ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет програмування та комп’ютерних

і телекомунікаційних систем

Кафедра інженерії програмного забезпечення

**Лабораторна робота №** 3

з дисципліни «Програмування в інтернет»

на тему:

«Моделі з простою структурою. Визначення таблиць. Створення логіки додатка. Виконання операцій з простими моделями - редагування, видалення, додавання.»

**Виконав:**

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

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

(підпис)

Хмельницький – 2020 р.

Мета. Створити примітивний додаток на MVC 4, який покликаний дати деяке початкове розуміння роботи MVC. Як шаблон проекту використовується Empty і проектується міні-додаток з нуля.

Завдання

Розробити web – додаток, що буде представляти примітивну емуляцію служби страхування автомобілів, в якому необхідно реалізувати операції - редагування, видалення, додавання.

Код Програми:

Моделі:

Type.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

namespace lab1.Models

{

public class Type

{

public int Id { get; set; }

public string Name { get; set; }

public int Price { get; set; }

}

}

Order.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

namespace lab1.Models

{

public class Order

{

public int OrderId { get; set; }

public string Person { get; set; }

public string Address { get; set; }

public int TypeId { get; set; }

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();

}

[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>

Вигляд програми:

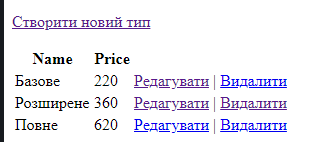


Рис 1.-Сторінка перегляду видів страхування

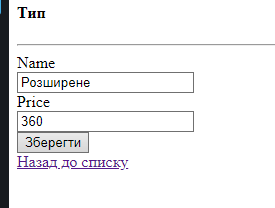


Рис 2.-Сторінка редагування

Контрольні питання:

1. Моделі в ASP.NET MVC 4. Моделі представляють собою прості класи і розташовуються в проекті в каталозі Models. Моделі описують логіку даних.
2. Взаємодія з базою даних

Ми створюємо звичайні класи, а Entity Framework вже сам визначає, як і де зберігати об'єкти цих класів. Випуск ASP.NET MVC 4 вже включає Entity Framework 5.0, однак у проектах по типу Empty вам доведеться підключати фреймворк через пакетний менеджер NuGet. Щоб підключитися до бази даних через Entity Framework, нам потрібен контекст даних.

1. Контекст даних в ASP.NET MVC 4. Щоб підключитися до бази даних через Entity Framework, нам потрібен контекст даних. Контекст даних являє собою клас, похідний від класу DbContext. Контекст даних містить одне або кілька властивостей типу DbSet, де Т представляє тип об'єкта, що зберігається в базі даних.
2. Шаблонні хелпери. Фреймворк MVC має шаблонні хелпери. Вони більш гнучкі в порівнянні з html-хелперами, так як в цьому випадку нам не треба думати, який нам треба створити елемент розмітки і який для цього вибрати хелпер. Ми просто повідомляємо шаблонному хелперу, яку властивість моделі ми хочемо використовувати, а фреймворк вже сам вибирає, який html-елемент згенерувати, виходячи з типу властивості і його метаданих.
3. Шаблонні хелпери для певної моделі.

Шаблонні хелпери:

- Display - створює елемент розмітки, який доступний тільки для читання, для зазначеного властивості моделі: Html.Display ("Name");

- DisplayFor - строго типізований аналог хелпера Display: Html.DisplayFor (e => e.Name);

- Editor - створює елемент розмітки, який доступний для редагування, для зазначеної властивості моделі: Html.Editor ("Name");

- EditorFor - строго типізований аналог хелпера Editor: Html.EditorFor (e => e.Name);

- DisplayText - створює вираз для зазначеного властивості моделі у вигляді простого рядка: Html.DisplayText ("Name");

- DisplayTextFor - строго типізований аналог хелпера DisplayText: Html.DisplayTextFor (e => e.Name). Крім даних шаблонів, які використовуються для окремої властивості моделі, є ще кілька шаблонів, які дозволяють згенерувати разом всі поля для певної моделі:

- DisplayForModel - створює поля для читання для всіх властивостей моделі: Html.DisplayForModel ();

- EditorForModel - створює поля для редагування для всіх властивостей моделі: Html.EditorForModel ()

1. Процес створення БД і підключення до неї

Ми можемо створити базу даних прямо в проекті, або ж створити її на сервері MS SQL. Створимо простий додаток на MVC 4 Bookstore2, як шаблон проекту ми виберемо шаблон Empty. Створимо модель. Після чого потрібно відкрити NuGet пакети, ввести там enable-migrate, потім додати Add-Migration “DataMigration” для додавання міграцій.

