 get
 {
 if (_mailSender == null)
 {
 _mailSender = (DependencyResolver.Current).GetService<IMailSender>();

 }
 return _mailSender;
 }
 }

 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));
 }
 }
}
```

/LessonProject/App\_Start/NinjectWebCommon.cs:

```
private static void RegisterServices(IKernel kernel)
{...}

kernel.Bind<IMailSender>().To<MailSender>();
}
```

Ну и в LessonProject.UnitTesting добавим MockMailSender (/Mock/Mail/MockMailSender.cs):

```
public class MockMailSender : Mock<IMailSender>
{
 public MockMailSender(MockBehavior mockBehavior = MockBehavior.Strict)
 : base(mockBehavior)
 {
 this.Setup(p => p.SendMail(It.IsAny<string>(), It.IsAny<string>(),
It.IsAny<string>(), It.IsAny<MailAddress>()))
 .Callback((string email, string subject, string body, MailAddress
address) =>
 Console.WriteLine(String.Format("Send mock email to: {0}, subject {1}",
email, subject)));
 }
}
```

В UnitTestSetupFixture.cs (/LessonProject.UnitTesting/Setup/UnitTestSetupFixture.cs):

```
protected virtual IKernel InitKernel()
{
 ...
 kernel.Bind<MockMailSender>().To<MockMailSender>();
 kernel.Bind<IMailSender>().ToMethod(p =>
kernel.Get<MockMailSender>().Object);
 return kernel;
}
```

Запускаем, тесты пройдены, но на почту уже ничего не отправляется.

```
=====
=====START=====
=====
Create DB = LessonProject_20130314_104218
Send mock email to: chernikov@googlemail.com, subject Регистрация на

=====
=====BYE!=====
=====
```

## Генерация данных

Кроме всего прочего, мы можем и не удалять базу данных после пробегов теста. (переписать) Я добавлю **GenerateData** проект в папку Test, но подробно рассматривать мы его не будем, просто чтобы был. Он достаточно тривиальный. Суть его – есть некоторые наименования, и мы используем их для генерации. Например, для генерации фамилии используются фамилии американских президентов (зная их, мы сразу отличаем их от других фамилий, которые скорее будут реальными).

Это также в будущем позволяет избежать «эффекта рыбы», когда в шаблоне тестовые данные были одной определенной, но не максимальной длины и шаблон выглядел прилично, но при использовании реальных данных всё поехало.

Создадим 100 пользователей и потом посмотрим на них:

```
[Test]
public void CreateUser_Create100Users_NoAssert()
{
 var repository = DependencyResolver.Current.GetService< IRepository>();
```

```

 var controller =
DependencyResolver.Current.GetService<LessonProject.Areas.Default.Controllers.UserController>();

 var httpContext = new MockHttpContext().Object;

 var route = new RouteData();

 route.Values.Add("controller", "User");
 route.Values.Add("action", "Register");
 route.Values.Add("area", "Default");

 ControllerContext context = new ControllerContext(new
RequestContext(httpContext, route), controller);
 controller.ControllerContext = context;

 controller.Session.Add(CaptchaImage.CaptchaValueKey, "1111");

 var rand = new Random((int)DateTime.Now.Ticks);
 for (int i = 0; i < 100; i++)
 {
 var registerUserView = new UserView()
 {
 ID = 0,
 Email = Email.GetRandom(Name.GetRandom(), Surname.GetRandom()),
 Password = "123456",
 ConfirmPassword = "123456",
 Captcha = "1111",
 BirthdateDay = rand.Next(28) + 1,
 BirthdateMonth = rand.Next(12) + 1,
 BirthdateYear = 1970 + rand.Next(20)
 };
 controller.Register(registerUserView);
 }
 }
}

```

В IntegrationTestFixture.cs отключим удаление БД после работы  
(/Setup/IntegrationTestFixture.cs):

```
protected static bool removeDbAfter = false;
```

В Web.config установим соединение с тестовой БД:

```
<add name="ConnectionString" connectionString="Data Source=SATURN-PC;Initial Catalog=LessonProject_20130314_111020;Integrated Security=True;Pooling=False"
providerName="System.Data.SqlClient" />
```

И запустим сайт:

ID	Email	Date
116	A.Harrison1989@gmail.com	14.03.2013 11:10:22
117	DennisBush1989@gmail.com	14.03.2013 11:10:22
118	Seaborne.Ford1942@mailbox.com	14.03.2013 11:10:22
119	ShamusAdams1937@mail.ru	14.03.2013 11:10:22
120	G.Tyler@mailbox.com	14.03.2013 11:10:22

## Итог

В этом уроке мы рассмотрели:

- Принципы TDD и когда они не срабатывают
- NUnit и как с ним работать
- Mock и как с ним работать
- Unit-тесты и как этот инструмент позволяет улучшить нам качество кода
- Integration-тесты, и как мы можем их использовать

Тестирование – это очень большая область, это даже отдельная профессия и склад ума (не совсем программистский). И качество кода будет зависеть не только от применения технологий, хотя, бесспорно, соблюдение логических принципов TDD и внутренних процессов при разработке программ позволяет избежать множества ошибок. Написание тестов – не панацея от всех бед, это инструмент, и важно правильно им пользоваться..

Мы обошли вниманием тестирование клиентской части, и честно говоря, я не знаю, как это должно происходить. В JQuery только в октябре 2011го начали развивать проект qUnit, но информации по нему почти нет .

## Урок F. Работа как она есть

Цель: финальный урок по созданию приложения. Написание технического задания. Создание БД. Переименование webTemplate. Применение скаффолдинга. Админка. Основной сайт. Тесты.

### О главном

Это финальный урок, и тут я немного отойду от конкретного программирования и поразмышляю о работе.

Программирование – это работа, это профессия, это творчество. Когда я учился в университете и с кем-то шел по дороге домой, мы часто спорили, что лучше Windows или Linux, Delphi или C++. Тогда мы могли не спать ночами, чтобы красиво переписать построение семантического дерева для компилятора. Мы изучали пролог, лисп, конечные автоматы, структуры данных. Мы учились видеть красоту быстрой сортировки Хоара реализованную на лиспе. ВО!:

