

ВТУ “Св. Св. Кирил и Методий”

Факултет “Математика и Информатика”

КУРСОВА РАБОТА

П0 ДИСЦИПЛИНАТА

,,УЕБ ПРОГРАМИРАНЕ С .NET”

На тема:

СИСТЕМА ЗА УПРАВЛЕНИЕ НА СЪБИТИЯ

Изготвил: Проверил:

Пламенна Викторова Петрова доц. д-р.

Специалност: Златко Георгиев Софтуерно инженерство Върбанов

Факултетен № 1909011132

Велико Търново

2020

Съдържание:

1. Цел на разработката…………………………………………………………………………………………..3
2. Процес на разработката…………………………………………………………………………............4

Стъпка 1. Еmpty Visual Studio Solution…………………………………………………………..4

Стъпка 2. Еmpty MVC Project with User Accounts…………………………………………..4

Стъпка 3. Персонализиране на MVC приложението…………………………………….4

Стъпка 4. Отделяне на EF Data Model от MVC Application…………………………….5

Стъпка 5. Дефиниране на Data Model…………………………….................................6

Стъпка 7. Попълваме Sample Data в базата дании – сийдваме базата………..9

Стъпка 8. Извеждаме всички събития в списъчен вид в хоум страницата..12

Стъпка 9. Извеждане на списък на събития с AJAX........................................17

Стъпка 10. Create New Event............................................................................22

Стъпка 11. Notification System (Info / Error Messages)....................................26

Стъпка 12. Date / Time / Duration UI Controls.................................................28

Стъпка 13. Client-Side Unobtrusive Validation..................................................30

Стъпка 14. Cписък на моите събития (List My Events)....................................30

Стъпка 15. Edit Existing Event Form..................................................................32

Стъпка 16. Edit Existing Event Logic..................................................................33

Стъпка 17. Handling HTML Special Characters..................................................35

Стъпка 18. Delete Existing Event Form and Logic..............................................35

1. Визуализация на приложението………………………………………………………………………38

**1. Цел на разработката**

Целта на разработката е да се създаде уеб приложение, базирано на ASP.NET MVC data-driven Web applications. MVC проектът дава възможност да за създаване, редактиране и изтриване на събития както и и за представянето им под формата на списък. IDE-то за разработка, което се използва за реализацията на това приложение, е Visual Studio 2019 с най-новите налични актуализации + SQL Server 2017 Express.

Дизайнът и имплементацията на уеб базираната система за управление на събития включва :

* Събитията имат заглавие, начална дата и по желание начален час. Събитията могат също да имат (по избор) продължителност, описание, местоположение и автор. Събития могат да бъдат публични (видими от всички) и частни (видими само за техния собственик автор). Събитията може да имат коментари. Коментарите принадлежат към определено събитие и имат съдържание (текст), дата и незадължително автор (собственик).
* Анонимните потребители (без вход) могат да преглеждат всички публични събития. Началната страница показва всички публични събития, в две групи: предстоящи и минали. Събитията се показват в кратка форма (заглавие, дата, продължителност, автор и местоположение) и имат бутон [View Details], който се зарежда динамично (от AJAX) - тяхното описание, коментари и бутони [Edit] / [Delete].
* Анонимните потребители могат да се регистрират в системата и да влязат / излязат. Потребителите трябва да имат задължителни имейл, парола и пълно име. Имейлът на потребителя трябва да бъде уникален. Потребителската парола не трябва да бъде празна, но може да съдържа само един знак.
* Потребителите, които са влезли в системата, трябва да могат да преглеждат собствените си събития, да създават нови събития, да редактират свои собствени събития и да изтриват собствените си събития. Изтриването на събития изисква потвърждение. Имплементирани са валидиране на данни от страна на клиент и сървърна страна.
* Специалният потребител „admin@admin.com“ трябва има ролята „Администратор“ и има пълни разрешения за редактиране / изтриване на събития и коментари.

**2. Процес на разработката**

Стъпка 1. Empty Visual Studio Solution

Създава се ново празно решение за Visual Studio, наречено „Event-Lab“. Това решение за VS съдържа проектите : проект за слой данни и проект за ASP.NET уеб приложение.

## Стъпка 2. Empty MVC Project with User Accounts

Създава се празен ASP.NET MVC проект с подразбиращия се шаблон на Visual Studio, който има вградени акаунти и удостоверяване. Добавя се с Add -> New Project. Избираме Visual C# -> Web -> ASP.NET Web Application. Кръщаваме приложението “Events.Web”. То съдържа контролерите, изгледите, view model-ите и др. Слоят за достъп до данни (entity класовете + Entity Framework data context) се отделя в друго приложение. Избираме шаблон MVC + Individual User Accounts за authentication type.

Стъпка 3. Персонализиране на MVC приложението

Променяме connectionString от файла Web.config – на Data Source поставяме . и на Initial Catalog – Events. Това ще създаде базата данни Events в MS SQL Server 2017 Express. Изтриваме файловете AssemblyInfo.cs, App\_Data, favico.ico, които не са нужни на приложението. Промяна на password validation policy. Модифицираме файла App\_Start\IdentityConfig.cs :

manager.PasswordValidator = new PasswordValidator

{

RequiredLength = 1,

RequireNonLetterOrDigit = false,

RequireDigit = false,

RequireLowercase = false,

RequireUppercase = false,

};

Модифицираме и минималната дължина на паролата във файла Models\AccountViewModel.cs :

[StringLength(100, ErrorMessage = "The {0} must be at least {2} characters long.", MinimumLength = 1)]

Модифицираме минималната дължина на паролата още в SetPasswordViewModel и ChangePasswordViewModel класовете, намиращи се в Models\ManageViewModels.cs

Можем да проверим местоположението на базата данни в MS SQL Server-а

Правим промени по оформлението на приложението. Намираме файла \Views\Shared\\_Layout.cshtml и написваме заглавие Public Events. Променяме и навигационните менюта. Целта е линковете MyEvents и CreateEvent да се виждат само за логнати посетители. Анонимните посетители ще могат да виждат само линка Events от хоум страницата, докато регистрираните посетители ще могат да виждат събитията, създадени от тях, и линка за създаване на нови събития.

