**ЛАБОРАТОРНА РОБОТА №4**

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

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

**4.1 Теоретичні відомості**

Контролер є центральним компонентом в архітектурі MVC. Контролер отримує введення користувача, обробляє його і посилає назад результат обробки, наприклад, у вигляді представлення. При використанні контролерів існують деякі умовності. Так, за угодами про іменування назви контролерів повинні закінчуватися на суфікс "Controller", інша ж частина до цього префікса вважається ім'ям контролера. 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.* Контролер це звичайний клас, який наслідується від базового класу System.Web.Mvc.Controller. У свою чергу клас Controller реалізує абстрактний базовий клас ControllerBase, а через нього і інтерфейс IController. Таким чином, формально, щоб створити свій клас контролера, достатньо створити клас, який реалізує інтерфейс IController і має в імені суфікс Controller. Інтерфейс IController визначає один єдиний метод Execute, який відповідає за обробку контексту запиту:

public interface IController

{

void Execute(RequestContext requestContext);

}

При зверненні до будь-якого контролеру система передає в нього контекст запиту. У цей контекст запиту включається все: куки, відправлені дані форм, рядка запиту, ідентифікаційні дані користувача і т.д. Реалізація інтерфейсу IController дозволяє отримати цей контекст запиту в методі Execute через параметр RequestContext. Хоча за допомогою реалізації інтерфейсу IController дуже просто створювати контролери, але в реальності частіше оперують більш високорівневими класами, як наприклад клас Controller, оскільки він надає більш потужні засоби для обробки запитів. І якщо при реалізації інтерфейсу IController ми маємо справу з одним методом Execute, і всі запити до цього контролеру, будуть оброблятися тільки одним методом, то при наслідуванні класу Controller ми можемо створювати безліч методів дій, які відповідатимуть за обробку вхідних запитів, і повертати різні результати дій.

Щоб створити стандартний контролер, ми можемо також додати в папку Controllers простий клас і наслідувати від класу Controller.

Методи дій (action methods) представляють такі методи контролера, які обробляють запити за певним URL. Наприклад, візьмемо проект з першої лабораторної роботи. У ньому був визначений наступний контролер:

    public class HomeController : Controller

{

//

// GET: /Home/

BookContext db = new BookContext();

public ActionResult Index()

{

// Отримуємо з БД всі об'єкти Book

IEnumerable<Book> books = db.Books;

// Передаємо всі отримані об'єкти в динамічну властивість Books в ViewBag

ViewBag.Books = books;

// Повертаємо представлення

return View();

}

[HttpGet]

public ActionResult Buy(int id)

{

ViewBag.BookId = id;

return View();

}

[HttpPost]

public string Buy(Purchase purchase)

{

purchase.Date = DateTime.Now;

// Додаємо інформацію про покупку в базу даних

db.Purchases.Add(purchase);

// Зберігаємо в БД всі зміни

db.SaveChanges();

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

}

}

Тут методи Index і Buy є методами дій чи просто діями контролера. При отриманні запиту типу */Home/Index* контролер передає обробку запиту дії Index.

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 розбито на два методи, 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.

Отримання вхідних даних. У додатку з першої лабораторної метод Buy використовував параметр purchase. Так як даний метод обробляє POST-запити, то ми можемо відправити йому таку форму:

<form method="post" action="">

<input type="hidden" value="@ViewBag.BookId" name="BookId" />

<table>

<tr>

<td><p>Введіть своє ім'я </p></td>

<td><input type="text" name="Person" /> </td>

</tr>

<tr>

<td><p>Введіть адресу :</p></td>

<td>

<input type="text" name="Address" />

</td>

</tr>

<tr><td><input type="submit" value="Відправити" /> </td><td></td></tr>

</table>

</form>

Значення атрибута name у всіх полів на цій формі відповідає назві властивості моделі, тому система автоматично зв'яже значення полів з відповідними властивостями. А в методі Buy весь цей набір властивостей перетвориться на модель Purchase. Крім POST-запитів у нас є також GET-запити, при яких всі параметри передаються в рядку запиту. Наприклад, друга версія методу Buy як параметр приймає значення типу int: public ActionResult Buy(int id). Стандартний get-запит приймає приблизно таку форму: *назва\_ресурса?параметр1=значення1&параметр2=значення2*. Тобто запит до даного методу міг би виглядати так: *Home/Buy?Id = 2*. Назва параметрів методу має співпадати з назвою параметрів в рядку запиту. Завдяки цьому система зможе їх автоматично зв'язати. Ну а в самому методі ми зможемо отримати цей параметр і використовувати його на свій розсуд. Крім того, система маршрутизації дозволяє створювати маршрути. Наприклад, за замовчуванням в проекті MVC визначається наступний маршрут: *Контролер/Метод/id*. Останній параметр є опціональним. І завдяки цьому ми можемо передати параметр id і так: *Home/Buy/2*

Для прикладу визначимо дію, яка буде підраховувати площу трикутника:

public string Square (int a, int h)

{

double s = a \* h / 2;

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

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

}

В цьому випадку ми можемо звернутися до дії, набравши в адресному рядку *Home/Square? a = 10 & h = 3*, і додаток видав би нам потрібний результат.

Ми також можемо поставити для параметрів значення за замовчуванням:

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

{

double s = a \* h / 2;

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

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

}

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

Отримання даних з контексту запиту. Крім того, ми можемо отримати параметри, та й не тільки параметри, але й інші дані, пов'язані із запитом, з об'єктів контексту запиту. Нам доступні наступні об'єкти контексту Request.

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

public string Square ()

{

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

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

double s = a \* h / 2;

return "<h2> Площа трикутника з основою" + a + "і висотою" + h + "дорівнює" + s + "</ h2>";

}

Результати дій. Коли користувач звертається до ресурсу, як правило, він очікує отримати певну відповідь, наприклад, у вигляді веб-сторінки з деякими даними. На стороні сервера метод контролера, отримуючи параметри, обробляє їх і формує певну відповідь у вигляді результату дії. Як правило, повертаним результатом є об'єкт класу, похідного від ActionResult. ActionResult представляє собою абстрактний клас, в якому визначено один метод ExecuteResult, що перевизначається в наслідуваних класах:

public abstract class ActionResult

{

public abstract void ExecuteResult(ControllerContext context);

}

Створимо свої результати дій. Вони будуть дуже простими. Візьмемо який-небудь проект, наприклад, з першої лабораторної, і додамо в нього нову папку Util, яка міститиме нові класи. Після додавання папки додамо в неї перший клас. Назвемо його HtmlResult. Він у нас буде містити наступний код:

public class HtmlResult : ActionResult

{

private string htmlCode;

public HtmlResult(string html)

{

htmlCode = html;

}

public override void ExecuteResult(ControllerContext context)

{

string fullHtmlCode = "<!DOCTYPE html><html><head>";

fullHtmlCode += "<title>Головна сторінка</title>";

fullHtmlCode += "<meta charset=utf-8 />";

fullHtmlCode += "</head> <body>";

fullHtmlCode += htmlCode;

fullHtmlCode += "</body></html>";

context.HttpContext.Response.Write(fullHtmlCode);

}

}

У конструкторі класу HtmlResult отримуємо переданий html-код, а в методі Execute вставляємо його в загальне оточення, щоб вийшла повноцінна html-сторінка, і пишемо її у вихідний потік: context.HttpContext.Response.Write (fullHtmlCode); Щоб використати цей клас підключимо в контролер простір імен нового класу: using BookStore.Util; і додамо новий метод:

public ActionResult GetHtml()

{

return new HtmlResult("<h2> Привіт світ! </ h2>");

}

І звернувшись до цього методу з браузера, наприклад, *Home/GetHtml*, ми отримаємо html-сторінку. Хоча даний приклад досить примітивний, але в цілому він демонструє, як працюють класи результатів дій. Створимо ще один клас результатів. Додамо в папку Util новий клас ImageResult:

public class ImageResult : ActionResult

{

private string path;

public ImageResult(string path)

