**ASP.NET MVC**

ASP.NET MVC – расшифровывается как модель(например – человек как набор органов), представление(одетый человек – как он себя показывает окружающему миру) и контроллер (мозг человека – как человек себя ведет).

Все модели лежат в папке Models. Например, модель задачи Request. Состоит из нескольких полей, некоторые из которых помечены как Required –то есть, обязательны для заполнения.

Например, у нас есть контроллер задачи, он лежит в общей папке Controllers, там же, где лежат и все остальные контроллеры. Называет контроллер задачи RequestController – он обрабатывается все задачи и реализуют логику по работе с базой данных(создание задачи, загрузка задач и тп)  
Но кто же отображает нашу модель ,полученную с контроллера?  
  
Отображает представление. Все представления лежат в папке Views и папки представлений совпадают по названию с названиеми контроллеров. А вот название представления на создание заявки называется Create.cshtml и содержит в себе форму для заполнения параметров заявки и отправления их на сервер.

Есть главное представление. Почему оно главное? Потому что в него, в зависимости от действия, включаются другие представления. Скажем так – оно как рамочка с меню, а нажатие кнопки меню – меняет содержимое этого главного представления.

Главное представление лежит в папке Views/\_Layout.cshtml и оно определяет – какой пользователь какие данные видит. Например, если пользователь в роли администратора, то он видит:  
@if (ViewContext.HttpContext.User.IsInRole("Администратор"))

{

<h2>Меню администратора: вы вошли как superadmin</h2>

<div class="menu">

<ul>

<li>

<a>Справочники</a>

<ul>

<li>@Html.ActionLink("Отделы", "Departments", "Service")</li>

<li>@Html.ActionLink("Активы", "Activ", "Service")</li>

<li>@Html.ActionLink("Категории", "Categories", "Service")</li>

</ul>

</li>

<li>

<a >Заявки</a>

<ul>

<li>@Html.ActionLink("Мои заявки", "Index", "Request")</li>

<li>@Html.ActionLink("Все заявки", "RequestList", "Request")</li>

<li>@Html.ActionLink("Новая", "Create", "Request")</li>

</ul>

</li>

<li>

<a >Пользователи</a>

<ul>

<li>@Html.ActionLink("Добавление", "Create", "User")</li>

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

</ul>

</li>

<li>@Html.ActionLink("Выйти", "LogOff", "Account")</li>

</ul>

</div>

}

Обязательно рекомендую это представление просмотреть глазами, многое станет понятно.

**О ПРОЕКТЕ:**

Описание доведу до контроллера аккаунта пользователей. Все остальные контроллеры работают по образу и подобию.

Наверное, все в общих чертах представляют стандартную систему назначения задач или документооборота(на примере университета): у пользователя возникает проблема, он ее вносит в систему организации, и затем кто-то (другой студент, преподаватель или другое должностное лицо) устраняет данную проблему. Казалось бы, ничего сложного, обычное приложение по работе с БД. Большие сложности представляют различные сопутствующие условия и общая реализация приложения.

В центре системы HelpDesk находятся заявки пользователей на проблемы, которые надо устранить. Каждая заявка имеет свой жизненный цикл: несколько этапов от начала работы по заявке до ее завершения. Пользователь определяет категорию заявки: к какой тематике примерно она может относиться - то ли это проблема с оборудованием, учебой, то ли у пользователя личные проблемы и он хочет о них поговорить.

Пользователь может добавить к заявке комментарий и прикрепить изображение, а также указать приоритет: что-то нужно сделать срочно, а что-то может подождать.

Сотрудник со статусом модератор просматривает новые заявки и назначает их исполнителям.

Конкретные исполнители уже просматривают назначенные им заявки и выполняют их, попутно обновляя статус.

Ну и во главе всей этой системы находится фигура администратора, который управляет системой пользователей - добавляет, удаляет и т.д., назначает пользователям статусы.

