Форма № Н-9.02

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет програмування та комп’ютерних

і телекомунікаційних систем

Кафедра інженерії програмного забезпечення

Лабораторна робота №3

з дисципліни «Програмування 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-17-1 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Хомяк Б. В.

(підпис)

**Перевірив:**

викладач \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Омельчук С. С.

(підпис)

Хмельницький – 2017 р.

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

**Мета.** Отримати навикирозробки Web - додатків з простими моделями.

**Завдання.** 2. Розробити web – додаток, що буде представляти примітивну емуляцію роботи бібліотеки, в якому необхідно реалізувати операції - редагування, видалення, додавання.

**Хід роботи.**

Створимо модель, що представляє книгу і її покупку:

public class Book

{

public int Id { get; set; }

public string Name { get; set; }

public string Author { get; set; }

public int Price { get; set; }

}

public class Purchase

{

public int PurchaseId { get; set; }

public string Person { get; set; }

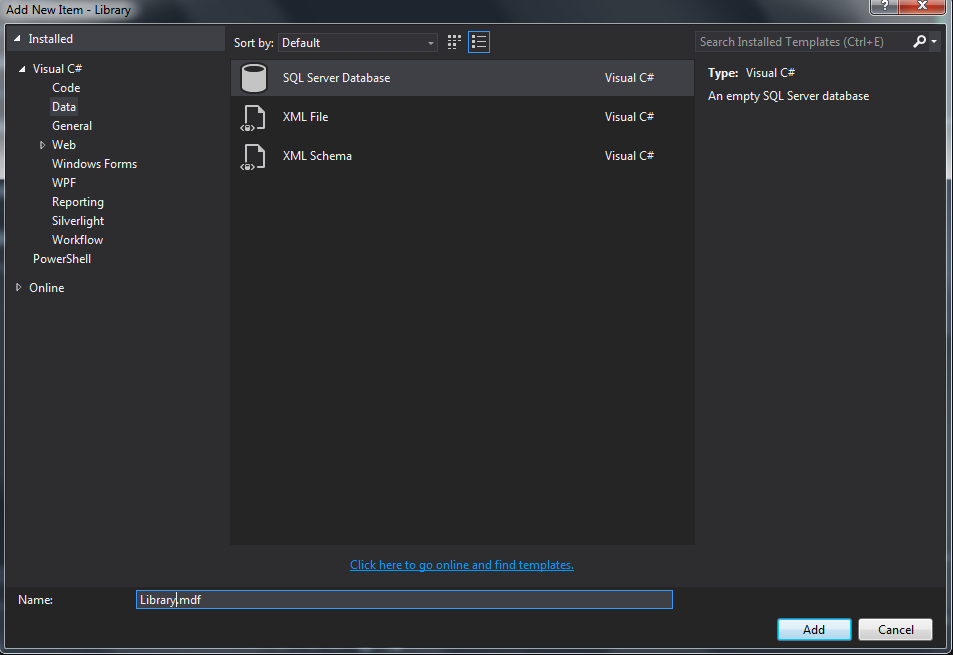
public string Address { get; set; }

public int BookId { get; set; }

public DateTime Date { get; set; }

}

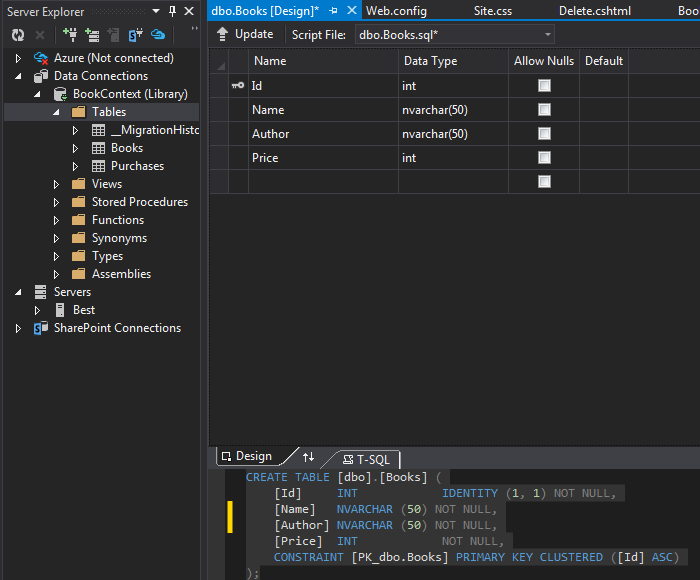
Для зберігання баз даних проекті у нас призначена папка App\_Data. Для цього натиснемо правою кнопкою миші на папку App\_Data і в контекстному меню виберемо Add-> New Item .... У вікні додавання нового елемента виберемо SQL Server Database і назвемо нову базу даних Library.mdf (рис. 3.1).



