## Урок 5. Создание записи в БД

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

### Введение

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

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

### Регистрация

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

public ActionResult Register()

{

var newUser = new User();

return View(newUser);

}

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

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

{

<fieldset>

<div class="control-group">

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

Email

</label>

<div class="controls">

@Html.ValidationMessage("Email")

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

</div>

</div>

<div class="control-group">

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

Password

</label>

<div class="controls">

@Html.ValidationMessage("Password")

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

</div>

</div>

<div class="form-actions">

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

Register

</button>

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

</div>

</fieldset>

}

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

Изучим основные Html-вставки:

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

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

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

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

@Html.ValidationMessage("Password")

- выводит тег ошибки если есть такая

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

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

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

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

[HttpGet]

public ActionResult Register()

{

var newUser = new User();

return View(newUser);

}

[HttpPost]

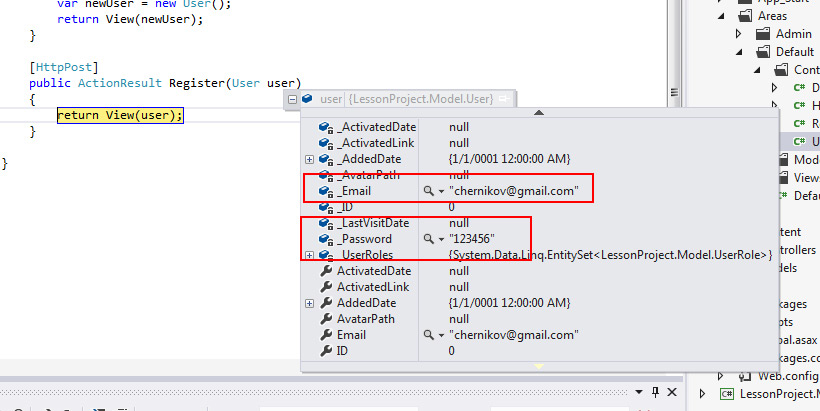
public ActionResult Register(User user)

{

return View(user);

}

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



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

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

public partial class User

{

public static string GetActivateUrl()

{

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

}

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

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

}

Добавим поля во View:

<div class="control-group">

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

Confirm Password

</label>

<div class="controls">

@Html.ValidationMessage("ConfirmPassword")

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

</div>

</div>

<div class="control-group">

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

Captcha

</label>

</div>

<div class="control-group">

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

Тут картинка 1234

</label>

<div class="controls">

@Html.ValidationMessage("Captcha")

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

</div>

</div>

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

### Валидация

Ок, вот условия для правильности данных:

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

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

#### IValidatableObject

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

Класс User:

public partial class User : **IValidatableObject**

{

public static string GetActivateUrl()

{

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

}

public string ConfirmPassword { get; set; }

public string Captcha { get; set; }

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

**{**

**//Не нулевой Email**

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

**{**

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

**}**

**//корректный Email**

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

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

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

**{**

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

**}**

**//пароль не нулевой**

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

**{**

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

**}**

**//пароли совпадают**

**if (Password != ConfirmPassword)**

**{**

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

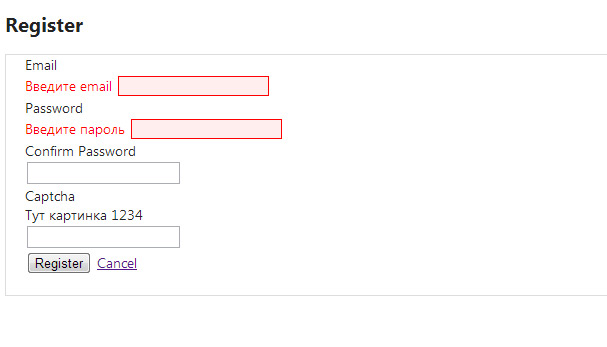
**}**

**}**

}

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

Выполняем форму, получаем:



Т.е. обе наши ошибки были отловлены.