Наше приложение будет задействовать несколько различных субстанций: пользователи, заявки, роли, аппаратные ресурсы и т.д. Каждая субстанция или сущность представляет собой какой-то объект, обладающий неким набором свойств, которые позволят нам эффективно им управлять. И на первом этапе, как мне кажется, было бы эффективнее не бросаться сразу в работу с контроллерами, представления, писать какую-то логику, а вначале выделить все сущности, которые будут использоваться в приложении. И затем создать БД и в ней определить таблицы для большинства сущностей.

Какие это будут сущности? Для нашего приложения я выделил следующие сущности: пользователь, роль пользователя, заявка, жизненный цикл заявки, категория возникшей проблемы, отдел (в котором работает пользователь), актив (департамент + кабинет).

Вроде все сущности перечислил. Но в зависимости от поставленной задачи, можно добавить и больше сущностей и классов, сделать программу побогаче, но мы остановимся на этом. И сейчас мы создадим для всех сущностей классы.

Итак, сущность Пользователь у нас будет обладать следующими свойствами:

Id

Имя пользователя

Логин, под кторым пользователь будет входить в систему

Пароль для входа в систему

Должность

Отдел, в котором пользователь работает

Роль (администратор, модератор, исполнитель, простой пользователь)

using System;

using System.ComponentModel.DataAnnotations;

namespace HelpDeskTrain.Models

{

    public class User

    {

        // ID

        public int Id { get; set; }

        // Фамилия Имя Отчество

        [Required]

        [Display(Name = "Фамилия Имя Отчество")]

        [MaxLength(50, ErrorMessage = "Превышена максимальная длина записи")]

        public string Name { get; set; }

        // Логин

        [Required]

        [Display(Name = "Логин")]

        [MaxLength(50, ErrorMessage = "Превышена максимальная длина записи")]

        public string Login { get; set; }

        // Пароль

        [Required]

        [Display(Name = "Пароль")]

        [MaxLength(50, ErrorMessage = "Превышена максимальная длина записи")]

        public string Password { get; set; }

        // Должность

        [Display(Name = "Должность")]

        [MaxLength(50, ErrorMessage = "Превышена максимальная длина записи")]

        public string Position { get; set; }

        // Отдел

        [Display(Name = "Отдел")]

        public int? DepartmentId { get; set; }

        public Department Department { get; set; }

        // Статус

        [Required]

        [Display(Name = "Статус")]

        public int RoleId { get; set; }

        public Role Role { get; set; }

    }

}

Теперь добавим вторую модель Role, которая будет представлять статус пользователя:  
public class Role

{

    public int Id { get; set; }

    public string Name { get; set; }

}

Класс заявки

Класс заявки будет представлять класс ключевого объекта системы. В зависимости от того, насколько полную информацию предоставит пользователь в объекте заявки, настолько у него будут выше шансы, что его заявка будет быстро и без проблем выполнена. Для полноты информации снабдим класс свойствами, которые помогут понять заявку и также упростят ее управление.

Итак, добавим в папку Models следующий класс Request:

using System;

using System.ComponentModel.DataAnnotations;

namespace HelpDeskTrain.Models

{

    // Модель Заявка

    public class Request

    {

        // ID

        public int Id { get; set; }

        // Наименование заявки

        [Required]

        [Display(Name = "Название заявки")]

        [MaxLength(50, ErrorMessage = "Превышена максимальная длина записи")]

        public string Name { get; set; }

        // Описание

        [Required]

        [Display(Name = "Описание")]

        [MaxLength(200, ErrorMessage = "Превышена максимальная длина записи")]

        public string Description { get; set; }

        // Комментарий к заявке

        [Display(Name = "Комментарий")]