**Рис. 3.1- Створення бази даних**

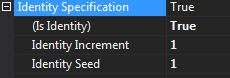
Розкриємо вузол Library.mdf і знайдемо вузол Tables. Натиснемо на цей вузол правою кнопкою миші і в меню виберемо пункт Add New Table.

Отже, створимо структуру таблиці Books (рис. 3.2).



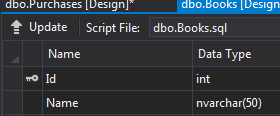
**Рис. 3.2 - Створення структури таблиці Books**

Не забудьте встановити нижче у вікні Properties для стовпця Id відповідні параметри для первинного ключа (рис.3.3).



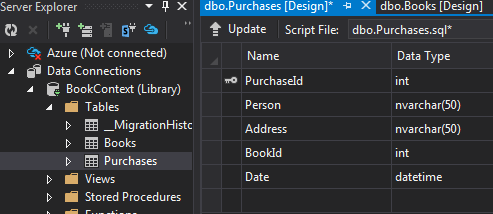
**Рис. 3.3 - Параметри для первинного ключа**

Необхідно згенерувати таблицю на основі заданого вище визначення. Для цього натиснемо на кнопку Update (рис.3.4).



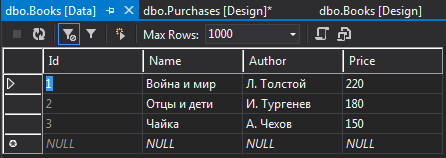
**Рис. 3.4 - Генерація таблиці**

У діалоговому вікні натиснемо на кнопку Update Database. В базу даних додається тільки що згенерувана таблиця. Подібним чином визначимо таблицю Purchases для моделі Purchase (рис.3.5).



**Рис. 3.5 - Генерація таблиць Books і Purchases**

Додамо в таблицю Books кілька записів(рис.3.6).



**Рис. 3.6 – Заповнення таблиці Books**

Тепер, по-перше, щоб взаємодіяти з БД, нам потрібен клас контексту даних, нехай це буде наступний клас BookContext:

public class BookContext : DbContext

{

public DbSet<Book> Books { get; set; }

public DbSet<Purchase> Purchases { get; set; }

}

По-друге, визначимо рядок підключення до БД. Для цього відкриємо файл Web.config і додамо в кінець секції configuration визначення рядка підключення. Визначення рядка підключення буде виглядати наступним чином:

<configuration>

.........................

<connectionStrings>

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

</connectionStrings>

</configuration>

Тепер ми можемо отримати вміст таблиці Books в контролері Home:

public class HomeController : Controller

{

//

BookContext db = new BookContext();

public ActionResult Index()

{

return View(db.Books);

}

}

І вивести дані в представленні Index.cshtml:

@using Library.Models

@model IEnumerable<Book>

@{

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

}

<div>

<h3>Распродажа книг</h3>

<table>

<tr class="header">

<td><p>Назва книги</p></td>

<td><p>Автор</p></td>

<td><p>Ціна</p></td>

<td></td>

</tr>

@foreach (Book b in Model)

{

<tr>

<td><p>@b.Name</p></td>

<td><p>@b.Author</p></td>

<td><p>@b.Price</p></td>

<td><p><a href="/Home/Buy/@b.Id">Купити</a></p></td>

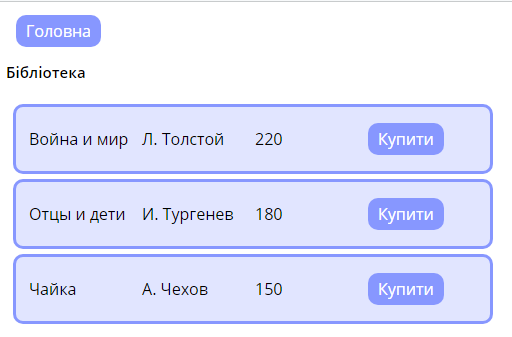
</tr>

}

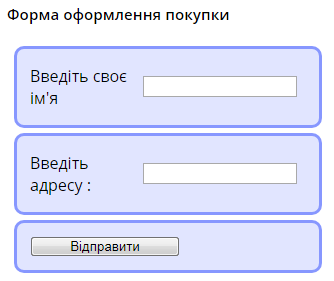
</table>

</div>

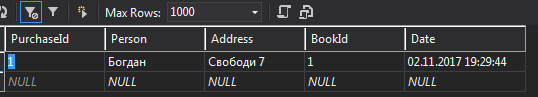
Результат роботи програми наведено на рисунках 3.7-3.9.