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

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

if (user.Captcha != "1234")

{

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

}

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

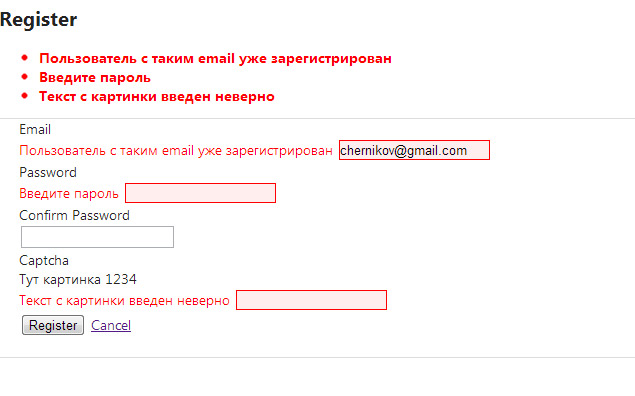
if (anyUser)

{

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

}

И результат:



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

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

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

public class UserView

{

public int ID { get; set; }

public string Email { get; set; }

public string Password { get; set; }

public string ConfirmPassword { get; set; }

public string Captcha { get; set; }

public string AvatarPath { get; set; }

}

### Automapping

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

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

Устанавливаем Automapper:

Install-Package AutoMapper

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

Создаем в /Mappers

public interface IMapper

{

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

}

Реализация:

public class CommonMapper : IMapper

{

static CommonMapper()

{

Mapper.CreateMap<User, UserView>();

Mapper.CreateMap<UserView, User>();

}

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

{

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

}

}

Регистрация (пусть будет как объект одиночка):

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

В BaseController:

public abstract class BaseController : Controller

{

[Inject]

public IRepository Repository { get; set; }

**[Inject]**

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

}

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

[HttpGet]

public ActionResult Register()

{

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

**return View(newUserView);**

}

[HttpPost]

public ActionResult Register(**UserView userView**)

{

if (userView.Captcha != "1234")

{

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

}

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

if (anyUser)

{

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

}

if (ModelState.IsValid)

{

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

//TODO: Сохранить

}

return View(userView);

}

И в Register.cshtml изменится первая строка:

@model LessonProject.Models.ViewModels.UserView

#### Атрибуты

Для UserView будем использовать для валидации атрибуты:

Добавим сборку:

using System.ComponentModel.DataAnnotations;

public class UserView

{

public int ID { get; set; }

[Required(ErrorMessage="Введите email")]

public string Email { get; set; }

[Required(ErrorMessage="Введите пароль")]

public string Password { get; set; }

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

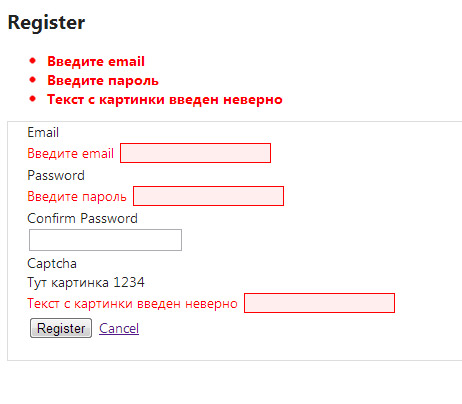
public string ConfirmPassword { get; set; }

public string Captcha { get; set; }

public string AvatarPath { get; set; }

}

Проверяем:



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

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

public class ValidEmailAttribute : ValidationAttribute

{

public override bool IsValid(object value)

{

if (value == null)

{

return true;

}

if (!(value is string))

{

return true;

}

var source = value as string;

if (string.IsNullOrWhiteSpace(source))

{

return true;

}

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

var match = regex.Match(source);

