ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет програмування та комп’ютерних

і телекомунікаційних систем

Кафедра інженерії програмного забезпечення

**Лабораторна робота №**4

з дисципліни «Програмування в інтернет»

на тему:

«Моделі зі складною структурою. Визначення таблиць. Створення логіки додатка. Виконання операцій з моделями - редагування, видалення, додавання. Відношення один до багатьох, багато до багатьох»

**Виконав:**

студент 1 курсу, групи ІПЗс-20-1, варіант 8 Н.В.Савич

**Перевірив:**\_\_\_\_\_\_\_\_\_\_\_\_\_\_ О.М.Яшина

(підпис)

Хмельницький – 2020 р.

Мета. Отримати навики розробки Web - додатків зі складними моделями. При цьому необхідно враховувати значення навігаційної властивості, наявне у складній моделі також відношення один до багатьох, багато до багатьох.

Завдання

8. Розробити web – додаток «Служба страхування автомобілів».

Моделі:

Type.cs

public class Type

{

public int Id { get; set; }

public string Name { get; set; }

public int Price { get; set; }

public virtual ICollection<Order> Orders { get; set; }

public Type()

{

Orders = new List<Order>();

}

}

Order.cs

public class Order

{

public int OrderId { get; set; }

public string Person { get; set; }

public string Address { get; set; }

public virtual ICollection<Type> Types { get; set; }

public Order()

{

Types = new List<Type>();

}

public DateTime Date { get; set; }

}

Контролери:

HomeController.cs

using lab1.Models;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace lab1.Controllers

{

public class HomeController : Controller

{

TypeContext db = new TypeContext();

public ActionResult Index()

{

IEnumerable<Models.Type> types = db.Types;

ViewBag.Types = types;

return View();

}

public ActionResult Details(int id = 0)

{

Models.Type type = db.Types.Find(id);

if (type == null)

{

return HttpNotFound();

}

return View(type);

}

[HttpGet]

public ActionResult Buy(int id)

{

ViewBag.BookId = id;

return View();

}

[HttpPost]

public string Buy(Order order)

{

order.Date = DateTime.Now;

db.Orders.Add(order);

db.SaveChanges();

return "Спасибі," + order.Person + ", за покупку!";

}

}

}

TypesController.cs

public class TypesController : Controller

{

private TypeContext db = new TypeContext();

public ActionResult Index()

{

return View(db.Types.ToList());

}

public ActionResult Create()

{

return View();

}

[HttpPost]

[ValidateAntiForgeryToken]

public ActionResult Create([Bind(Include = "Id,Name,Price")] Models.Type type)

{

if (ModelState.IsValid)

{

db.Types.Add(type);

db.SaveChanges();

return RedirectToAction("Index");

}

return View(type);

}

public ActionResult Edit(int? id)

{

if (id == null)

{

return new HttpStatusCodeResult(HttpStatusCode.BadRequest);

}

Models.Type type = db.Types.Find(id);

if (type == null)

{

return HttpNotFound();

}

return View(type);

}

[HttpPost]

[ValidateAntiForgeryToken]

public ActionResult Edit([Bind(Include = "Id,Name,Price")] Models.Type type)

{

if (ModelState.IsValid)

{

db.Entry(type).State = EntityState.Modified;

db.SaveChanges();

return RedirectToAction("Index");

}

return View(type);

}

public ActionResult Delete(int? id)

{

if (id == null)

{

return new HttpStatusCodeResult(HttpStatusCode.BadRequest);

}

Models.Type type = db.Types.Find(id);

if (type == null)

{

return HttpNotFound();

}

return View(type);

}

[HttpPost, ActionName("Delete")]

[ValidateAntiForgeryToken]

public ActionResult DeleteConfirmed(int id)

{

Models.Type type = db.Types.Find(id);

db.Types.Remove(type);

db.SaveChanges();

return RedirectToAction("Index");

}

protected override void Dispose(bool disposing)

{

if (disposing)

{

db.Dispose();

}

base.Dispose(disposing);

}

}

}

Представлення:

Index.cshtml

@model IEnumerable<lab1.Models.Type>

@{

Layout = null;

}

<!DOCTYPE html>

<html>

<head>

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

<title>Index</title>

</head>

<body>

<p>

@Html.ActionLink("Створити новий тип", "Create")

</p>

<table class="table">

<tr>

<th>

@Html.DisplayNameFor(model => model.Name)

</th>

<th>

@Html.DisplayNameFor(model => model.Price)