```
(defun quicksort (lis) (if (null lis) nil
 (let* ((x (car lis)) (r (cdr lis)) (fn (lambda (a) (< a x))))
 (append (quicksort (remove-if-not fn r)) (list x)
 (quicksort (remove-if fn r))))))
```

Но теперь я рассматриваю программирование как услугу. Как что-то, за что мне платят деньги. Я занимаюсь фрилансом уже три года. В начале работы фрилансером я программировал не только веб и не только на asp.net mvc. Был и php на ZendFramework, и написание модулей для расчета стратегий для торговли на РТС на Quirk.

Но потом выделил направление asp.net mvc и стал в нем развиваться. Глобальная, стратегическая задача стояла следующая: «**Увеличить скорость разработки**». Скорость разработки является самым важным параметром. Прежде всего, мы сохраняем свое время, а также лучше придерживаемся сроков. Вместе с тем, я не хотел скатиться в конвейерную разработку, где программист, раз за разом делает одни и те же банальные вещи. Я выделил для себя метод накопления и создания тех инструментов, которые нужны почти всегда.

Так и появился webTemplate. По сути – это шаблон всех моих проектов. Он сам уже четвертой версии. Это основной мой продукт. Но вначале я расскажу о принципах взаимоотношениях с заказчиками, далее о правилах составления технического задания, потом о режиме работы. И в самом конце об webTemplate.

### О принципах

Отношения заказчик-работник всегда вызывают много споров. То заказчик обидел, то работник не прав. Фриланс – это то еще болото, и часто заказчик уже «опытный» и при первом разговоре ставит в известность, что «никаких предоплат, ибо повидали!». Вот мои принципы:

- **Всегда берите от 30% предоплату.** Злопамятные заказчики, которые обобщают всех фрилансеров в одну кучу, в общем-то, не совсем адекватные люди. Но(!) если необходимо это правило нарушить, то нужно договариваться на следующее: «я показываю прототип, а вы тут же перечисляете мне деньги». Ну и понятно, что тут bootstrap + скаффолдинг и 2-3 дня, у нас уже есть что показать, заказчик уходит в лояльность и высыпает деньги.
- **Всегда сами пишите ТЗ. Хотя бы тезисно.** Если заказчик предоставляет ТЗ, а в большинстве своем какой-то документ есть, то не факт что вы поймете правильно. И у меня были такие случаи, что когда я переписал ТЗ, то заказчик сообщил что «всё совсем не так». Так что ТЗ пишите сами. О нем еще поговорим.

- **Уделяйте внимание общению.** Созвонитесь по скайпу и как можно более подробно проговорите подробности. Не беритесь за создание проекта, пока не закроете все неточности. Если такой возможности нет, то в бюджет проекта пропишите сумму, которая нужна для решения вот этой непонятной задачи. Конечно, сумма может быть большая, и заказчика сразу взволнует этот момент, и, скорее всего, он найдет время сэкономить эти деньги. Если же данной копейки он не заметил, то тем лучше, задача заранее оплачена сполна.
- **Никогда, ни при каких обстоятельствах не пропадайте надолго, не предупредив.** Этим грешат все. Запороли дедлайн. Должны были стартовать проект, а старый еще недописан, банальная ошибка выбила из колеи на 2 дня. Сроки горят. Всё это фигня, никогда не пропадайте. Если спросят, что происходит – ответьте честно. Если давят на ранюю договоренность, что вы подводите, что «говорили что будет в понедельник, а уже пятница!» - почувствуйте, что вы большой, толстый и меланхоличный слон. Худшее, что вы можете, это поддаться на эту провокацию, хамить в ответ, не сделать работу и свалить.
- **Имейте запас денег.** Иметь денежную «подушку» на 2-3 месяца такой же качественной жизни, как вы ведете в данный момент, - очень важно. Во-первых, вы не работаете в запарке, что вот долги растут. Во-вторых, не попадете в кабалу текущего проекта. В третьих, если вас кинут, то вы спокойно разрулитесь на следующем проекте.
- **Не отдавайте проект на чужой сервер до полной оплаты.** Даже, если вам внесли предоплату, и вы отдаете проект, то у заказчика есть возможность вас кинуть. Давайте тестировать на своем сервере.
- **Будьте лояльны к доработкам и правкам.** Некоторые вещи тяжело предусмотреть, но они могут иметь ключевое значение для проекта. Если это произошло и заказчик настаивает на том, что это было изначально проговорено, то тут два варианта. Всё, что вы напишете в ТЗ, имеет больший вес. Но, если доработки незначительны (чаще всего так и бывает), то проще сделать. Не обязательно каждую мелочь превращать в предмет для дискуссии, но и не позволяйте садиться на голову.