Після чого Update-Database для їх запуску.

1. Визначення рядка підключення

Визначення рядка підключення буде виглядати наступним чином:

<configuration>

<connectionStrings>

<add name="BookContext2" connectionString="Data Source=(LocalDB)\v11.0;AttachDbFilename='|DataDirectory|\Bookstore2.mdf';IntegratedSecurity=True" providerName="System.Data.SqlClient" />

</connectionStrings>

</configuration>

1. Файл Web.config. Файл конфігурації програми, який знаходиться в кореневій теці програми
2. Використання підстановки | DataDirectory | дозволяє опустити повний фізичний шлях до бази даних, яка зберігається в папці App\_Data.Підключення файла компонування
3. Редагування моделі

Нехай в деякій дії контролера ми отримуємо об'єкт моделі по Id і виводимо її поля для редагування в представленні: [HttpGet] public ActionResult EditBook(int? id) { if (id == null) { return HttpNotFound(); } Book book = db.Books.Find(id); if (book == null) { return HttpNotFound(); } return View(book); } На випадок, якщо користувач не вкаже Id, ми встановлюємо в якості параметра не int, а int ?. І якщо такий параметр не переданий, то повертаємо результат методу HttpNotFound. А представлення у нас буде містити набір хелперів EditorFor для деяких полів моделі: @{ ViewBag.Title = "Редагувати книгу"; Layout = "~/Views/Shared/\_Layout.cshtml"; } @model Bookstore2.Models.Book

Книга № @Model.Id

@using (Html.BeginForm("EditBook", "Home", FormMethod.Post)) { @Html.HiddenFor(m => m.Id)

@Html.LabelFor(m => m.Name, "Назва книги")   
@Html.EditorFor(m => m.Name)

@Html.LabelFor(m => m.Author, "Автор")   
@Html.EditorFor(m => m.Author)

@Html.LabelFor(m => m.Price, "Ціна")   
@Html.EditorFor(m => m.Price) }

Так як унікальний ідентифікатор Id книги нам не треба редагувати, то поле для його відображення зробимо прихованим, тобто скористаємося хелперів Html.HiddenFor. Тепер нам потрібен сам код збереження. Визначимо в контролері дію EditBook, яка буде обробляти POST-запити: 60 [HttpPost] public ActionResult EditBook(Book book) { db.Entry(book).State = EntityState.Modified; db.SaveChanges(); return RedirectToAction("Index"); }

1. Видалення моделі

Додамо просту дію, яке видаляє модель з бази даних:

public ActionResult Delete(int id)

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

63

if (b != null)

{ db.Books.Remove(b);

db.SaveChanges();

}

return RedirectToAction("Index");

}

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

public ActionResult Delete (int id)

{ Book b = new Book {Id = id};

db.Entry(b).State = EntityState.Deleted;

db.SaveChanges ();

return RedirectToAction ("Index");

}

Подібний підхід має один плюс - ми уникаємо першого запиту до БД, який у нас був у вираженні Book b = db.Books.Find(id);. Тобто замість двох запитів до БД тепер у нас тільки один. Але в цілому подібний метод на видалення має один мінус в плані безпеки. Один - GET-запит до методу Delete несе потенційну вразливість. Тому переробимо метод таким чином:

[HttpGet]

public ActionResult Delete(int? id) {

if (id == null)

{ return HttpNotFound(); }

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

if (b == null)

{ return HttpNotFound(); }

return View(b);

}

[HttpPost, ActionName("Delete")]

public ActionResult DeleteConfirmed(int? id)

{ if (id == null)