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

}

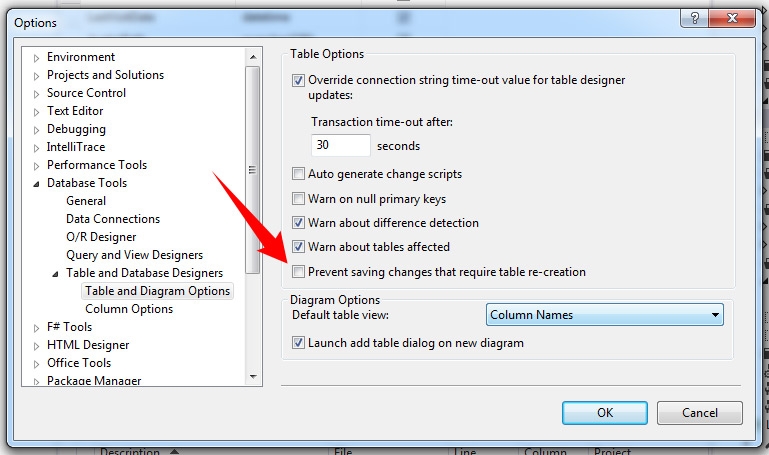
}

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

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

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

1. Добавляем поле в БД. Birthdate datetime null.

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

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

public bool UpdateUser(User instance)

{

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

if (cache != null)

{

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

cache.AvatarPath = instance.AvatarPath;

cache.Email = instance.Email;

Db.Users.Context.SubmitChanges();

return true;

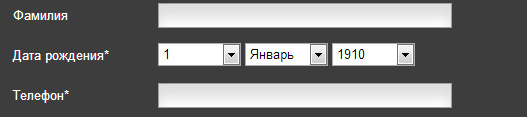
}

return false;

}

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

Выбор дня рождения у нас будет таким:



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

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

Смотрим SelectListItem:

public class SelectListItem

{

public SelectListItem();

public bool Selected { get; set; }

public string Text { get; set; }

public string Value { get; set; }

}

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

public IEnumerable<SelectListItem> SelectFruit

{

get

{

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

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

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

}

}

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

Ок, создадим реализацию для выбора дня рождения:

public int BirthdateDay { get; set; }

public int BirthdateMonth { get; set; }

public int BirthdateYear { get; set; }

public IEnumerable<SelectListItem> BirthdateDaySelectList

{

get

{

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

{

yield return new SelectListItem

{

Value = i.ToString(),

Text = i.ToString(),

Selected = BirthdateDay == i

};

}

}

}

public IEnumerable<SelectListItem> BirthdateMonthSelectList

{

get

{

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

{

yield return new SelectListItem

{

Value = i.ToString(),

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

Selected = BirthdateMonth == i

};

}

}

}

public IEnumerable<SelectListItem> BirthdateYearSelectList

{

get

{

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

{

yield return new SelectListItem

{

Value = i.ToString(),

Text = i.ToString(),

Selected = BirthdateYear == i

};

}

}

}

И во View:

<div class="control-group">

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

Birth date

</label>

<div class="controls">

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

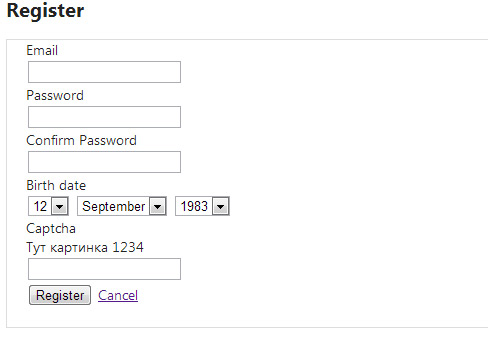
@Html.DropDownList("BirthdateMonth", Model.BirthdateMonthSelectList)

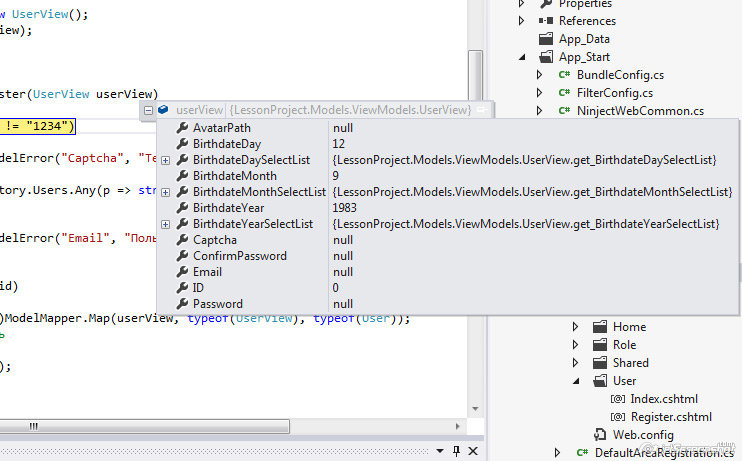
@Html.DropDownList("BirthdateYear", Model.BirthdateYearSelectList)