</th>

<th></th>

</tr>

@foreach (var item in Model) {

<tr>

<td>

@Html.DisplayFor(modelItem => item.Name)

</td>

<td>

@Html.DisplayFor(modelItem => item.Price)

</td>

<td>

@Html.ActionLink("Редагувати", "Edit", new { id=item.Id }) |

@Html.ActionLink("Видалити", "Delete", new { id=item.Id })

</td>

</tr>

}

</table>

</body>

</html>

Delete.cshtml

@model lab1.Models.Type

@{

Layout = null;

}

<!DOCTYPE html>

<html>

<head>

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

<title>Видалення</title>

</head>

<body>

<h3>Ви впевненні, чи хочете видалити дане страхування</h3>

<div>

<h4>Type</h4>

<hr />

<dl class="dl-horizontal">

<dt>

@Html.DisplayNameFor(model => model.Name)

</dt>

<dd>

@Html.DisplayFor(model => model.Name)

</dd>

<dt>

@Html.DisplayNameFor(model => model.Price)

</dt>

<dd>

@Html.DisplayFor(model => model.Price)

</dd>

</dl>

@using (Html.BeginForm()) {

@Html.AntiForgeryToken()

<div class="form-actions no-color">

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

@Html.ActionLink("Назад до списку", "Index")

</div>

}

</div>

</body>

</html>

Edit.cshtml

@model lab1.Models.Type

@{

Layout = null;

}

<!DOCTYPE html>

<html>

<head>

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

<title>Edit</title>

</head>

<body>

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

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

@using (Html.BeginForm())

{

@Html.AntiForgeryToken()

<div class="form-horizontal">

<h4>Тип</h4>

<hr />

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

@Html.HiddenFor(model => model.Id)

<div class="form-group">

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

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

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

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

</div>

</div>

<div class="form-group">

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

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

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

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

</div>

</div>

<div class="form-group">

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

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

</div>

</div>

</div>

}

<div>

@Html.ActionLink("Назад до списку", "Index")

</div>

</body>

</html>

Details.cshtml

@using TeamMvc4.Models

@model Type

@{

ViewBag.Title = "Details";

}

<fieldset>

<legend>Информація про тип страхування</legend>

<div class="display-label"><b>Назва</b></div>

<div class="display-field">

@Html.DisplayFor(model => model.Name)

</div>

<div class="display-label"><b>Фамилия</b></div>

<div class="display-field">

@Html.DisplayFor(model => model.Price)

</div>

<div class="display-label"><b>Замовлення</b></div>

<ul>

@foreach (Order c in Model.orders)

{

<li>@c.Name</li>

}

</ul>

</fieldset>

Вигляд програми:

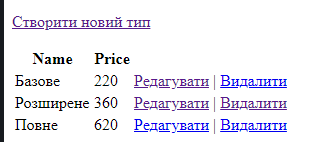


Рис 1.-Сторінка перегляду видів страхування

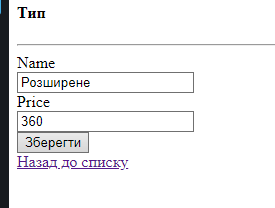


Рис 2.-Сторінка редагування

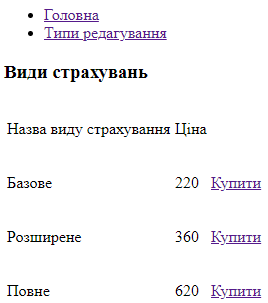


Рис 3.-Головна сторінка

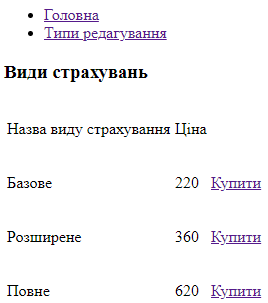


Рис 3.-Головна сторінка

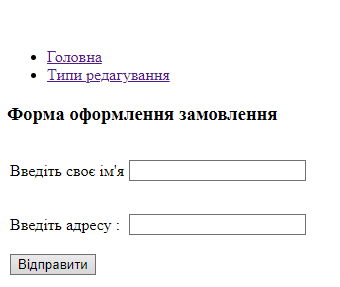


Рис 4.- Сторінка замовлення

Контрольні питання:

1. Призначення контролера.

Контролер є центральним компонентом в архітектурі MVC.Контролер отримує введення користувача, обробляє його і посилає назад результат обробки, наприклад, у вигляді представлення.

1. Звернення до контролера з веб-браузера.