- **Не переходите на «ты», пока сам заказчик этого не захочет.** Всегда лучше общаться на «вы», даже если вы ровесники. Тем самым это создает более сдержанную и вежливую атмосферу. Хотя в будущем будет легче перейти на «ты». Это уже больше относится к старым заказчикам.
- **Не беритесь за поддержку чужих проектов.** Не стоит недооценивать то, что вам хотят подсунуть. У меня было несколько проектов, которые нужно было поддерживать, и они были написаны до меня. Мне приходилось разбираться с чужим кодом, на это уходило время, и часто заказчику нельзя было объяснить, какую пользу это приносит. Вторым моментом было то, что часто случался коллапс на ровном месте, код переставал работать, и нужно было искать причину, которая была не до конца ясна.
- **Выуживайте как можно больше информации о целях проекта.** Чаще заказчик может не знать достаточно о технологиях, и в ТЗ описывает решение, которое, по его мнению, должно работать. Если вы знаете более элегантное и оптимальное решение, то не стесняйтесь продолжить.
- **Спорьте о проекте, но помните, что он платит деньги.** Да, возможно, что некоторые его идеи и размышления не верны, и вы можете предложить лучшее решение. Предлагайте, убеждайте, даже если это решение и будет стоить дороже. Страйтесь избавиться от глупых решений, но если заказчик сильно настаивает – принимайте его сторону. Он платит, даже за свои ошибки.

- **Полюбите работу.** Посмотрите на проект, как на свой собственный проект. Сделайте лучшую реализацию, даже если это будет выходить за рамки бюджета и времени.
- **Показывайте прогресс.** Уведомляйте заказчика о движении по проекту. Ничто так не успокаивает как Progress Bar. Даже если вы опаздываете со сроками, показывайте, что дело идет, не оставляйте в неведении.
- **Если нечего показать – спрашивайте.** Даже если вам нечего показать, покажите, что задача у вас в голове. Наведите много уточняющих вопросов, вовлекитесь в процесс обсуждения.
- **Не работайте по ночам.** Не будьте рабом работы, не старайтесь сделать работу за ночь/за выходные. Вы сорветесь, и будете долго восстанавливаться. Но если работа идет и уже три часа ночи – работайте, об этих моментах вы будете вспоминать с удовольствием.
- **Посмотрите на проблему издалека.** Изучите ТЗ полностью прежде, чем что-то писать. Сформируйте образ, представьте, как будут реализованы блоги, но поставьте вопрос «а как можно это сделать быстрее в три раза, за 2 часа?», «Как можно внести много сложных данных без единой ошибки?», «Быстрее ли будет написать парсер или нет?», «Применимо ли для данного случая разработать свой шаблон или скаффолдер?»
- **Не объявляйте сроки и стоимость до тех пор, пока не напишете ТЗ.** Тут даже не пытайтесь что-то в уме посчитать, всегда ошибетесь.

### **Техническое задание**

Техническое задание – это ваше всё, это 70% успеха. Техническое задание должно отвечать на следующие вопросы и состоять из следующих частей:

- **Цель.** Тут кратко описываем цель проекта. Что мы делаем? Для чего нужен этот проект?
- **Части проекта.** Они должны логично происходить из цели проекта и будут прототипом для создания таблиц в БД. Тут должно быть:
  - Язык сайта (если многозычный, то какие именно языки)
  - Сущности проекта. Это может быть:
    - Пользователь
    - Пост
    - Новость
    - Видео
    - Фотоальбом
    - Фотокарточки
    - Машина
    - Друзья
    - Музыка
    - Игра и т.д.
  - Функции сайта. К этому относится:
    - Регистрация
    - Личный кабинет
    - Мои фотографии
    - Статус
  - Админка сайта.
    - Пользователи
      - С возможностью активировать
      - Забанить

- Видео
  - Добавить
  - Изменить
  - Удалить
- Новость
  - Добавить
  - Изменить
  - Удалить
- Работа с смс
- Регистрация через социальные сети
- Работа с Webmoney или другими онлайн-системами расчетов
- **Сроки и бюджет.** После описания всех частей проекта составляем табличку
  - **Работа.** Например, «создание БД проекта» или «Регистрация»
  - **Сроки (в часах).** Страйтесь, чтобы в каждой ячейке сроки не были больше 20 часов. И уж точно не 40+ часов. Иначе, попробуйте разбить задание на более мелкие части.
  - **Деньги.** Просто умножаете на текущую ставку.