{ return HttpNotFound(); }

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

if (b == null) { return HttpNotFound(); }

db.Books.Remove(b);

db.SaveChanges();

return RedirectToAction("Index"); }

Тепер замість одного методу Delete цілих два. Атрибут ActionName ("Delete") вказує, що метод DeleteConfirmed буде сприйматися як дія Delete. Перший метод передає модель, що видаляється в представлення. На представленні за допомогою натискання кнопки ми зможемо підтвердити видалення. І Id піде другим методом за запитом POST. Таким чином, ми відійдемо від вразливості GETзапиту. Ну і саме представлення: @{ Layout = "~/Views/Shared/\_Layout.cshtml"; } @model Bookstore2.Models.Book

Видалення книги

Назва

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

Автор

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

Ціна

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

@using (Html.BeginForm())

1. a

Для додавання моделі спочатку визначимо пару дій: [HttpGet] public ActionResult Create() { return View(); } [HttpPost] public ActionResult Create(Book book) { db.Books.Add(book); db.SaveChanges(); return RedirectToAction("Index"); } Перший метод повертає користувачеві представлення з формою для додавання, а другий - приймає дані цієї форми. Тепер створимо представлення. Представлення буде виглядати наступним чином: @model Bookstore2.Models.Book @{ Layout = "~/Views/Shared/\_Layout.cshtml"; }

Нова книга

@using (Html.BeginForm()) { @Html.LabelFor(model => model.Name, "Назва книги")   
@Html.EditorFor(model => model.Name)   
  
62 @Html.LabelFor(model => model.Author, "Автор")   
@Html.EditorFor(model => model.Author)   
  
@Html.LabelFor(model => model.Price, "Ціна")   
@Html.EditorFor(model => model.Price)   
}При отриманні моделі book в дії Create метод db.Books.Add(book); буде встановлювати значення Added в якості стану моделі. Тому метод db.SaveChanges () згенерує вираз INSERT для вставки моделі в таблицю. Тобто метод Create ми могли б переписати таким чином: [HttpPost] public ActionResult Create (Book book) { db.Entry (book) .State = EntityState.Added; db.SaveChanges (); return RedirectToAction ("Index"); }

1. Хелпер Html.HiddenFor

Так як унікальний ідентифікатор Id нам не треба редагувати, то поле для його відображення зробимо прихованим, тобто скористаємося хелперів Html.HiddenFor.

1. По суті тут ми визначили одну дію Buy, проте в одному випадку вона виконується при отриманні запиту GET, а в другому випадку - при отриманні запиту POST, що ми і визначили за допомогою атрибутів [HttpGet] і [HttpPost]. Метод public ActionResult Buy (int id) бере id обраної книги і повертає відповідне представлення.
2. За допомогою рядка db.Entry(book).State = EntityState.Modified; ми вказуємо, що об'єкт book існує вже в базі даних, і для нього треба внести в базу змінене значення, а не створювати новий запис.
3. За допомогою рядка db.Entry(book).State = EntityState.Added; ми вказуємо, що об'єкт book треба створити.
4. За допомогою рядка db.Entry(b).State = EntityState.Deleted; ми вказуємо, що об'єкт book існує вже в базі даних, і для нього треба видалити запис з бази.
5. Get і POST-запити при видалені моделі

[HttpGet]

public ActionResult Delete(int? id)

{ if (id == null)

{ return HttpNotFound(); }

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

if (b == null)

{ return HttpNotFound(); }

return View(b); }

[HttpPost, ActionName("Delete")]

public ActionResult DeleteConfirmed(int? id)

{ if (id == null)

{ return HttpNotFound(); }

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

if (b == null)

{ return HttpNotFound(); }

db.Books.Remove(b);

db.SaveChanges();

return RedirectToAction("Index"); }

1. Шаблони формування

Оскільки часто розробники змушені створювати представлення для одних і тих же дій: додавання, зміни, видалення і перегляду записів з БД, то команда розробників MVC впровадила таку корисну функцію, як шаблони формування (scaffolding templates). Ці шаблони дозволяють для заданої моделі і контексту даних сформувати всю необхідну розмітку для представлень і контролера, за допомогою яких можна управляти записами у БД.