Щоб звернутися до контролера з веб-браузера, нам треба в адресному рядку набрати адрес\_сайта / Імя\_контроллера /. Так, за запитом адрес\_сайта / Home / система маршрутизації за замовчуванням викличе метод Index контролера HomeController для обробки вхідного запиту. Якщо ми хочемо відправити запит до конкретного методу контролера, то потрібно вказувати цей метод явно: адрес\_сайта / Імя\_контроллера / Метод\_контроллера, наприклад, адрес\_сайта / Home / Buy - звернення до методу Buy контролера HomeController.

1. Методи дій.

Методидій (actionmethods) представляютьтакіметоди контролера, якіобробляютьзапити за певнимURL.

1. Зверненнядо методудій з веб-браузера.

Так як запити бувають різних типів, наприклад, GET і POST, фреймворк ASP.NET MVC дозволяє визначити тип оброблюваного запиту для дії, застосувавши до нього відповідний атрибут: [HttpGet], [HttpPost], [HttpDelete] або [HttpPut]. Так, дія Buy розбите на два методу, по одному для кожного типу запиту.

Однак не всі методи контролера є методами дій. Методи дій завжди мають модифікатор public. Закритих приватних методів дій не буває. Але контролер може також включати і звичайні методи, які можуть використовуватися в допоміжних цілях. наприклад,

[HttpPost]

public string Buy (Purchase purchase)

{

purchase.Date = getToday ();

db.Purchases.Add (purchase);

db.SaveChanges ();

return "Спасибі," + purchase.Person + ", за покупку!";

}

private DateTime getToday ()

{

return DateTime.Now;

}

Відповідно ми не можемо відправити з браузера запит Home / getToday /, тому що метод getToday не є методом дії.

1. Опціональний параметр.

Параметр, який є не обов’язковим і виконується за окремою вимогою користувача.

1. Значенняпараметрів за замовчуванням.

Ми також можемо поставити для параметрів значення за замовчуванням:

public string Square (int a = 10, int h = 3)

{

double s = a \* h / 2.0;

return "<h2> Площа трикутника з підставою" + a +

"І висотою" + h + "дорівнює" + s + "</ h2>";

}

В цьому випадку при запиті сторінки ми можемо вказати тільки один параметр або взагалі не вказувати (Home / Square? H = 5).

1. Отриманняданих з контексту запиту.

Отриманняданих з контексту запиту. Крім того, ми можемоотриматипараметри, та й не тількипараметри, але й іншідані, пов'язанііз запитом, з об'єктів контексту запиту. Нам доступнінаступніоб'єкти контексту Request. Об'єктRequestміститьколекціюParams, яка зберігаєвсіпараметри, передані в запити. І ми їхможемоотримати:

public string Square () {

int a = Int32.Parse (Request.Params ["a"]);

int h = Int32.Parse (Request.Params ["h"]);

double s = a \* h / 2; return "Площатрикутниказосновою" + a + "івисотою" + h + "дорівнює" + s + "";

}

1. Об’єкт Request.

Об'єкт Request містить колекцію Params, яка зберігає всі параметри, передані в запити. І ми їх можемо отримати:

1. Клас ActionResult.

ActionResult представляє собою абстрактний клас, в якому визначено один метод ExecuteResult, що перевизначається в наслідуваних класах: publicabstractclassActionResult {

public abstract void ExecuteResult(ControllerContextcontext);

}

1. Створення власних результатів дій.

Створимо свої результати дій. Вони будуть дуже простими. Візьмемо який-небудь проект, наприклад, з минулого розділу, і додамо в нього нову папку Util, яка буде містити нові класи. Після додавання папки додамо в неї перший клас. Назвемо його HtmlResult.

1. Навігаційна властивість.

Навігаційна властивість – властивість, при отримання даних з якої вона автоматично буде отримувати дані з БД.

1. Звичайна властивість.

Звичайна повинна приймати одне з наступних варіантів імені: 1. Ім’я\_навігаційної\_властивості + Ім'я ключа із зв'язаної таблиці - в нашому випадку ім'я навігаційного властивості Team, а ключа з моделі Team - Id, тому в нашому випадку нам треба назвати властивість TeamId, що власне і було зроблено у вищенаведеному коді. 2. Ім’я \_класу\_зв’язаної\_таблиці + Ім'я ключа із зв'язаної таблиці - в нашому випадку клас Team, а ключа з моделі Team - Id , тому знову ж в цьому випадку виходить TeamId.