        [MaxLength(200, ErrorMessage = "Превышена максимальная длина записи")]

        public string Comment { get; set; }

        // Статус заявки

        [Display(Name = "Статус")]

        public int Status { get; set; }

        // Приоритет заявки

        [Display(Name = "Приоритет")]

        public int Priority { get; set; }

        // Номер кабинета

        [Display(Name = "Кабинет")]

        public int? ActivId { get; set; }

        public Activ Activ { get; set; }

        // Файл ошибки

        [Display(Name = "Файл с ошибкой")]

        public string File { get; set; }

        // Внешний ключ Категория

        [Display(Name = "Категория")]

        public int? CategoryId { get; set; }

        public Category Category { get; set; }

        // Внешний ключ

        // ID Пользователя - обычное свойство

        public int? UserId { get; set; }

        // Отдел пользователя - Навигационное свойство

        public User User { get; set; }

        // Внешний ключ

        // ID Пользователя - обычное свойство

        public int? ExecutorId { get; set; }

        // Отдел пользователя - Навигационное свойство

        public User Executor { get; set; }

        // Внешний ключ

        // ID жизненного цикла заявки - обычное свойство

        public int LifecycleId { get; set; }

        // Ссылка на жизненный цикл заявки - Навигационное свойство

        public Lifecycle Lifecycle { get; set; }

    }

}

Большинство свойств, я думаю, понятны, но все же несколько свойств я прокомментирую подробнее. public string File { get; set; } - данное свойство у нас буде хранить путь к файлу изображения с ошибкой (при наличии).

Теперь в файл с классом Request добавим класс Lifecycle, который будет отвечать за жизненный цикл заявки:

// Модель жизенного цикла заявки

    public class Lifecycle

    {

        // ID

        public int Id { get; set; }

        // Дата открытия

        [DisplayFormat(DataFormatString = "{0:dd/MM/yyyy H:mm:ss}", ApplyFormatInEditMode = true)]

        [DataType(DataType.Date)]

        public DateTime Opened { get; set; }

        // Дата распределения

        [DisplayFormat(DataFormatString = "{0:dd/MM/yyyy H:mm:ss}", ApplyFormatInEditMode = true)]

        [DataType(DataType.Date)]

        public DateTime? Distributed { get; set; }

        // Дата обработки

        [DisplayFormat(DataFormatString = "{0:dd/MM/yyyy H:mm:ss}", ApplyFormatInEditMode = true)]

        [DataType(DataType.Date)]

        public DateTime? Proccesing { get; set; }

        // Дата проверки

        [DisplayFormat(DataFormatString = "{0:dd/MM/yyyy H:mm:ss}", ApplyFormatInEditMode = true)]

        [DataType(DataType.Date)]

        public DateTime? Checking { get; set; }

        // Дата закрытия

        [DisplayFormat(DataFormatString = "{0:dd/MM/yyyy H:mm:ss}", ApplyFormatInEditMode = true)]

        [DataType(DataType.Date)]

        public DateTime? Closed { get; set; }

    }

Деление на этапы жизненного цикла довольно условно. Ну вот я выделил такие этапы, и каждый этап увязан с датой перехода к данному этапу. Предполагается, что сначала при создании у заявки статус 'Открыто', затем ее модератор распределяет по исполнителям. Исполнители в свою очередь при начале выполнения устанавливают у заявки статус 'В процессе' (Proccesing), затем при окончании проверяют (статус Checking), и когда все уже сделано, закрывают ее.

Для свойств с типом DateTime дополнительно атрибутом определяется формат даты, поскольку система может неправильно работать с датой. А таким образом мы даем детальное указание, как дату надо интерпретировать.

Также в файл класса заявки добавим два вспомогательных перечисления:

// Перечисление для статуса заявки

public enum RequestStatus

{

    Open = 1,

    Distributed = 2,

    Proccesing = 3,

    Checking = 4,

    Closed = 5

}

// Перечисление для приоритета заявки

public enum RequestPriority

{

    Low = 1,

    Medium = 2,

    High = 3,

    Critical = 4

}

Перечисление RequestStatus содержи все статусы заявки. Оно увязывается со свойством Status у класса Request.

Перечисление RequestPriority содержит все возможные приоритеты, которые пользователь может присвоить заявке: низкий, средний, высокий, критический. Для установки приоритета в классе Request предназначено свойство Priority.