{

this.path = path;

}

public override void ExecuteResult(ControllerContext context)

{

context.HttpContext.Response.Write("<div style = 'width: 100%; text-align: center;'>"+"<img style='max-width: 600px;' src = '"+path + "' /> </ div> ");

}

}

Даний клас не складніший попереднього і просто передає зображення в html-коді. Тоді метод, що використовує даний результат дій, міг би виглядати так:

public ActionResult GetImage()

{

string path = "../Images/4.jpg";

return new ImageResult(path);

}

Тут передбачається, що в проекті є папка Images, в якій є зображення 4.jpg. І тоді, якщо ми в браузері звернемося до цієї дії, наприклад, Home / GetImage, то зможемо побачити зображення.

**4.2 Хід роботи**

**4.2.1 Моделі зі складною структурою. Відношення один до багатьох**

Раніше використані нами моделі Book і Purchase мали досить просту структуру - одні прості властивості типу int або string. Розглянемо, як приклад, наступні дві моделі:

public class Player

{

public int Id { get; set; }

public string Name { get; set; }

public int Age { get; set; }

public string Position { get; set; }

public int? TeamId { get; set; }

public Team Team { get; set; }

}

public class Team

{

public int Id { get; set; }

public string Name { get; set; }

public string Coach { get; set; }

public IEnumerable<Player> Players { get; set; }

}

Тут клас Player має властивість Team, яка показує приналежність гравця до певної команди. Властивість Team називається навігаційним властивістю - при отриманні даних про гравця вона автоматично буде отримувати дані з БД. Однак для цього нам треба також встановити зовнішній ключ.

Зовнішній ключ складається з двох властивостей: навігаційної і звичайної. Навігаційну ми розглянули вище. А 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.

Тепер створимо контекст даних, що використовує моделі:

public class SoccerContext: DbContext

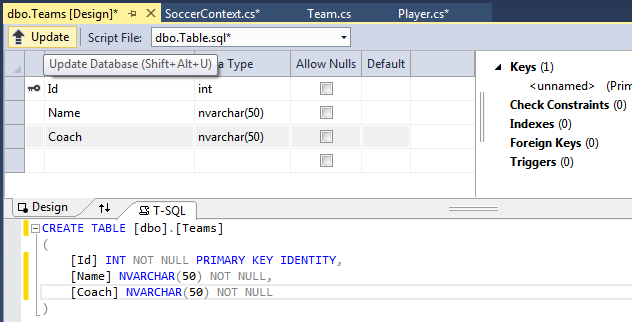
{

public DbSet<Player> Players { get; set; }

public DbSet<Team> Teams { get; set; }

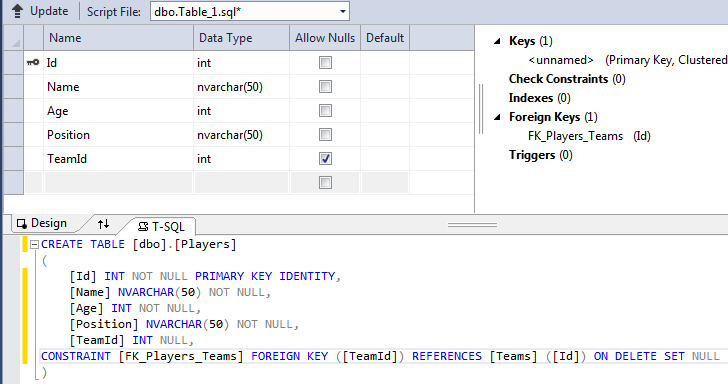
}

Тепер подивимося, як би це все розташовувалося в БД. Припустимо, у нас є деяка база даних Soccer\_Info.mdf. Нехай наші моделі Player і Team розташовуються відповідно в таблицях Players і Teams. Визначення таблиці Teams, яка буде зберігати об'єкти моделі Team, виглядає наступним чином (рис.4.1).