**Рис. 3.7- Вміст таблиці Books**



**Рис. 3.8- Форма оформлення покупки**



**Рис. 3.9-Занесені в таблицю Purchases результати покупки**

Розглянемо приклад використання шаблонного хелпера.

Визначимо в контролері дію BookView, яке по id буде виводити інформацію про певну книгу:

public ActionResult BookView(int id)

{

var book = db.Books.Find(id);

if (book != null)

{

return View(book);

}

return RedirectToAction("Index");

}

Тепер створимо представлення BookView.cshtml, в якому всі поля генеруються одним хелпером:

@{

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

}

@model Library.Models.Book

<h2>Книга № @Model.Id</h2>

@Html.DisplayForModel()

І звернемося до цього ресурсу, набравши в адресному рядку браузера Home / BookView / 1 (рис.3.10), або додати рядок в представлення Index.cshtml:

@foreach (Book b in Model)

{

<tr>

<td><p>@b.Name</p></td>

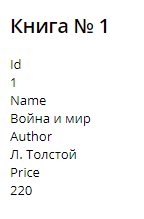
<td><p>@b.Author</p></td>

<td><p>@b.Price</p></td>

<td><p><a href="/Home/Buy/@b.Id">Купити</a></p></td> <td><p><ahref="/Home/BookView/@b.Id">Перегляд</a></p></td>

</tr>

}



**Рис. 3.10- Генерація результату одним хелпером**

Редагування моделі. Розглянемо, як зробити саму логіку 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 Library.Models.Book

<h2>Книга № @Model.Id</h2>

@using (Html.BeginForm("EditBook", "Home", FormMethod.Post))

{

<fieldset>

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

<p>

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

<br />

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

</p>

<p>

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

<br />

@Html.EditorFor(m => m.Author)

</p>

<p>

@Html.LabelFor(m => m.Price, "Ціна")

<br />

@Html.EditorFor(m => m.Price)

</p>

<p><input type="submit" value="Відпрвити" /></p>

</fieldset>

}

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:

[HttpPost]

public ActionResult EditBook(Book book)

{

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

db.SaveChanges();

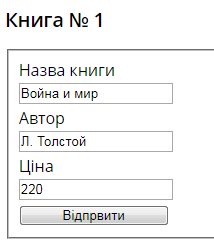
return RedirectToAction("Index");

}

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. Після чого перенаправлення на головну сторінку.

Варто відзначити, що хоча Entity Framework дозволяє нам абстрагуватися від запитів SQL і структури БД, але на низькому рівні, коли ми встановлюємо значення db.Entry(book).State = EntityState.Modified;, то ми тим самим вказуємо методу db.SaveChanges (), що треба згенерувати і виконати команду UPDATE для оновлення моделі в БД.

Звернемося до методу EditBook, наприклад, Home/EditBook/1 (рис.3.11).



**Рис. 3.11- Хелпер Html.EditorFor згенерував поля для редагування**

Хелпер Html.EditorFor згенерував нам поля для редагування. Ми можемо змінити модель, і відправити її на сервер, де відбудеться її збереження (рис. 3.12).



**Рис. 3.12 – Результати редагування**

Додавання моделі. Продовжимо роботу з моделлю Book і тепер подивимося, як ми можемо її додати і видалити з БД. 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";

}

<h2>Нова книга</h2>

@using (Html.BeginForm())

{

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

<br />

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

<br /><br />

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

<br />

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

<br /><br />

@Html.LabelFor(model => model.Price, "Ціна")

<br />

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

<br /><br />

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

}

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)

{

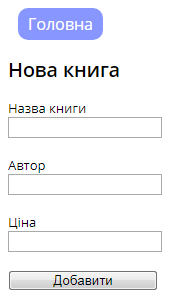
db.Entry (book) .State = EntityState.Added;

db.SaveChanges ();

return RedirectToAction ("Index");

}

Результат додавання книги показано на рисунку 3.13.