Остальные классы

Теперь добавим все остальные классы сущностей. Эти классы будут выглядеть намного проще. Итак, добавим в папку Models класс Category:

public class Category

{

    public int Id { get; set; }

    [Required]

    [Display(Name = "Название категории")]

    [MaxLength(50, ErrorMessage = "Превышена максимальная длина записи")]

    public string Name { get; set; }

}

Эта модель будет указывать на категорию проблемы, например, проблема с оформлением документов

Теперь добавим модель отдела Department:

public class Department

{

    public int Id { get; set; }

    [Required]

    [Display(Name = "Название отдела")]

    [MaxLength(50, ErrorMessage = "Превышена максимальная длина записи")]

    public string Name { get; set; }

}

Тут же определим связанный с этим класс Activ:

// Модель Активы

public class Activ

{

    public int Id { get; set; }

    // номер кабинета

    [Required]

    [Display(Name = "Номер кабинета")]

    [MaxLength(50, ErrorMessage = "Превышена максимальная длина записи")]

    public string CabNumber { get; set; }

    // Внешний ключ

    // ID Отдела - обычное свойство

    [Required]

    [Display(Name = "Отдел")]

    public int? DepartmentId { get; set; }

    // Отдел - Навигационное свойство

    public Department Department { get; set; }

}

Эта модель будет представлять актив - кабинет, принадлежащий определенному отделу. В процессе заполнения заявки пользователь сможет указать кабинет, в котором происходит проблема.

Таким образом, мы создали весь набор моделей, которые будут использоваться в приложении, и можем перейти к самому неинтересному этапу - созданию БД и таблиц.

**Создание системы авторизации**

**Создание провайдера ролей**

Первым делом создадим систему авторизации. Итак, у нас в приложении будут определены 4 типа пользователей: администратор, модератор, исполнитель и обычный пользователь.

Администратор будет управлять системой пользователей и добавлять различные данные в справочники оборудования, в общем, будет нести стандартные управленческие функции.

Модератор будет назначать заявкам пользователей исполнителей, которые затем будут решать данную проблему.

Исполнитель будет определенным образом решать возникшую задачу и затем обновлять статус этой задачи. При завершении выполнения исполнитель закрывает задачу.

И обычные пользователи просто формируют заявки по возникшим у них проблемам.

Деление на данные типы довольно условно, можно сделать меньше, можно, наоборот, добавить дополнительных типов.

Каждый тип пользователей будет обладать определенными ограничениями. Так, нам надо сделать, чтобы создавать и удалять пользователей мог только администратор, но никак не модератор, не говоря уже об обычных пользователях.

Также, например, нам надо, чтобы только модератор мог назначать исполнителей по заявкам, а не сами исполнители. Ну и прочие другие моменты, которые мы впоследствии применим в приложении.

Итак, добавим в проект папку Providers, в которую затем мы добавим классы провайдеров пользователей и ролей.

Добавим в нее класс провайдера ролей:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Security;

using System.Web.Helpers;

using System.Web.WebPages;

using Microsoft.Internal.Web.Utils;

using HelpDeskTrain.Models;

namespace HelpDeskTrain.Providers

{

    public class HelpdeskRoleProvider : RoleProvider

    {

        public override string[] GetRolesForUser(string login)

        {

            string[] role = new string[] { };

            using (HelpdeskContext \_db = new HelpdeskContext())

            {

                try

                {

                    // Получаем пользователя

                    User user = (from u in \_db.Users

                                 where u.Login == login

                                 select u).FirstOrDefault();

                    if (user != null)

                    {

                        // получаем роль

                        Role userRole = \_db.Roles.Find(user.RoleId);

                        if (userRole != null)

                        {

                            role = new string[] { userRole.Name };

                        }

                    }

                }

                catch

                {

                    role = new string[] { };

                }

            }

            return role;

        }

        public override bool IsUserInRole(string username, string roleName)

        {

            bool outputResult = false;

            // Находим пользователя

            using (HelpdeskContext \_db = new HelpdeskContext())