<div class="navbar-collapse collapse">

<ul class="nav navbar-nav">

@if (Request.IsAuthenticated)

{

<li>@Html.ActionLink("My Events", "My", "Events")</li>

<li>@Html.ActionLink("Create Event", "Create", "Events")</li>

}

</ul>

@Html.Partial("\_LoginPartial")

</div>

Променяме и фуутъра :

<footer>

<p>&copy; @DateTime.Now.Year - Events Application</p>

</footer>

Премахваме джънк код от изгледа \Views\Home\Index.cshtml. Премахваме ненужните Action методи About и Contacts в HomeController-a. Изтриваме ненужните му изгледи About.cshtml и Contact.cshtml.

Стъпка 4. Отделяне на EF Data Model от MVC Application

Създаваме нов Class Library project Events.Data в solution-а във Visual Studio. Изтриваме ненужните файлове от Events.Data – AssemblyInfo.cs и Class1.cs. Преместваме Еvents.Web\Models\IdentityModels.cs в новия проект Events.Data. Извличаме класа ApplicationUser във файл ApplicationUser.cs. Променяме пространството от имена на Events.Data. Преименуваме IdentityModel.cs на ApplicationDbContext.cs. Този файл съдържа Entity Framework контекста на базата на данни. Променяме namespace-a на Еvents.Data. Right Click на Еvents.Data -> Manage NuGet Packages... Инсталираме пакетите Microsoft.AspNet.Identity.Core и Microsoft.AspNet.Identity.EntityFramework. Оправяме липсващите using-и. Добавяме референция от Events.Web до Еvents.Data за успешна компилация. Оправяме липсващите using-и в Events.Web проекта.

Стъпка 5. Дефиниране на Data Model

Първо, добавяме full name в класа ApplicationUser. То съдържа името на автора на събитието.

[Required]

public string FullName { get; set; }

Създаваме класа Events, който съхранява събитията и техните пропъртита.

namespace Events.Data

{

using System;

using System.Collections.Generic;

using System.ComponentModel.DataAnnotations;

public class Event

{

public Event()

{

this.IsPublic = true;

this.StartDateTime = DateTime.Now;

this.Comments = new HashSet<Comment>();

}

public int Id { get; set; }

[Required]

[MaxLength(200)]

public string Title { get; set; }

[Required]

public DateTime StartDateTime { get; set; }

public TimeSpan? Duration { get; set; }

public string AuthorId { get; set; }

public virtual ApplicationUser Author { get; set; }

public string Description { get; set; }

[MaxLength(200)]

public string Location { get; set; }

public bool IsPublic { get; set; }

public virtual ICollection<Comment> Comments { get; set; }

}

}

Създаваме и клас за коментари Comment :

namespace Events.Data

{

using System;

using System.ComponentModel.DataAnnotations;

public class Comment

{

public Comment()

{

this.Date = DateTime.Now;

}

public int Id { get; set; }

[Required]

public string Text { get; set; }

[Required]

public DateTime Date { get; set; }

public string AuthorId { get; set; }

public virtual ApplicationUser Author { get; set; }

public int EventId { get; set; }

[Required]

public virtual Event Event { get; set; }

}

}

Добавяне на коментари към класа Event става с this.Comments = new HashSet<Comment>();

Модифицираме класа ApplicationDbContext да съдържа събития и коментари чрез IDbSet<T>:

public IDbSet<Event> Events { get; set; }

public IDbSet<Comment> Comments { get; set; }

Стъпка 6. Конфигурация на Database Migrations

Конфигурираме автоматична миграционна стратегия за базата данни, за да се опростят промените в схемата на базата данни по време на развиването на проекта. В Package Manager Console с Default project – Events.Data написваме командата Enable-Migrations. Тази команда генерира файл Migrations\Configuration.cs в проекта Events.Data. Преименуваме файла на DbMigrationsConfig.cs -> internal sealed class DbMigrationsConfig. Променяме нивото на достъп на public и позволяваме автоматичните миграции.

AutomaticMigrationsEnabled = true;

AutomaticMigrationDataLossAllowed = true;

Поставяме database миграцията в инициализатора на базата данни в Global.asax файла.

Database.SetInitializer(new MigrateDatabaseToLatestVersion<ApplicationDbContext, DbMigrationsConfig>());

За да тръгне успешно приложението, дропваме базата данни MS SQL Server-a и се опитваме да се логнем наново. Рефрешваме базите данни и решаваме проблема с invalid login.

Променяме RegisterViewModel, Register.cshtml view и Register action в AccountController, като добавяме пропъртито FullName.

public class RegisterViewModel

{

[Required]

[EmailAddress]

[Display(Name = "Email")]

public string Email { get; set; }

[Required]

[Display(Name = "Full Name")]

public string FullName { get; set; }

...

}

<div class="form-group">

@Html.LabelFor(m => m.FullName, new { @class = "col-md-2 control-label" })

<div class="col-md-10">

@Html.TextBoxFor(m => m.FullName, new { @class = "form-control" })

</div>

</div>

var user = new ApplicationUser

{

UserName = model.Email,

Email = model.Email,

FullName = model.FullName

};

Стъпка 7. Попълваме Sample Data в базата данни – сийдваме базата

Шаблонните данни ще помогнат по време на разработката, за да улеснят тестването. Създаваме admin account и няколко събития с няколко коментари. Най-лесният начин да се направи това е да се създаде Seed() метод в стратегията за миграции на базата данни.