  - **Итог.** Далее подбиваете итог, не глядя и не округляя. 1695\$, например? Да, так и пишете. Сроки и бюджет – это важная часть, она дает нам определение, сколько времени займет разработка (рассчитывайте на 6 часов в день, а не 8 или еще хуже 10) и сколько это будет стоить.

Таким образом, ТЗ декларирует три вещи, самые важные:

- Что надо сделать
- Сколько это займет времени
- Сколько это будет стоить

Пропуская это звено, вы изначально гробите проект. Вы ошибаетесь по всем трем пунктам, по объему работы, по времени, по стоимости. Страйтесь избегать этого.

Да и еще важно: ТЗ не должно быть большим, идеально – до 20 страниц. Не детализируйте слишком сильно, а то заказчик не прочитает его, и будет настаивать на использовании своего документа. Ваш документ должен быть лучше. Он и будет лучше, потому что там будет строка с ценой проекта. Но не перестарайтесь с ним.

Уточню еще раз: ТЗ использует один из самых важных принципов взаимодействия - Принцип декларации. Написано «работаем с 10:00 по 20:00», пришел клиент в 20:10, не стоит и объяснять, почему вы не можете их обслужить. Но в 19:50 закрывать дверь перед носом клиента – нарушение собственных правил, собственной декларации.

ТЗ утверждено. Можно приступать к работе.

### **Структура базы, webTemplate и Scaffolding**

Пробегая по техническому заданию, я выписываю все сущности отдельно в файлы, оцениваю связи. И потом описываю их в базе данных. На это уходит достаточно много времени, до 8 часов. После ТЗ – это самая важная часть проекта.

После этого я копирую проект webTemplate и переименовываю webTemplate → [новое название проекта]. Это занимает около 30 минут.

После этого в новом проекте я запускаю Scaffolding для нужных таблиц ProviderRepository и Model. Можно сразу составить все команды и скопировав в Package Manager Console запустить процесс и пойти пить чай.

Далее я во ViewModels дописываю необходимые SelectReference, убираю ненужные поля и добавляю необходимые атрибуты из ManagedAttribute и запускаю Scaffolding Controller.

После этого поправляю проект, чтобы он компилировался, и работаю с админкой.

Дальше, начиная от главной страницы, все остальные модули делаю день за днем.

### **Собственный ритм**

Бывало ли у вас такое, что в понедельник вы приходите еще разбитые, а в пятницу уходите уже «убитые», и цель понедельника – это дожить до пятницы. А цель месяца – это дожить до зарплаты. Или, поработав ночью, вы потом неделю ходите «убитыми».

Понаблюдав за собой, я заметил, что выходные у меня наступали в середине пятницы, а если я работал без выходных, то запал заканчивался гораздо быстрее. И еще всякие дела по хозяйству (например, нужно съездить в город) убивали весь рабочий настрой и целый день.

К тому же заказчики проектов, которые я поддерживаю, практически постоянно обращаются ко мне с правками. И их надо делать. А иногда нет времени на это.

И я сделал недельный ритм, который позволяет мне больше успевать и точнее планировать. Начинается он с воскресенья:

- Воскресенье – день планирования (рабочий). В этот день необходимо спланировать и начать дела, которые будут сделаны в следующие два дня.
- Понедельник – обычный рабочий день. В этот день я начинаю делать то, что запланировал в воскресенье.
- Вторник – ударный рабочий день. В этот день я обычно выключаю скайп и делаю очень большую часть работы. Рабочий день длится около 10-12 часов. Обычно этого хватает, чтобы сделать какой-то кусок, на который бы уходило два-три обычных дня. Цель воскресенья – это как раз спланировать именно этот большой объем работы.
- Среда – полувыходной день. В среду я отвлекаюсь на правки старых проектов, и решаю дела в городе. Может быть, даже полностью выходной день.
- Четверг – это или возврат после вторника к текущему проекту, или продолжение правок, если они еще остались.
- Пятница – это обычный рабочий день, обычно по текущему проекту доделывается что-то рутинное.
- Суббота – никакого программирования. Отдых.

Суть такова, что в понедельник я знаю, что будет адовый вторник и настраиваюсь на него, но тешит то, что в среду у меня выходной.

Далее сила в декларации, все знают, что во вторник я могу не ответить на звонок. Что их правки будут сделаны в среду или в четверг. Что, скорее всего, следующий большой объем проекта будет

загружен во вторник вечером, но в течение четверга и пятницы будут поправлены ошибки. Даже родные стараются подстраиваться под этот график.

Ищите свой ритм, реализуйте свои идеи.