1. Зовнішній ключ в моделі зі складною структурою.

Зовнішній ключ складається з двох властивостей: навігаційної і звичайної.

Зовнішній ключ - властивість TeamId тепер буде посилатися на властивість Id з таблиці Teams. Щоб задати зовнішній ключ, ми додаємо в панелі SQL внизу під дизайнером таблиці наступний рядок: CONSTRAINT [FK\_Players\_Teams] FOREIGN KEY ([TeamId]) REFERENCES [Teams] ([Id]) ON DELETE SET NULL

1. Метод Include.

За допомогою методу Include фреймворк підвантажує для кожного гравця також і команду, асоційовану з певним гравцем. А при виведенні моделі в представлення Index.cshtml фреймворк буде виводити для кожного гравця назву команди:

1. Рядок з’єднання connectionStrings.

<connectionStrings><addname="SoccerContext" connectionString="Data

Source=(LocalDB)\v11.0;AttachDbFilename='|DataDirectory|\Soccer\_Info.

mdf';IntegratedSecurity=True"

providerName="System.Data.SqlClient" />

</connectionStrings>

1. Призначення властивості Players в моделі Team.

В моделі Team властивість Players, призначення якої зберігати пов'язаних з командою гравців. Використаємо її. Наприклад, виведемо всі дані про команду, в тому числі про її гравцях.

1. Створення моделі.

Для створення моделі потрібно правою кнопкою миші натиснути по папціModels–Додати – Клас. І вписати ім’я моделі, після чого потрібно через консоль прописати Add-Migration “Назва міграції”. І запустити всі міграції прописавши Update-Database

1. Видалення моделі.

Тепер найважливіша частина - видалення даних. Навіть не в плані реалізації, скільки в плані безпеки. Додамо проста дія, яке видаляє об'єкт з бази даних:

public ActionResult Delete (int id)

{

Book b = db.Books.Find (id);

if (b! = null)

{

db.Books.Remove (b);

db.SaveChanges ();

}

return RedirectToAction ("Index");

}

Спочатку ми перевіряємо, а чи є такий об'єкт в бд, і якщо є, то викликаємо метод db.Books.Remove (b). Він встановить статус об'єкта моделі в Deleted, завдяки чому EntityFramework при виклику методу db.SaveChanges згенерує sql-вираз DELETE. Але ми можемо самі вказати статус явно:

1. Редагування моделі.

Для редагування моделі потрібно знайти потрібну модель в папці Models, відкрити її, відредагувати, після чого потрібно через консоль прописати Add-Migration “Назва міграції”. І запустити всі міграції прописавши Update-Database

1. Призначення класу SelectList.

Перший варіант дії Create обробляє Get-запит і видає представлення, передаючи в нього об'єкт SelectList - список всіх команд.

1. Віртуальні властивості Studentі Courses.

Моделі досить прості за винятком віртуальних властивостей - Students і Courses - завдяки цим властивостям і відбуватиметься зв'язок багатодо-багатьох.

1. Призначення public StudentsContext(): base(“DefaultConnection”){}

По-перше, щоб використовувати рядок підключення за замовчуванням, встановимо для нього контекст даних в конструкторі: public StudentsContext (): base ("DefaultConnection"). У створюваній 8

базі даних всі дані про студентів будуть зберігатися в таблиці Students, а дані про університетські курси - у таблиці Courses. Але ці таблиця повинні бути якось пов'язані зв'язком багато-до-багатьох. І цей зв'язок забезпечить ще одна таблиця, яка буде називатися CourseStudent. Для побудови цієї таблиці ми перевизначаються метод OnModelCreating, в якому за допомогою об'єкта modelBuilder створюємо нову таблицю і визначаємо її поля. Одне її поле - CourseId - буде посилатися на таблицю Courses і зберігати в собі id курсу. А друге поле - StudentId - буде посилатися на таблицю студентів і зберігати id студента. У результаті у нас вийде набір пар id курсу - id студента, завдяки цьому ми зможемо визначити зв'язок багато-до-багатьох. Проініціалізуємо базу даних початковими даними.

1. Призначення Database.SetInitializer(new CourseDbInitializer());

Зверніть увагу, як забезпечується зв'язок між курсами і студентами: ми просто додаємо набір створених студентів в колекцію Students для кожного курсу. І щоб все це запрацювало, додамо в файл Global.asax.cs в метод Application\_Start наступний рядок: Database.SetInitializer(newCourseDbInitializer());