**Рис. 4.1 - Визначення таблиці Teams**

Тоді щоб проектувати модель Player на таблицю Players в базі даних, нам треба задати наступне визначення стовпців таблиці (рис.4.2).

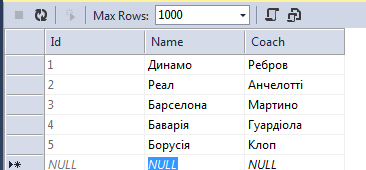


**Рис. 4.2 - Визначення таблиці Players**

Крім того, тут ми задаємо 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:

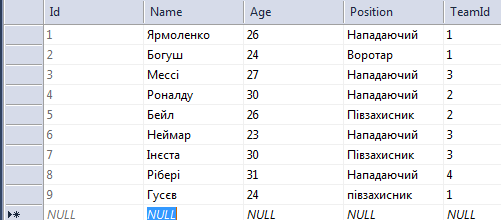
CONSTRAINT [FK\_Players\_Teams] FOREIGN KEY ([TeamId]) REFERENCES [Teams] ([Id]) ON DELETE SET NULL

Це звичайний вираз мови SQL, який пов'язує стовпці двох таблиць. Остання частина цього виразу (ON DELETE SET NULL) вказує, що при видаленні об'єкту з таблиці Teams, властивості TeamId, яка посилалася на віддалений об'єкт, буде присвоєно значення null. Це треба, щоб у нас гравці при видаленні команд не відносилися більше до видалених команд. Однак ми можемо задати й іншу дію, наприклад, при видаленні команди видалити всіх її гравців. Для цього нам треба написати ON DELETE CASCADE. Тепер після визначення таблиць наповнимо їх деякими початковими даними. Наприклад додамо деякі дані в таблицю (рис.4.3).



**Рис. 4.3 - Заповнення таблиці Teams**

І в таблицю Players (де стовпець TeamId містить деяке існуюче значення з рядка Id таблиці Teams ) (рис.4.4).



**Рис. 4.4 - Заповнення таблиці Players**

Тепер перейдемо до створення логіки додатка. Додамо в додаток контролер і визначимо в ньому вивід всіх гравців на сторінку:

public class HomeController : Controller

{

//

// GET: /Home/

SoccerContext db = new SoccerContext();

// Виводимо всіх футболістів

public ActionResult Index()

{

var players = db.Players.Include(p => p.Team);

return View(players.ToList()); }

}

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:

@model IEnumerable<NavigationProperty.Models.Player>

@{

ViewBag.Title = "Каталог гравців";

}

<h2>Каталог гравців</h2>

<p>

@Html.ActionLink("Добавити гравця", "Create")

</p>

<table>

<tr>

<th>Ім'я гравця</th>

<th>Вік</th>

<th>Позиція на полі</th>

<th>Команда</th>

<th></th>

</tr>

@foreach (var item in Model)

{

<tr>

<td>

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

</td>

<td>

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

</td>

<td>

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

</td>

<td>

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

</td>

<td>

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

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

</td>

</tr>

}

</table>

<p>

@Html.ActionLink("Каталог команд", "ListTeams")

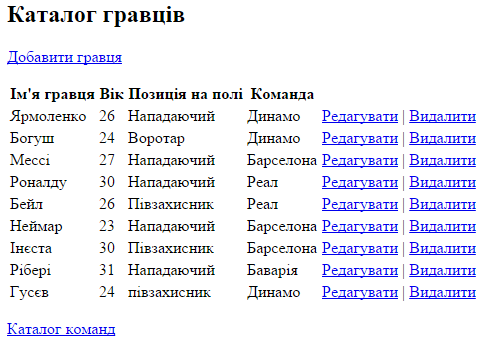
</p>

Оскільки ми в контролері за допомогою методу Include отримали пов'язаний з гравцем об'єкт Team з навігаційного властивості TeamId, то при передачі в контролер всіх об'єктів Player до них також чіпляються пов'язані з ними об'єкти Team, тому ми можемо цілком використовувати в представленні властивість item.Team.Name для отримання імені команди (рис.4.5). Рядок зєднання:

<connectionStrings><add name="SoccerContext" connectionString="Data Source=(LocalDB)\v11.0;AttachDbFilename='|DataDirectory|\Soccer\_Info.mdf';Integrated Security=True"

providerName="System.Data.SqlClient" />

</connectionStrings>



**Рис. 4.5 - Вивід для кожного гравця назву команди**

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. Спочатку додамо в контролер наступний метод:

public ActionResult TeamDetails(int? id)

{

if (id == null)

{

return HttpNotFound();

}

Team team = db.Teams.Find(id);

if (team == null)

{

return HttpNotFound();

}

team.Players = db.Players.Where(m => m.TeamId == team.Id);

return View(team);

}

По-перше, щоб обробити введення при відсутності переданого значення, як параметр використовуємо int? id. По-друге, ми довантажили всіх гравців, пов'язаних з командою, у виразі

team.Players = db.Players.Where(m => m.TeamId == team.Id);

Представлення TeamDetails.cshtml для відображення даних про команду (рис.4.6) може виглядати наступним чином:

@using NavigationProperty.Models

@model Team

@{

ViewBag.Title = "Команда " + @Model.Name;

Layout = "~/Views/Shared/\_Layout.cshtml";

}

<div>

<h4>Команда @Model.Name</h4>

<hr />

<dl>

<dt>Назва</dt>

<dd>

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

</dd>

<dt>Тренер</dt>

<dd>

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

</dd>

<dt>Гравці</dt>

<dd>

<ul>

@foreach (Player player in Model.Players)

{

<li>@player.Name (@player.Position)</li>

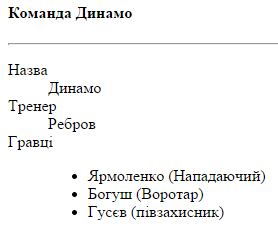
}

</ul>

</dd>

</dl>

</div>



**Рис. 4.6- Гравці пов'язані з командою**

Розглянемо, як виконати операції з моделями - редагування, видалення, додавання. Ці дії не сильно відрізняються від логіки роботи з простими моделями. Єдина відмінність - ми повинні враховувати значення навігаційне властивість, наявну у складній моделі.

Додавання моделі. Розглянемо логіку додавання моделі Player в таблицю Players. Спочатку пропишемо в контролері наступну дію Create:

[HttpGet]

public ActionResult Create()

{

//Формуємо список команд для передачі в представлення

SelectList teams = new SelectList(db.Teams, "Id", "Name");

ViewBag.Teams = teams;

return View();

}

[HttpPost]

public ActionResult Create(Player player)

{

// Додаємо гравця в таблицю

db.Players.Add(player);

db.SaveChanges();

// перенаправляємо на головну сторінку

return RedirectToAction("Index");

}

Перший варіант дії Create обробляє Get-запит і видає представлення, передаючи в нього об'єкт SelectList - список всіх команд. Другий варіант дії Create обробляє Post-запит отримує введену користувачем у представленні модель і додає її в БД (рис.4.7). Розглянемо саме представлення Create.cshtml:

@model NavigationProperty.Models.Player

@{

ViewBag.Title = " Додавання гравця ";

Layout = "~/Views/Shared/\_Layout.cshtml";

}

<h2> Додавання нового гравця </h2>

@using (Html.BeginForm())

{

<fieldset>

<legend>Футболіст</legend>

<p>

Ім’я гравця <br />

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

</p>

<p>

Вік <br />

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

</p>

<p>

Позиція на полі <br />

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

</p>

<p>

Команда <br />

@Html.DropDownListFor(model => model.TeamId, ViewBag.Teams as SelectList)