            {

                try

                {

                    // Получаем пользователя

                    User user = (from u in \_db.Users

                                 where u.Login == username

                                 select u).FirstOrDefault();

                    if (user != null)

                    {

                        // получаем роль

                        Role userRole = \_db.Roles.Find(user.RoleId);

                        //сравниваем

                        if (userRole != null && userRole.Name == roleName)

                        {

                            outputResult = true;

                        }

                    }

                }

                catch

                {

                    outputResult = false;

                }

            }

            return outputResult;

        }

        public override void AddUsersToRoles(string[] usernames, string[] roleNames)

        {

            throw new NotImplementedException();

        }

        public override void CreateRole(string roleName)

        {

            throw new NotImplementedException();

        }

        public override string ApplicationName

        {

            get

            {

                throw new NotImplementedException();

            }

            set

            {

                throw new NotImplementedException();

            }

        }

        public override bool DeleteRole(string roleName, bool throwOnPopulatedRole)

        {

            throw new NotImplementedException();

        }

        public override string[] FindUsersInRole(string roleName, string usernameToMatch)

        {

            throw new NotImplementedException();

        }

        public override string[] GetAllRoles()

        {

            throw new NotImplementedException();

        }

        public override string[] GetUsersInRole(string roleName)

        {

            throw new NotImplementedException();

        }

        public override void RemoveUsersFromRoles(string[] usernames, string[] roleNames)

        {

            throw new NotImplementedException();

        }

        public override bool RoleExists(string roleName)

        {

            throw new NotImplementedException();

        }

    }

}

Чтобы реализовать свой провайдер ролей, нам надо наследовать класс от RoleProvider. В данном случае мы переопределяем два метода базового класса: GetRolesForUser, который возвращает все роли пользователя, и IsUserInRole, который показывает, связан ли пользователь с данной ролью.

Теперь подключим наш провайдер ролей к проекту в файле web.config в секции system.web и заменим:

<roleManager enabled="true" defaultProvider="HelpdeskRoleProvider">

  <providers>

    <add name="HelpdeskRoleProvider" type="HelpDeskTrain.Providers.HelpdeskRoleProvider" />

  </providers>

</roleManager>

При входе пользователя на сайт его работа буде начинаться с авторизации, и первое что пользователь увидит - это страница авторизации. А за всю логику авторизации будет отвечать контроллер AccountController. Поэтому добавим в проект в папку Controllers класс AccountController:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

using System.Web.Security;

using HelpDeskTrain.Models;

namespace HelpDeskTrain.Controllers

{

    [AllowAnonymous]

    public class AccountController : Controller

    {

        public ActionResult Login()

        {

            return View();

        }

        [HttpPost]

        public ActionResult Login(LogViewModel model, string returnUrl)

        {

            if (ModelState.IsValid)

            {

                if (ValidateUser(model.UserName, model.Password))

                {

                    FormsAuthentication.SetAuthCookie(model.UserName, model.RememberMe);

                    if (Url.IsLocalUrl(returnUrl))

                    {

                        return Redirect(returnUrl);

                    }

                    else

                    {

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

                    }

                }

                else

                {

                    ModelState.AddModelError("", "Неправильный пароль или логин");

                }

            }

            return View(model);

        }

        public ActionResult LogOff()

        {

            FormsAuthentication.SignOut();

            return RedirectToAction("Login", "Account");

        }

        private bool ValidateUser(string login, string password)

        {

            bool isValid = false;

            using (HelpdeskContext \_db = new HelpdeskContext())

            {

                try

                {

                    User user = (from u in \_db.Users

                                 where u.Login == login && u.Password==password

                                 select u).FirstOrDefault();

                    if (user != null)

                    {

                        isValid = true;

                    }

                }

                catch

                {

                    isValid = false;

                }

            }

            return isValid;

        }

    }

}

Поскольку обращаться к контроллеру может любой, то для него установим атрибут [AllowAnonymous], который открывает анонимный доступ.