В метода Seed() проверява дали базата данни е празна. Ако е празна, попълваме sample data. Seed() методът ще се изпълни всеки път, когато приложението се стартира след промяна на схемата на базата данни.

// Seed initial data only if the database is empty

if (!context.Users.Any())

{

var adminEmail = "admin@admin.com";

var adminUserName = adminEmail;

var adminFullName = "System Administrator";

var adminPassword = adminEmail;

string adminRole = "Administrator";

CreateAdminUser(context, adminEmail, adminUserName, adminFullName,

adminPassword, adminRole);

CreateSeveralEvents(context);

}

Дифоултният администратор е [adnin@admin.com](mailto:adnin@admin.com) със същата парола. Това ще улесни тестването, защото някои actions изискват администраторски привилегии в системата.

Създаваме admin user и administrator role и assign administrator role за admin user:

private void CreateAdminUser(ApplicationDbContext context, string adminEmail, string adminUserName, string adminFullName, string adminPassword, string adminRole)

{

//Create the "admin" user

var adminUser = new ApplicationUser

{

UserName = adminUserName,

FullName = adminFullName,

Email = adminEmail

};

var userStore = new UserStore<ApplicationUser>(context);

var userManager = new UserManager<ApplicationUser>(userStore);

userManager.PasswordValidator = new PasswordValidator

{

RequiredLength = 1,

RequireNonLetterOrDigit = false,

RequireDigit = false,

RequireLowercase = false,

RequireUppercase = false,

};

var userCreateResult = userManager.Create(adminUser, adminPassword);

if (!userCreateResult.Succeeded)

{

throw new Exception(string.Join("; ", userCreateResult.Errors));

}

//Create the "Administrator role"

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

var roleCreateResult = roleManager.Create(new IdentityRole(adminRole));

if (!roleCreateResult.Succeeded)

{

throw new Exception(string.Join("; ", roleCreateResult.Errors));

}

//Add the "admin" user to "Administrator" role

var addAdminRoleResult = userManager.AddToRole(adminUser.Id, adminRole);

if (!addAdminRoleResult.Succeeded)

{

throw new Exception(string.Join("; ", addAdminRoleResult.Errors));

}

}

Създаваме събития с коментари :

private void CreateSeveralEvents(ApplicationDbContext context)

{

context.Events.Add(new Event()

{

Title = "Passed Event <Anonymous>",

StartDateTime = DateTime.Now.Date.AddDays(-2).AddHours(10).AddMinutes(30),

Duration = TimeSpan.FromHours(1.5),

Comments = new HashSet<Comment>() {

new Comment() { Text = "<Anonymous> comment"},

new Comment() { Text = "User comment", Author = context.Users.First() }

}

});

context.Events.Add(new Event()

{

Title = "Passed Event",

StartDateTime = DateTime.Now.Date.AddDays(-2).AddHours(12).AddMinutes(0),

Duration = TimeSpan.FromHours(2),

});

context.Events.Add(new Event()

{

Title = "ASP.NET MVC Lab",

StartDateTime = DateTime.Now.AddDays(3).AddHours(11).AddMinutes(30),

Author = context.Users.First(),

Location = "Veliko Tarnovo University (Veliko Tarnovo)",

Description = "This lab will focus on practical <ASP.NET MVC>" +

"Web application development",

Comments = new HashSet<Comment>() {

new Comment() { Text = "<Anonymous> comment"},

new Comment() { Text = "User comment", Author = context.Users.First() },

new Comment() { Text = "Another <user> comment ", Author = context.Users.First() }

}

});

context.Events.Add(new Event()

{

Title = "Party @ Home",

StartDateTime = DateTime.Now.Date.AddDays(5).AddHours(21).AddMinutes(30),

Author = context.Users.First(),

});

context.Events.Add(new Event()

{

Title = "Passed Event <Again>",

StartDateTime = DateTime.Now.Date.AddDays(-10).AddHours(18).AddMinutes(0),

Duration = TimeSpan.FromHours(2.5)

});

}

Добавяме малко код за създаване на още няколко събития :

* Няколко предстоящи събития
* Няколко минали събития
* Няколко анонимни събития (без автор)
* Събития с / без коментари

Дропваме базата данни и рестартираме уеб приложението

Проверяваме данни в базата данни Events. Трябва да съдържа информация в Events, Comments, AspNetUsers, AspNetRoles и AspNetUserRoles.

Стъпка 8. Извеждаме всички събития в списъчен вид в хоум страницата.

Списъчното извеждане на всички публични събития в хоум страницата преминава през няколко стъпки :

* Създаваме EventViewModel, който съдържа данните за събитията, които се извеждат на хоум страницата.
* Създаваме UpcomingPassedEventsViewModel, който съдържа списък на предстоящите и миналите събития.
* Написваме логиката, която зарежда предстоящите и миналите събития в подразбиращия се action на HomeController
* Написваме изглед, който показва събитията на хоум страницата

Създаваме клас Models\EventViewModel.cs да съдържа данните за показване на home page:

namespace Events.Web.Models

{

using Events.Data;

using System;

using System.Linq.Expressions;

public class EventViewModel

{

public int Id { get; set; }

public string Title { get; set; }

public DateTime StartDateTime { get; set; }

public TimeSpan? Duration { get; set; }

public string Author { get; set; }

public string Location { get; set; }

public static Expression<Func<Event, EventViewModel>> ViewModel

{

get

{

return e => new EventViewModel()