</div>

</div>

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





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

Mapper.CreateMap<User, UserView>()

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

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

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

Mapper.CreateMap<UserView, User>()

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

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

### Captcha

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

Создадим Tools/CaptchaImage.cs

/// <summary>

/// Генерация капчи

/// </summary>

public class CaptchaImage

{

public const string CaptchaValueKey = "CaptchaImageText";

public string Text

{

get { return text; }

}

public Bitmap Image

{

get { return image; }

}

public int Width

{

get { return width; }

}

public int Height

{

get { return height; }

}

// Internal properties.

private string text;

private int width;

private int height;

private string familyName;

private Bitmap image;

// For generating random numbers.

private Random random = new Random();

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

{

text = s;

SetDimensions(width, height);

GenerateImage();

}

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

{

text = s;

SetDimensions(width, height);

SetFamilyName(familyName);

GenerateImage();

}

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

// This member overrides Object.Finalize.

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

~CaptchaImage()

{

Dispose(false);

}

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

// Releases all resources used by this object.

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

public void Dispose()

{

GC.SuppressFinalize(this);

Dispose(true);

}

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

// Custom Dispose method to clean up unmanaged resources.

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

protected virtual void Dispose(bool disposing)

{

if (disposing)

// Dispose of the bitmap.

image.Dispose();

}

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

// Sets the image aWidth and aHeight.

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

private void SetDimensions(int aWidth, int aHeight)

{

// Check the aWidth and aHeight.

if (aWidth <= 0)

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

if (aHeight <= 0)

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

width = aWidth;

height = aHeight;

}

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

// Sets the font used for the image text.

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

private void SetFamilyName(string aFamilyName)

{

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

try

{

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

familyName = aFamilyName;

font.Dispose();

}

catch (Exception)

{

familyName = FontFamily.GenericSerif.Name;

}

}

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

// Creates the bitmap image.

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

private void GenerateImage()

{

// Create a new 32-bit bitmap image.

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

// Create a graphics object for drawing.

Graphics g = Graphics.FromImage(bitmap);

g.SmoothingMode = SmoothingMode.AntiAlias;

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

// Fill in the background.

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

g.FillRectangle(hatchBrush, rect);

// Set up the text font.

SizeF size;

float fontSize = rect.Height + 1;

Font font;

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

do

{

fontSize--;

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

size = g.MeasureString(text, font);

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

// Set up the text format.

StringFormat format = new StringFormat();

format.Alignment = StringAlignment.Center;

format.LineAlignment = StringAlignment.Center;

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

GraphicsPath path = new GraphicsPath();

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

float v = 4F;

PointF[] points =

{

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

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

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

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

};

Matrix matrix = new Matrix();

matrix.Translate(0F, 0F);

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

// Draw the text.

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

g.FillPath(hatchBrush, path);

// Add some random noise.

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

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

{

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

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

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

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

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

}

// Clean up.

font.Dispose();

hatchBrush.Dispose();

g.Dispose();

// Set the image.

image = bitmap;

}

}

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

Теперь сделаем метод вывода UserController.Captcha():

public ActionResult Captcha()

{

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

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

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

this.Response.Clear();

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

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

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

// Dispose of the CAPTCHA image object.

ci.Dispose();

return null;

}

Что тут происходит:

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

Запрашиваем картинку из Register.cshtml:

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

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

</label>

Проверка:

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

{

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

}

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

if (ModelState.IsValid)

{

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

**Repository.CreateUser(user);**

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

}