**Рис. 3.13- Додавання моделі в БД**

Видалення моделі. Тепер найважливіша частина - видалення моделі. Навіть не в плані реалізації, скільки в плані безпеки. Додамо просту дію, яке видаляє модель 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, 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 (рис. 3.14). І Id піде другим методом за запитом POST. Таким чином, ми відійдемо від вразливості GET-запиту. Ну і саме представлення:

@{

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

}

@model Library.Models.Book

<h2>Видалення книги</h2>

<dl>

<dt>Назва</dt>

<dd>

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

</dd>

<dt>Автор</dt>

<dd>

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

</dd>

<dt>Ціна</dt>

<dd>

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

</dd>

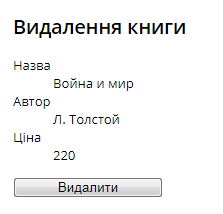
</dl>

@using (Html.BeginForm())

{

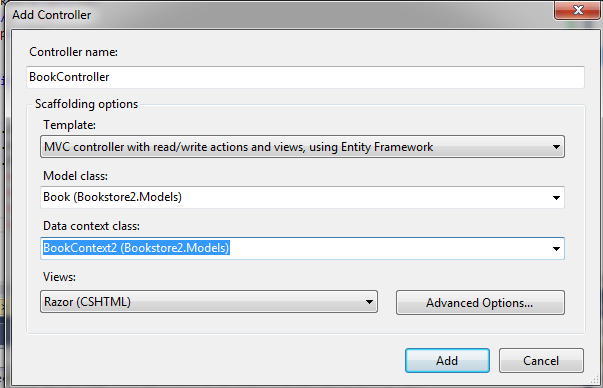
<input type="submit" value="Видалити" />

}



**Рис. 3.14 – Підтвердження на видалення книги**

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. Додамо новий контролер. У вікні створення нового контролера встановимо наступні параметри (рис. 3.15).



**Рис. 3.15 – Шаблони формування**

**Контрольні запитання:**

1. Моделі представляють собою прості класи і розташовуються в проекті в каталозі Models. Моделі описують логіку даних.
2. Дані моделей зберігаються в базі даних. Щоб взаємодіяти з базою даних, дуже зручно користуватися фреймворком Entity Framework.
3. Контекст даних являє собою клас, похідний від класу DbContext. Контекст даних містить одне або кілька властивостей типу DbSet<Т>, де Т представляє тип об'єкта, що зберігається в базі даних.
4. 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.
5. DisplayForModel - створює поля для читання для всіх властивостей моделі: Html.DisplayForModel ();

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

1. Ми можемо створити базу даних рівнозначним чином і на сервері. Після цього база даних додається в проект, і ми можемо побачити її в папці App\_Data. Тепер в оглядачі баз даних (вікно Database Explorer) ми можемо підключитися до неї і створити таблиці, які будуть зберігати дані. Розкриємо вузол Bookstore.mdf і знайдемо вузол Tables.
2. Рядок підключення

<configuration>

.........................

<connectionStrings>

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

</connectionStrings>

</configuration>

1. Це файл конфігурації, що задає глобальні параметри для проекту, вказує на компоненти, що у ньому використані та підключення до БД.
2. Вираз | DataDirectory | представляє заступник, який вказує, що база даних буде створюватися в проекті в папці App\_Data.
3. Розглянемо, як зробити саму логіку редагування моделі. Нехай в деякій дії контролера ми отримуємо об'єкт моделі по Id і виводимо її поля для редагування в представленні.

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.

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. Після чого перенаправлення на головну сторінку.

Варто відзначити, що хоча Entity Framework дозволяє нам абстрагуватися від запитів SQL і структури БД, але на низькому рівні, коли ми встановлюємо значення db.Entry(book).State = EntityState.Modified;, то ми тим самим вказуємо методу db.SaveChanges (), що треба згенерувати і виконати команду UPDATE для оновлення моделі в БД.

1. Видалення моделі. Тепер найважливіша частина - видалення моделі. Навіть не в плані реалізації, скільки в плані безпеки. Додамо просту дію, яке видаляє модель з бази даних

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:

1. Перший метод повертає користувачеві представлення з формою для додавання, а другий - приймає дані цієї форми.
2. Використовуэмо для полыв, що потрібно скрити та не редагувати.
3. В GET – дані відображаються явно, в POST вони заховані.
4. Вказує, що запис було модифіковано.
5. Вказує що запис новий.
6. Вказує, що запис було видалено.
7. Post допомагає уникнути вразливоті GET запиту.
8. 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.

**Висновок:** на даній лабораторній роботі я створив веб додаток, ознайомився з його можливостями, функціями, синтаксисом, простими операціями з моделями: редагування, видалення, додавання.