{

Id = e.Id,

Title = e.Title,

StartDateTime = e.StartDateTime,

Duration = e.Duration,

Author = e.Author.FullName,

Location = e.Location

};

}

}

}

}

Създаваме клас Models\UpcomingPassedEventsViewModel.cs, който съдържа списък на предстоящите и миналите събития.

namespace Events.Web.Models

{

using System.Collections.Generic;

public class UpcomingPassedEventsViewModel

{

public IEnumerable<EventViewModel> UpcomingEvents { get; set; }

public IEnumerable<EventViewModel> PassedEvents { get; set; }

}

}

Създаваме клас Controllers\BaseController.cs, който съдържа Entity Framework data context. Той ще се използва като базов (parent) клас за всички контролери в MVC приложението :

namespace Events.Web.Controllers

{

using Events.Data;

using Microsoft.AspNet.Identity;

using System.Web.Mvc;

[ValidateInput(false)]

public class BaseController : Controller

{

protected ApplicationDbContext db = new ApplicationDbContext();

}

}

Разширяваме BaseController от HomeController да наследи DB context :

public class HomeController : BaseController

{

public ActionResult Index()

{

var events = this.db.Events

.OrderBy(e => e.StartDateTime)

.Where(e => e.IsPublic)

.Select(EventViewModel.ViewModel);

var upcomingEvents = events.Where(e => e.StartDateTime > DateTime.Now);

var passedEvents = events.Where(e => e.StartDateTime <= DateTime.Now);

return View(new UpcomingPassedEventsViewModel() {

UpcomingEvents = upcomingEvents,

PassedEvents = passedEvents

});

}

}

Идеята на приложения горе код е да селектира всички публични събития, подредени по стартова дата и да ги трансформира от entity class Event към view model class EventViewModel и да ги раздели на две колекции: предстоящи и минали събития.

Най-накрая създаваме изгледа, който показва събитията на хоум страницата - \Views\Home\Index.cshtml

<div class=”row”>

@if (Model.UpcomingEvents.Any())

{

@Html.DisplayFor(x => x.UpcomingEvents)

}

else

{

<div class=”col-md-4 col-sm-6 col-xs-12”>No events</div>

}

</div>

Подобен код извежда и миналите събития

Трябва да се дефинира display template за EventViewModel класа. Използва се, когато Razor engine изпълнява кода: @Html.DisplayFor(x => x.UpcomingEvents). Създаваме display шаблона във файла \Views\Shared\DisplayTemplates\EventViewModel.cshtml

@model Events.Web.Models.EventViewModel

<div class=”col-md-4 col-sm-6 col-xs-12”>

<div class=”event-box”>

<h2>@Model.Title</h2>

<span class=”date”>@Model.StartDateTime</span>

@if (Model.Duration != null)

{

<span class=”duration”>

(@Model.Duration.Value.ToString(@”hh\:mm”) hours)

</span>

}

@if (Model.Author != null)

{

<div class=”author”>Author: @Model.Author</div>

}

@if (Model.Location != null)

{

<div class=”location”>Location: @Model.Location</div>

}

</div>

</div>

Добавяме CSS стилове, за да направим така, че UI да изглежда по-добре. Прилагаме следния код в края на файла \Content\Site.css:

.event-group-heading {

color: black;

background-color: lightgrey;

padding: 5px 10px;

border-radius: 5px;

}

.event-box {

color: black;

width: 100%;

background-color: lightgrey;

padding: 10px;

margin: 5px 0 5px 0;

word-wrap: break-word;

}

Рефрешваме хоум страницата на приложението, за да видим промените в стила

Оптимизираме HomeController. Той съдържа код, който зарежда Event обект в нов EventViewModel обект. Този код може да бъде преместен в класа EventViewModel, за да направи HomeController-a по-чист:

public static Expression<Func<Event, EventViewModel>> ViewModel

{

get

{

return e => new EventViewModel()

{

Id = e.Id,

Title = e.Title,

StartDateTime = e.StartDateTime,

Duration = e.Duration,

Author = e.Author.FullName,

Location = e.Location

};

}

}

Статичното пропърти се използва .Select(...) заявки, за да трансформира Event в EventViewModel. Такива трансоформации работят по-добре капсулирани във view model класа. Сега HomeController може да използва статичното пропърти :

public ActionResult Index()

{

var events = this.db.Events

.OrderBy(e => e.StartDateTime)

.Where(e => e.IsPublic)

.Select(EventViewModel.ViewModel);

}

Стъпка 9. Извеждане на списък на събитията с AJAX

Тази стъпка в разработката на Events management приложението цели извеждането на детайли за събитията динамично с AJAX

Първо, дефинираме класа EventDetailsViewModel, който ще съдържа детайли за събитията :

namespace Events.Web.Models

{

using Events.Data;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Linq.Expressions;

public class EventDetailsViewModel

{

public int Id { get; set; }

public string Description { get; set; }

public string AuthorId { get; set; }

public IEnumerable<CommentViewModel> Comments { get; set; }

public static Expression<Func<Event, EventDetailsViewModel>> ViewModel

{

get

{

return e => new EventDetailsViewModel()

{

Id = e.Id,

Description = e.Description,

Comments = e.Comments.AsQueryable().Select(CommentViewModel.ViewModel),

AuthorId = e.Author.Id

};

}

}

}

}

Той съдържа вътрешна референция към класа CommentViewModel :

namespace Events.Web.Models

{

using Events.Data;

using System;

using System.Linq.Expressions;

public class CommentViewModel

{

public string Text { get; set; }

public string Author { get; set; }

public static Expression<Func<Comment, CommentViewModel>> ViewModel

{

get

{

return c => new CommentViewModel()

{

Text = c.Text,

Author = c.Author.FullName

};

}

}

}

}

След това пишем AJAX controller action, който ще връща детайли за събитията, когато потребителите кликнат върху бутона [View Details] :

public ActionResult EventDetailsById(int id)

{

var currentUserId = this.User.Identity.GetUserId();

var isAdmin = this.IsAdmin();

var eventDetails = this.db.Events

.Where(e => e.Id == id)

.Where(e => e.IsPublic || isAdmin || (e.AuthorId != null && e.AuthorId == currentUserId))

.Select(EventDetailsViewModel.ViewModel)

.FirstOrDefault();

var isOwner = (eventDetails != null && eventDetails.AuthorId != null &&

eventDetails.AuthorId == currentUserId);

this.ViewBag.CanEdit = isOwner || isAdmin;

return this.PartialView("\_EventDetails", eventDetails);

}

Гореописаната логика не е много праволинейна, защото детайлите за събитията трябва да бъдат показани само когато потребителят има разрешение за достъп до тях :

* Авторът на събитието може да преглежда и променя собствените си събития (дори и частните)
* Системните администратори могат да преглеждат и променят всички събития (дори и частните)
* Публичните събития са видими за всички, но не могат да се променят от всекиго

Добра идея е да се извлече и провери дали текущият потребител е администратор в BaseController :

public bool IsAdmin()

{

var currentUserId = this.User.Identity.GetUserId();

var isAdmin = (currentUserId != null && this.User.IsInRole("Administrator"));

return isAdmin;

}

След това създаваме частичен изглед, който ще бъде върнат, когато се прави AJAX request за детайли на дадено събитие. Този частичен изглед се намира в \Views\Home\\_EventDetails.cshtml

Първо, показваме описанието :

@if (Model.Description != null)

{

<div class="description">Description: @Model.Description</div>

}

След това извеждаме коментарите като списък :

@if (Model.Comments.Any())

{

@:Comments:

<ul>

@foreach (var comment in Model.Comments)

{

<li>

@comment.Text

@if (comment.Author != null)

{

@: (by @comment.Author)

}

</li>

}

</ul>

}

else

{

<p>No comments</p>

}

Накрая показваме бутоните [Edit] и [Delete], когато потребителят има edit permissions :

@if (ViewBag.CanEdit)

{

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

new { @class = "btn btn-defualt"})

<span></span>

@Html.ActionLink("Delete", "Delete", "Events", new { id = Model.Id },

new { @class = "btn btn-default" })

}

Променяме display template-а на събитията в \View\Shared\DisplayTemplates\EventViewModel.cshtml и добавяме код, който да извиква AJAX:

<div id = “event-details-@Model.Id”>

@Ajax.ActionLink(“View Details >>”, “EventDetailsById”, “Home”,

new { id = Model.Id },

new AjaxOptions

{

InsertionMode = InsertionMode.Replace,

UpdateTargetId = “event-details-” + Model.Id

}, new { @class = “btn btn-default” })

</div>

Инсталираме NuGet package – Microsoft.jQuery.Unobtrusive.Ajax в проекта Events.Web – скрипт, който ще поддържа извикванията на AJAX. Регистрираме Unobtrusive AJAX script-a в bundles configuration файла App\_Start\BundleConfig.cs, точно след регистрацията на jQuery bundle :

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

"~/Scripts/jquery-{version}.js"));

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

"~/Scripts/jquery.unobtrusive-ajax.min.js"));

Включваме unobtrusive AJAX script в изглед, който се нуждае от AJAX : \Views\Home\Index.cshtml :

@section scripts {

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

}

Този код ще изпълни бъндъла “~/bundles/ajax” в секцията scripts в layout-a на сайта :

@RenderSection("scripts", required: false)

Сега функционалността на AJAX започва да работи правилно :

Стъпка 10. Create New Event

Следващата функционалност, която трябва да се имплементира в Системата за управление на събития е “Create New Event”. Това изисква създаването на “New Event” форма (Razor view) + input model за формата + controller action, който ще поддържа получената форма с данни

Дефинираме input model за create и edit event form. Той ще съдържа всички event пропъртита, които потребителят ще попълни, когато създава или променя събитие. Създаваме класа Models\EventInputModel.cs :

namespace Events.Web.Models

{

using Events.Data;

using System;

using System.ComponentModel.DataAnnotations;

public class EventInputModel

{

[Required(ErrorMessage = "Event title is required.")]

[StringLength(200, ErrorMessage = "The {0} must be between {2} and {1} characters long.",

MinimumLength = 1)]

[Display(Name = "Title \*")]

public string Title { get; set; }

[DataType(DataType.DateTime)]

[Display(Name = "Date and Time \*")]

public DateTime StartDateTime { get; set; }

public TimeSpan? Duration { get; set; }

public string Description { get; set; }

[MaxLength(200)]

public string Location { get; set; }

[Display(Name = "Is Public?")]

public bool IsPublic { get; set; }

}

}

Закачаме валидационни анотации като [Required] и [MaxLength] за всяко пропърти, за да се улесни валидацията на формата.

Създаваме формата “New Event”. Използваме Razor view generator на Visual Studio. Създаваме папка \Views\Events. Right click на папката Events и избираме Add -> View. Записваме име на изгледа Create. Избираме шаблон Create и model class EventInputModel (Events.Web.Models). Използваме само layout page. Натискаме Add, за да генерираме изгледа.

Visual Studio генерира формата Create Event :

@model Events.Web.Models.EventInputModel

@{

ViewBag.Title = "Create New Event";

}

<h2>@ViewBag.Title</h2>

<hr />

@using (Html.BeginForm())

{

@Html.AntiForgeryToken()

<div class="form-horizontal">

@Html.ValidationSummary(true, "", new { @class = "text-danger" })

<div class="form-group">

@Html.LabelFor(model => model.Title, htmlAttributes: new { @class = "control-label col-md-2" })

<div class="col-md-10">

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

@Html.ValidationMessageFor(model => model.Title, "", new { @class = "text-danger" })

</div>

</div>

<div class="form-group">

@Html.LabelFor(model => model.StartDateTime, htmlAttributes: new { @class = "control-label col-md-2" })

<div class="col-md-10">

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

@Html.ValidationMessageFor(model => model.StartDateTime, "", new { @class = "text-danger" })

</div>

</div>

…

Къстомизираме генерираната форма. Променяме заглавието на Create Event и премахваме линка Back to List. Добавяме [Cancel] линк към My Event точно след бутона [Create].

Създаваме EventsController, който поддържа action-ите, които се отнасят до събитията : create / edit / delete / list events. Избираме MVX 5 Controller – Empty.

Добавяме Create action (HTTP GET), който ще покаже New Event формата :

namespace Events.Web.Controllers

{

public class EventsController : Controller

{

// GET : Event/Create

public ActionResult Create()

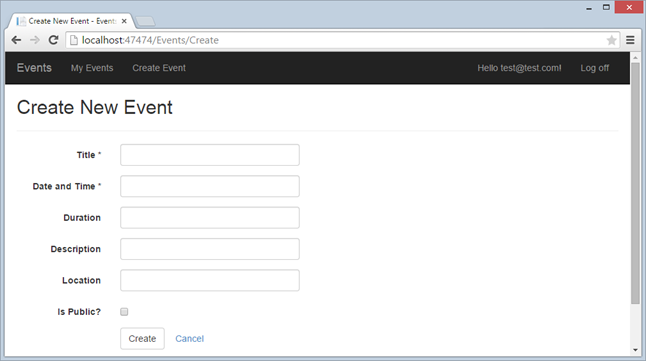
{

return View();

}

}

}



Дефинираме action My и изгледа зад него в EventsController:

public class EventsController : BaseController

{

// GET : Events/My

public ActionResult My()

{

return View();

}

}

Създаваме празен изглед \Views\Events\My.cshtml за по-нататъшна имплементация

Правим check box-a на New Event формата checked по подразбиране :

<div class="form-group">

@Html.LabelFor(model => model.IsPublic, htmlAttributes: new { @class = "control-label col-md-2" })

<div class="col-md-10">

<div class="checkbox">

@Html.CheckBoxFor(model => model.IsPublic, htmlAttributes: new { @checked = "true" })

@Html.ValidationMessageFor(model => model.IsPublic, "", new { @class = "text-danger" })

</div>

</div>

</div>

Стъпка 11. Notification System (Info / Error Messages)

По принцип MVC controller action-ите, които модифицират DB обекти например EventsController.Create работят по следния начин :

* Проверка на състоянието на модела. Ако валидацията се провали, състоянието на модела ще бъде невалидно и контролерът ще изпълни наново формата. Формата ще покаже грешките за всяко некоректно поле.
* Ако състоянието на модела е правилно, се създава / променя обекта на базата данни зад формата.
* Добавяне на нотификационно съобщение, което трябва да бъде показано в горната част на следващата страница, изведена в уеб приложението.
* Пренасочване към друга страница, която извежда като списък създадени / модифицирани DB обекти.

Липсваща част в ASP.NET MVC e нотификационната система, така че разработчиците трябва или да я създадат, или да използват някой NuGet package, който предава нотификационни съобщения в ASP.NET MVC.

От package management console във Visual Studio инсталираме NuGet пакета BootstrapNotifications в проекта Events.Web. NuGet ще добави класа Extensions\NotificationExtensions.cs. NuGet ще добави също и частичен изглед \Views\Shared\\_Notifications.cshtml

За да изведем нотификационни съобщения (когато има достъп) в горния край на всяка страница в MVC проекта, изпълняваме \_Notifications partial view в site layout, точно преди @RenderBody() :

<div class="container body-content">

@Html.Partial("\_Notifications")

@RenderBody()

<hr />

<footer>

<p>&copy; @DateTime.Now.Year - Events Application</p>

</footer>

</div>

Добавяме нотификации в controller actions след успешна промяна на базата данни.

Използваме NotificationType.INFO за информационни съобщения и NotificationType.ERROR за съобщения за грешка. Добавяме нотификационни съобщения след метода db.SaveChanges() – които променят базата данни, последвани от RedirectToAction(...) :

public ActionResult Create(EventInputModel model)

{

if (model != null && this.ModelState.IsValid)

{

var e = new Event()

{

AuthorId = this.User.Identity.GetUserId(),

Title = model.Title,

StartDateTime = model.StartDateTime,

Duration = model.Duration,

Description = model.Description,

Location = model.Location,

IsPublic = model.IsPublic

};

this.db.Events.Add(e);

this.db.SaveChanges();

//Display notification message "Event created."

this.AddNotification("Event created.", NotificationType.INFO);

return this.RedirectToAction("My");

}

return this.View(model);

}

Стилизираме в \Content\Site.css :

.alert {

margin-top: 10px;

}

Стъпка 12. Date / Time / Duration UI Controls

Формата New Event показва полета за постъпване на дата + час и продължителност. Добавяме date-time picker за полетата на данните и продължителността с цел постигане на по-добра визуализация.

Инсталираме NuGet package Bootstrap.v3.Datetimepicker.CSS. Той добавя следните файлове в MVC проекта :

* \Scripts\bootstrap-datetimepicker.js
* \Scripts\moment.js
* \Content\bootstrap-datetimepicker.css

Създаваме CSS и JavaScript bundles за date-time picker :

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

"~/Scripts/moment.js",

"~/Scripts/bootstrap-datetimepicker.js"));

bundles.Add(new StyleBundle("~/Content/bootstrap-datetimepicker").Include(

"~/Content/bootstrap-datetimepicker.min.css"));

Създаваме placeholder за CSS bundles в \_Layout.cshtml :

<head>

<meta charset="utf-8" />

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>@ViewBag.Title - Events</title>

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

@RenderSection("styles", required: false);

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

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

</head>

Добавяме date-time picker’s scripts и CSS файлове в New Event формата, в \Views\Create.cshtml:

@section scripts{

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

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

}

@section styles{

@Styles.Render("~/Content/bootstrap-datetimepicker")

}

Този код ще инжектира специализирания CSS от bundle-ите в header-a на страницата. Ще инжектира и специализирания JavaScript от bundle-ите в края на страницата. Добавяме скрипт за полето StartDateTime в \Views\Events\Create.cshtml :

<script>

$(function () {

$('#StartDateTime').datetimepicker({

format: 'DD-MMM-YYYY HH:mm',

sideBySide: true,

showTodayButton: true

});

});

</script>

Преместваме jQuery референцията в началото на HTML кода в \_Layout.cshtml :

<head>

<meta charset="utf-8" />

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>@ViewBag.Title - Events</title>

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

@RenderSection("styles", required: false);

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

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

</head>

По подобен начин добавяме time picker за Duration полето :

<script>

$(function () {

var date = new Date();

date.setHours(1);

date.setMinutes(0);

$('#Duration').datetimepicker({

format: 'HH:mm',

defaultDate: date,

showClear: true

});

});

</script>

Стъпка 13. Client-Side Unobtrusive Validation

New Event формата работи правилно, но има малък UI проблем : когато се въведат невалидни данни, валидацията на формата се изпълнява от страна на сървъра и потребителят вижда валидационните грешки след това, с малко закъснение. Добавяме валидационна форма от клиентска среда. Поставяме Microsoft jQuery Unobtrusive validation JavaScript bundle на страницата, в която се намира формата:

@section scripts{

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

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

}

@section styles{

@Styles.Render("~/Content/bootstrap-datetimepicker")

}

Стъпка 14. Списък на моите събития (List My Events)

Извеждаме събитията на текущия потребител на персоналната страница на събитията му : \Events\My

Първо, добавяме [Authorize] атрибут в EventsController :

[Authorize]

public class EventsController : BaseController

{

// GET: Events/Create

public ActionResult Create()

{

return View();

}

[Authorize] атрибутът ще пренасочи анонимните посетители към логин формата. Той дава правото за достъп на всички controller actions на EventsController да бъдат единствено от логнати потребители

Добавяме HTTP GET controller action My в EventsController, който ще покаже събитията на текущия посетител :

public ActionResult My()

{

string currentUserId = this.User.Identity.GetUserId();

var events = this.db.Events

.Where(e => e.AuthorId == currentUserId)

.OrderBy(e => e.StartDateTime)

.Select(EventViewModel.ViewModel);

var upcomingEvents = events.Where(e => e.StartDateTime > DateTime.Now);

var passedEvents = events.Where(e => e.StartDateTime <= DateTime.Now);

return View(new UpcomingPassedEventsViewModel()

{

UpcomingEvents = upcomingEvents,

PassedEvents = passedEvents

});

}

Накрая създаваме изглед My.cshtml зад по-горния action метод. Преизползваме кода. Първо извеждаме частичен изглед \Views\Shared\\_Events.cshtml, после го рефенсираме от \Views\Events\My.cshtml и още веднъж от \Views\Home\Index.cshtml :

@model Events.Web.Models.UpcomingPassedEventsViewModel

<h1 class="event-group-heading">Upcoming Events</h1>

<div class="row">

@if (Model.UpcomingEvents.Any())

{

@Html.DisplayFor(x => x.UpcomingEvents)

}

else

{

<div class="col-md-4 col-sm-6 col-xs-12">No events</div>

}

</div>

<h1 class="event-group-heading">Passed Events</h1>

<div class="row">

@if (Model.UpcomingEvents.Any())

{

@Html.DisplayFor(x => x.PassedEvents)

}

else

{

<div class="col-md-4 col-sm-6 col-xs-12">No events</div>

}

</div>

@model Events.Web.Models.UpcomingPassedEventsViewModel

@{

ViewBag.Title = "Public Events";

}

@Html.Partial("\_Events")

@section scripts {

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

}

@model Events.Web.Models.UpcomingPassedEventsViewModel

@{

ViewBag.Title = "My Events";

}

@Html.Partial("\_Events")

@section scripts {

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

}

Тестваме функционалността. Ако се опитаме да стигнем до My Events като анонимни потребители, приложението ще ни върне към login страницата :

Стъпка 15. Edit Existing Event Form

Функционалността Edit Event e много подобна на Create Event. Използва се същият input model като Events.Web.Models.UpcomingPassedEventsViewModel. Създаваме изглед \Views\Events\Edit.cshtml и преизползваме логиката от \Views\Events\Create.cshtml, като извеждаме частичния изглед \Views\Events\\_EventEditorForm :

@model Events.Web.Models.EventInputModel

@{

ViewBag.Title = "Edit Event";

}

<h2>@ViewBag.Title</h2>

<hr />

@Html.Partial("\_EventEditorForm")

@section scripts {

@\*@Scripts.Render("~/bundles/jqueryval")\*@

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

}

@section styles {

@Styles.Render("~/Content/bootstrap-datetimepicker")

}

Стъпка 16. Edit Existing Event Logic

Написваме controller action за промяна на събития в EventsController. Написваме HTTP GET action метод Edit, за да се подготви формата за промяна на събитие. В случай на невалидно event ID се показва съобщение за грешка и се пренасочва към MyEvents :

[HttpGet]

public ActionResult Edit(int id)

{

var eventToEdit = this.LoadEvent(id);

if(eventToEdit == null)

{

this.AddNotification("Cannot edit event #" + id,

NotificationType.ERROR);

return this.RedirectToAction("My");

}

var model = EventInputModel.CreateFromEvent(eventToEdit);

return this.View(model);

}

Логиката в LoadEvent(id) метода зарежда съществуващо събитие в случай че потребителят има правата да го промени :

private Event LoadEvent(int id)

{

var currentUserId = this.User.Identity.GetUserId();

var isAdmin = this.IsAdmin();

var eventToEdit = this.db.Events

.Where(e => e.Id == id)

.FirstOrDefault(e => e.AuthorId == currentUserId || isAdmin);

return eventToEdit;

}

Написваме HTTP POST action Edit, за да запазим промени след събмитване на формата за editing на събитията. Трябва първо да провери еvent ID и да покаже грешка в случай, че има несъществуващо събитие или липсващи права. След това проверява за валидационни грешки. В случай на валидационни грешки, същият формуляр се изпълнява отново (ще се покажат валидационните грешки). Накрая методът Edit модифицира базата данни и пренасочва към MyEvents :

[HttpPost]

[ValidateAntiForgeryToken]

public ActionResult Edit(int id, EventInputModel model)

{

var eventToEdit = this.LoadEvent(id);

if (eventToEdit == null)

{

this.AddNotification("Cannot edit event #" + id, NotificationType.ERROR);

return this.RedirectToAction("My");

}

if (model != null && this.ModelState.IsValid)

{

eventToEdit.Title = model.Title;

eventToEdit.StartDateTime = model.StartDateTime;

eventToEdit.Duration = model.Duration;

eventToEdit.Description = model.Description;

eventToEdit.IsPublic = model.IsPublic;

this.db.SaveChanges();

this.AddNotification("Event edited.", NotificationType.INFO);

return this.RedirectToAction("My");

}

return this.View(model);

}

Стъпка 17. Handling HTML Special Characters

Ако се опитаме да създадем събитие с име <New Event>, ще се провалим, защото по подразбиране ASP.NET MVC не позволява формулярите да изпращат HTML тагове в полетата за стойности. Това се прави с цел защита от Cross-Site Scripting (XSS) атаки. Оправяме го, като добавим [ValidateInput(false)] атрибут в BaseController :

[ValidateInput(false)]

public class BaseController : Controller

{

protected ApplicationDbContext db = new ApplicationDbContext();

...

}

Стъпка 18. Delete Existing Event Form

Създаваме формуляр и controller action, подобно на Edit Event. Формулярът трябва да зареди данните за събитието в read-only режим с [Confirm] и [Cancel] бутони. Логиката зад Delete Event също e подобна на Edit Event :

\Views\Events\Delete.cshtml :

@model Events.Web.Models.EventInputModel

@{

ViewBag.Title = "Delete Event?";

}

<h2>@ViewBag.Title</h2>

<hr />

@using (Html.BeginForm())

{

@Html.AntiForgeryToken()

<div class="form-horizontal">

<div class="form-group">

@Html.LabelFor(model => model.Title, htmlAttributes: new { @class = "control-label col-md-2" })

<div class="col-md-10">

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

@Html.ValidationMessageFor(model => model.Title, "", new { @class = "text-danger" })

</div>

</div>

<div class="form-group">

@Html.LabelFor(model => model.StartDateTime, htmlAttributes: new { @class = "control-label col-md-2" })

<div class="col-md-10">

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

@Html.ValidationMessageFor(model => model.StartDateTime, "", new { @class = "text-danger" })

</div>

</div>

<div class="form-group">

<div class="col-md-offset-2 col-md-10">

<input type="submit" value="Delete" class="btn btn-default" />

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

</div>

</div>

</div>

}

В EventsController :

[HttpGet]

public ActionResult Delete(int id)

{

var eventToDelete = this.LoadEvent(id);

if (eventToDelete == null)

{

this.AddNotification("Cannot delete event #" + id, NotificationType.ERROR);

return this.RedirectToAction("My");

}

var model = EventInputModel.CreateFromEvent(eventToDelete);

return this.View(model);

}

//[HttpPost]

//[ValidateAntiForgeryToken]

//public ActionResult Delete(int id, EventInputModel model)

//{

// var eventToDelete = this.LoadEvent(id);

// if (eventToDelete == null)

// {

// this.AddNotification("Cannot delete event #" + id, NotificationType.ERROR);

// return this.RedirectToAction("My");

// }

// this.db.Events.Remove(eventToDelete);

// this.db.SaveChanges();

// this.AddNotification("Event deleted.", NotificationType.INFO);

// return this.RedirectToAction("My");

//}

[HttpPost]

[ValidateAntiForgeryToken]

public ActionResult Delete(int id, EventInputModel model)

{

var eventToDelete = this.LoadEvent(id);

if (eventToDelete == null)

{

this.AddNotification("Cannot delete event #" + id, NotificationType.ERROR);

return this.RedirectToAction("My");

}

this.db.Events.Remove(eventToDelete);

this.db.SaveChanges();

this.AddNotification("Event deleted.", NotificationType.INFO);

return this.RedirectToAction("My");

}

private Event LoadEvent(int id)

{

var currentUserId = this.User.Identity.GetUserId();

var isAdmin = this.IsAdmin();

var eventToEdit = this.db.Events

.Where(e => e.Id == id)

.FirstOrDefault(e => e.AuthorId == currentUserId || isAdmin);

return eventToEdit;

}

}

3. Визуализация на приложението