</p>

<p>

<input type="submit" value="Добавити гравця " />

</p>

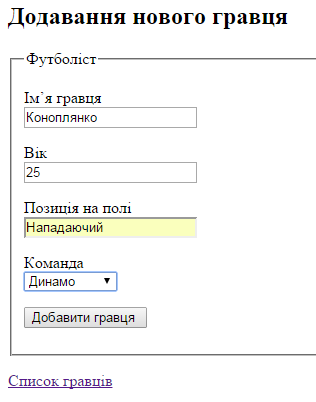
</fieldset>

}

<div>

@Html.ActionLink("Список гравців", "Index")

</div>



**Рис. 4.7- Додавання нового гравця**

Редагування моделі. Редагування працює подібним чином. Визначимо в контролері дії, які відповідають за редагування:

[HttpGet]

public ActionResult Edit(int? id)

{

if (id == null)

{

return HttpNotFound();

}

// Знаходимо в БД футболіста

Player player = db.Players.Find(id);

if (player != null)

{

// Створюємо список команд для передачі в представлення

SelectList teams = new SelectList(db.Teams, "Id", "Name", player.TeamId);

ViewBag.Teams = teams;

return View(player);

}

return RedirectToAction("Index");

}

[HttpPost]

public ActionResult Edit(Player player)

{

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

db.SaveChanges();

return RedirectToAction("Index");

}

Тут ми також формуємо список команд, які ми отримуємо з БД, в об'єкті SelectList. І після отримання запиту на редагування певної моделі Player контролер передає цю модель і список команд в представлення Edit.cshtml:

@model NavigationProperty.Models.Player

@{

ViewBag.Title = "Edit";

}

<h2>Редагування гравця </h2>

@using (Html.BeginForm())

{

<fieldset>

<legend>Футболіст</legend>

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

<p>

Ім’я гравця <br />

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

</p>

<p>

Вік <br />

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

</p>

<p>

Позиція на полі <br />

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

</p>

<p>

Команда <br />

@Html.DropDownListFor(model => model.TeamId, ViewBag.Teams as SelectList)

</p>

<p>

<input type="submit" value="Зберегти" />

</p>

</fieldset>

}

<div>

@Html.ActionLink("Повернутися до списку футболістів", "Index")

</div>

Видалення проводиться так, як і у випадку із звичайною моделлю.

**4.2.2 Моделі зі складною структурою. Відношення багато до багатьох**

Крім моделей зі зв'язком один–до–одного і один–до–багатьох є моделі по типу "багато–до–багатьох". Найбільш поширений приклад - навчання в університеті, де різна кількість студентів може відвідувати різну кількість дисциплін. І при цьому у нас може виникнути необхідність, як вести облік студентів з конкретної дисципліни, так і вести облік різних дисциплін для конкретного студента. Спробуємо змоделювати цю ситуацію в додатку ASP.NET MVC. По-перше, створимо новий додаток за шаблоном Basic. Насамперед нам треба створити моделі. Для простоти будемо використовувати підхід Code First. Отже, у нас є дві моделі - студент і курс університетської дисципліни. Спочатку додамо модель Student з наступним змістом:

public class Student

{

public int Id { get; set; }

public string Name { get; set; }

public string Surname { get; set; }

public virtual ICollection<Course> Courses { get; set; }

}

public class Course

{

public int Id { get; set; }

public string Name { get; set; }

public virtual ICollection<Student> Students { get; set; }

}

Моделі досить прості за винятком віртуальних властивостей - Students і Courses - завдяки цим властивостям і відбуватиметься зв'язок багато-до-багатьох. Наступний етап - створення контексту даних. Додамо в проект наступний клас StudentsContext:

public class StudentsContext : DbContext

{

public DbSet<Student> Students { get; set; }

public DbSet<Course> Courses { get; set; }

public StudentsContext()

: base("DefaultConnection")

{ }

protected override void OnModelCreating(DbModelBuilder modelBuilder)

{

modelBuilder.Entity<Course>().HasMany(c => c.Students)

.WithMany(s => s.Courses)

.Map(t => t.MapLeftKey("CourseId")

.MapRightKey("StudentId")

.ToTable("CourseStudent"));

}

}

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

public class CourseDbInitializer : DropCreateDatabaseAlways<StudentsContext>

{

protected override void Seed(StudentsContext context)

{

Student s1=new Student { Id = 1, Name = "Єгор", Surname = "Іванов" };

Student s2=new Student { Id=2, Name="Марія", Surname="Васильєва" };

Student s3=new Student { Id=3, Name="Олег", Surname = "Кузнєцов" };

Student s4=new Student { Id=4, Name="Ольга", Surname = "Петрова" };

context.Students.Add(s1);

context.Students.Add(s2);

context.Students.Add(s3);

context.Students.Add(s4);

Course c1 = new Course

{

Id = 1,

Name = "Операційні системи",

Students = new List<Student>() { s1, s2, s3 }

};

Course c2 = new Course

{

Id = 2,

Name = "Алгоритми і структури даних",

Students = new List<Student>() { s2, s4 }

};

Course c3 = new Course

{

Id = 3,

Name = "Основи HTML і CSS",

Students = new List<Student>() { s3, s4, s1 }

};

context.Courses.Add(c1);

context.Courses.Add(c2);

context.Courses.Add(c3);

base.Seed(context);

}

}

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(new CourseDbInitializer());

Створимо контролер. Так як виведення таблиці як курсів, так студентів не представляє складності, то ми його разбирати не будемо. Нас цікавить отримання пов'язаних даних. Тому додамо контролер HomeController і визначимо в ньому наступний метод Details, який буде виводити інформацію по конкретному студенту:

public class HomeController : Controller

{

//

// GET: /Home/

private StudentsContext db = new StudentsContext();

public ActionResult Index()

{

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

}

public ActionResult Details(int id = 0)

{

Student student = db.Students.Find(id);

if (student == null)

{

return HttpNotFound();

}

return View(student);

}

protected override void Dispose(bool disposing)

{

db.Dispose();

base.Dispose(disposing);

}

}

Метод Details це звичайний метод отримання інформації по об'єкту Student. І завдяки визначенню в моделі Student віртуальної властивості Courses, всі пов'язані з даними студентом університетські курси будуть автоматично підключатися (рис.4.8). Тепер додамо представлення Details.cshtml:

@using Student\_Course.Models

@model Student

@{

ViewBag.Title = "Details";

}

<fieldset>

<legend>Інформація про студента</legend>

<div class="display-label"><b>Ім’я</b></div>

<div class="display-field">

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

</div>

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

<div class="display-field">

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

</div>

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

<ul>

@foreach (Course c in Model.Courses)

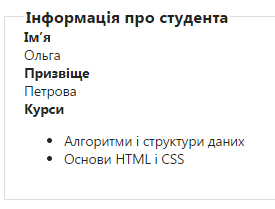
{

<li>@c.Name</li>

}

</ul>

</fieldset>



**Рис. 4.7 - Отримання інформації по об'єкту Student**

Простий вивід пов'язаних даних зі зв'язком "багато-до-багатьох" не представляє особливих проблем. Розглянемо, як робити створення і редагування подібних моделей. Продовжимо приклад зі студентами та курсами і внесемо в нього функціональність редагування студентів. Додамо в контролер HomeController наступну дію Edit:

[HttpGet]

public ActionResult Edit(int id = 0)

{

Student student = db.Students.Find(id);

if (student == null)

{

return HttpNotFound();

}

ViewBag.Courses = db.Courses.ToList();

return View(student);

}

[HttpPost]

public ActionResult Edit(Student student, int[] selectedCourses)

{

Student newStudent = db.Students.Find(student.Id);

newStudent.Name = student.Name;

newStudent.Surname = student.Surname;

newStudent.Courses.Clear();

if (selectedCourses != null)

{

//отримуємо вибрані курси

foreach (var c in db.Courses.Where(co => selectedCourses.Contains(co.Id)))

{

newStudent.Courses.Add(c);

}

}

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

db.SaveChanges();

return RedirectToAction("Index");

}

Дія Edit представляє два методи - для запиту get і для запиту post. Метод, що обробляє запит get, стандартний - він передає в представлення модель, що редагується, а також список усіх курсів через ViewBag.Courses = db.Courses.ToList ();, щоб ми могли потім ці курси вивести в представленні. Метод post приймає отримані дані, тільки крім моделі Student сюди також передаються всі вибрані курси у вигляді масиву id курсів. В цей масив і будуть міститися всі значення всіх зазначених на формі прапорців. У самому методі після проходження валідації ми встановлюємо нові значення властивостей моделі. Потім нам треба встановити в колекції Courses у студента всі зазначені курси. Для цього проходимо по всіх курсах з бази даних, і якщо вони були відзначені на формі і відсутні у списку, то додаємо їх. Невідмічені видаляємо (якщо вони є в списку). У підсумку інформація в базі даних буде відповідним чином оновлена. Додамо представлення Edit.cshtml:

@using Student\_Course.Models

@model Student

@{

ViewBag.Title = "Edit";

}

@using (Html.BeginForm())

{

<fieldset>

<legend>Студент</legend>

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

<div class="editor-label"><b>Ім’я</b></div>

<div class="editor-field">

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

</div>

<div class="editor-label"><b>Призвіще</b></div>

<div class="editor-field">

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

</div>

<div class="editor-label"><b>Курси</b></div>

@{

List<Course> courses = ViewBag.Courses;

foreach (Course c in courses)

{

<input type="checkbox" name="selectedCourses" value="@c.Id"

@(Model.Courses.Contains(c) ? "checked=\"checked\"" : "") />@c.Name <br />

}

}

<p>

<input type="submit" value="Зберегти" />

</p>

</fieldset>

}

Збереження моделі буде багато в чому ідентично редагуванню.

**4.3 Завдання до лабораторної роботи**

Завдання 1. Визначити моделі зі складною структурою. Відношення між моделями один до багатьох. Реалізувати виконання операцій з моделями - редагування, видалення, додавання.

1.Розробити web – додаток «Магазин по продажу комп’ютерної техніки».

2. Розробити web – додаток «Бібліотека».

3.Розробити web – додаток «Довідкове бюро залізничного вокзалу по продажу білетів».

4.Розробити web – додаток «Склад по продажу товару сільськогосподарської продукції».

5.Розробити web – додаток «Довідкового бюро автовокзалу по продажу білетів».

6.Розробити web – додаток «Салон по продажу мобільної техніки».

7.Розробити web – додаток, «Служба таксі».

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

9. Розробити web – додаток «Надання кредитів банком».

10.Розробити web – додаток «Салон по продажу автомобілів».

11.Розробити web – додаток «Кінотеатри міста».

12.Розробити web – додаток «Регістрація ДТП».

13.Розробити web – додаток «Регістрація порушення правил дорожнього руху робітниками ДАІ».

14.Розробити web – додаток «Виставлення оцінок за результатами екзаменаційної сесії».

15.Розробити web – додаток, «Виконання дипломних робіт студентами».

Завдання 2. Завершити проект розглянутий в п. 4.2.2 лабораторної роботи.

**4.4. Контрольні питання**

1. Призначення контролера
2. Звернення до контролера з веб-браузера
3. Методи дій
4. Звернення до методу дій з веб-браузера
5. Опціональний параметр
6. Значення параметрів за замовчуванням
7. Отримання даних з контексту запиту
8. Об'єкт Request
9. Клас ActionResult
10. Створення власних результатів дій
11. Навігаційна властивість
12. Звичайна властивість
13. Зовнішній ключ в моделі зі складною структурою
14. Метод Include
15. Рядок зєднання connectionStrings
16. Призначення властивості Players в моделі Team
17. Створення моделі
18. Видалення моделі
19. Редагування моделі
20. Призначення класу SelectList
21. Віртуальні властивості - Students і Courses
22. Призначення public StudentsContext() : base("DefaultConnection") { }
23. Призначення Database.SetInitializer(new CourseDbInitializer());

**Література:** [1,3,5,7,8,9].