В контроллере определено действие Login, которое авторизует пользователя. GET-вариант метода выдает страницу авторизации. А POST-версия обрабатывает введенные данные и, в зависимости от того, был ли найден в базе данных пользователь с подобными логином и паролем, авторизует его, устанавливая соответствующие куки.

На метод RedirectToAction("Index", "Request"); пока можно не обращать внимания. Чуть позже мы определим контроллер RequestContoller, на который в данном случае будет идти переадрессация.

Метод Login использует вспомогательный метод ValidateUser, который служит для валидации пользователя. В случае, если бы у нас использовался провайдер членства, то мы бы в нем определили этот метод. А в данном случае мы определяем его в контроллере.

Также, для упрощения работы с данными в методе Login используется модель LogViewModel, которая и будет передавать в контроллер введенные пользователем данные. Добавим класс этой модели в папку Models:

using System.ComponentModel.DataAnnotations;

namespace HelpDeskTrain.Models

{

    public class LogViewModel

    {

        [Required]

        [Display(Name = "Логин")]

        public string UserName { get; set; }

        [Required]

        [DataType(DataType.Password)]

        [Display(Name = "Пароль")]

        public string Password { get; set; }

        [Required]

        [Display(Name = "Запомнить?")]

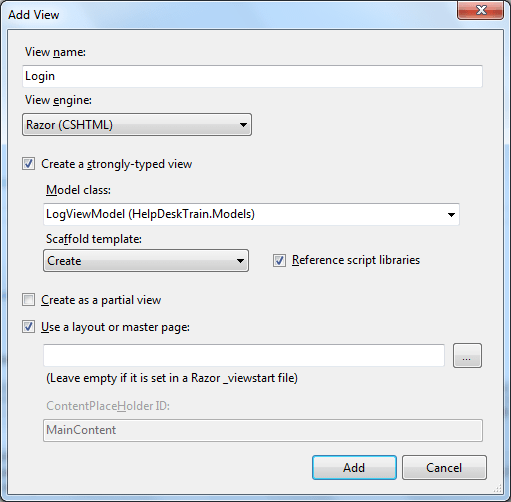
        public bool RememberMe { get; set; }

    }

}

Cкомпилируем проект с помощью функции Build, чтобы нам стали доступными все созданные модели.

Теперь добавим в проект первое представление, которое будет предназначено для авторизации. Это будет строго типизированное представление - типизировано оно будет моделью LogViewModel. Итак, добавим в проект представление Login:



Для облегчения работы используем в настройках шаблон Create, который нам создаст шаблонное представление. А мастер-страница layout в данном случае использоваться не будет.

Немного изменим стандартное представление следующим образом:

@model HelpDeskTrain.Models.LogViewModel

@{

Layout = null;

}

<!DOCTYPE html>

<html>

<head>

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

<link href="~/Content/Site.css" rel="stylesheet" type="text/css" />

<title>Вход в систему</title>

</head>

<body>

<script src="~/Scripts/jquery-1.7.1.min.js"></script>

<script src="~/Scripts/jquery.validate.min.js"></script>

<script src="~/Scripts/jquery.validate.unobtrusive.min.js"></script>

<h2>Вход в систему</h2>

<p>Введите свой логин и пароль для входа в систему</p>

@using (Html.BeginForm()) {

@Html.ValidationSummary(true)

<fieldset>

<legend>Войти в систему</legend>

<div class="editor-label">

@Html.LabelFor(model => model.UserName)

</div>

<div class="editor-field">

@Html.EditorFor(model => model.UserName)

@Html.ValidationMessageFor(model => model.UserName)

</div>

<div class="editor-label">

@Html.LabelFor(model => model.Password)

</div>

<div class="editor-field">

@Html.EditorFor(model => model.Password)

@Html.ValidationMessageFor(model => model.Password)

</div>

<div class="editor-label">

@Html.LabelFor(model => model.RememberMe)

</div>

<div class="editor-field">

@Html.EditorFor(model => model.RememberMe)

</div>

<p>

<input type="submit" value="Войти" />

</p>

</fieldset>

}

</body>

</html>

И при входе пользователь увидит:

