

ПРАКТИЧНА РОБОТА №4

Розробка WEB додатків на базі ASP.NET MVC



Реалізація моделей різної структури в ASP.NET MVC. Шаблони хелпери. Редагування, додавання і видалення моделі.



Отримати практичні навички реалізації моделей різної структури, а також реалізовувати модифікацію моделей в ASP.NET MVC.

ПОРЯДОК ВИКОНАННЯ ЗАВДАННЯ

Крім стандартних html-хелперів, розглянутих в минулій практичній роботі, які генерують певні елементи розмітки html, фреймворк MVC також має шаблонні (також називають шаблонізовані) хелпери. Навіщо вони потрібні? Вони більш гнучкі в порівнянні з html-хелперами, так як в даному випадку нам не треба думати, який треба створити елемент розмітки і який для цього вибрати хелпер. Ми просто повідомляємо шаблонного хелпера, яку властивість моделі ми хочемо використовувати, а фреймворк вже сам вибирає, який html-елемент згенерувати, виходячи з типу властивості і його метаданих.

Шаблони хелпери:**Display**

Створює елемент розмітки, який доступний тільки для читання, для зазначеної властивості моделі: `Html.Display ("Name")`

DisplayFor

Строго типізований аналог хелпера Display: `Html.DisplayFor (e => e.Name)`

Editor

Створює елемент розмітки, який доступний для редагування, зазначеної властивості моделі: `Html.Editor ("Name")`

EditorFor

Строго типізований аналог хелпера Editor: `Html.EditorFor (e => e.Name)`

DisplayText

Створює вираз для зазначеної властивості моделі у вигляді простої стрічки: `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 ()`

Представлення моделі

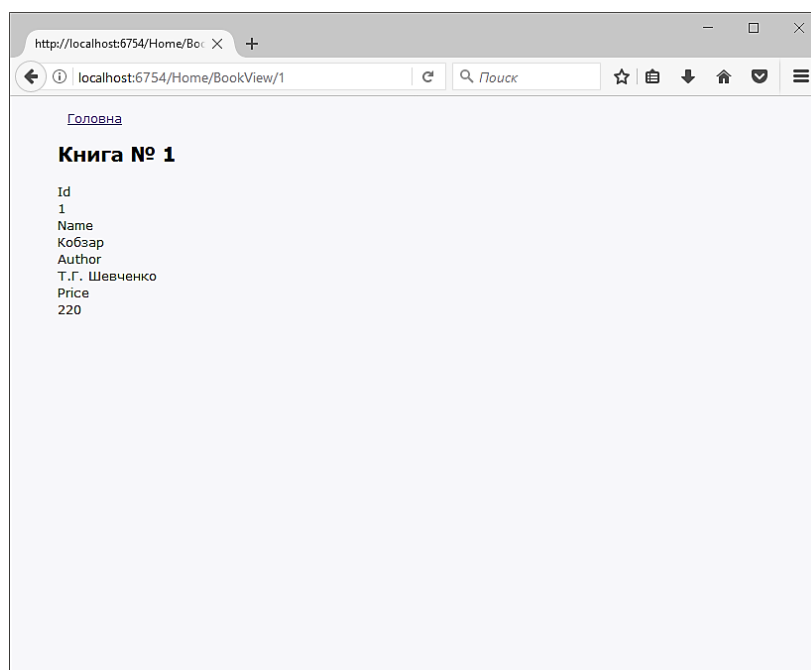
Наприклад, визначимо в контролері деяку дію BookView, яке по id буде виводити інформацію про певну книгу:

```
01 public ActionResult BookView(int id)
02 {
03     var book = db.Books.Find(id);
04     if (book != null)
05     {
06         return View(book);
07     }
08     return RedirectToAction("Index");
09 }
```

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

```
01 @{
02     Layout = "~/Views/Shared/_Layout.cshtml";
03 }
04
05 @model BookStore.Models.Book
06
07 <h2>Книга № @Model.Id</h2>
08 @Html.DisplayForModel()
```

І звернемося до цього ресурсу, набравши в адресному рядку браузера *Home / BookView / 1*:



Редагування моделі

В попередньому пункті ми побачили, що для редагування полів моделі зручно застосовувати хелпер *Editor / EditorFor*. Тепер подивимося, 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:

```
01 [HttpGet]
02 public ActionResult EditBook(int? id)
03 {
04     if (id == null)
05     {
06         return HttpNotFound();
07     }
08     Book book = db.Books.Find(id);
09
10     if (book == null)
11     {
12         return HttpNotFound();
13     }
14     return View(book);
15 }
```

На випадок, якщо користувачі не вкажуть *id*, ми встановлюємо в якості варіанту 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:

```
01 @{
02     ViewBag.Title = "Редагувати книгу";
03     Layout = "~/Views/Shared/_Layout.cshtml";
04 }
05 @model BookStore.Models.Book
06 <h2>Книга № @Model.Id</h2>
07 @using (Html.BeginForm("EditBook", "Home", FormMethod.Post))
08 {
09     <fieldset>
10         @Html.HiddenFor(m => m.Id)
11         <p>
```

```
12         @Html.LabelFor(m => m.Name, "Назва книги")
13         <br />
14         @Html.EditorFor(m => m.Name)
15     </p>
16     <p>
17         @Html.LabelFor(m => m.Author, "Автор")
18         <br />
19         @Html.EditorFor(m => m.Author)
20
21     </p>
22     <p>
23         @Html.LabelFor(m => m.Price, "Ціна")
24         <br />
25         @Html.EditorFor(m => m.Price)
26     </p>
27     <p><input type="submit" value="Відправити" /></p>
28 </fieldset>
29 }
```

Так як унікальний ідентифікатор `id` книги нам не потрібно редагувати, то поле для його відображення зробимо прихованим, тобто скористаємося хелпером **Html.HiddenFor**.

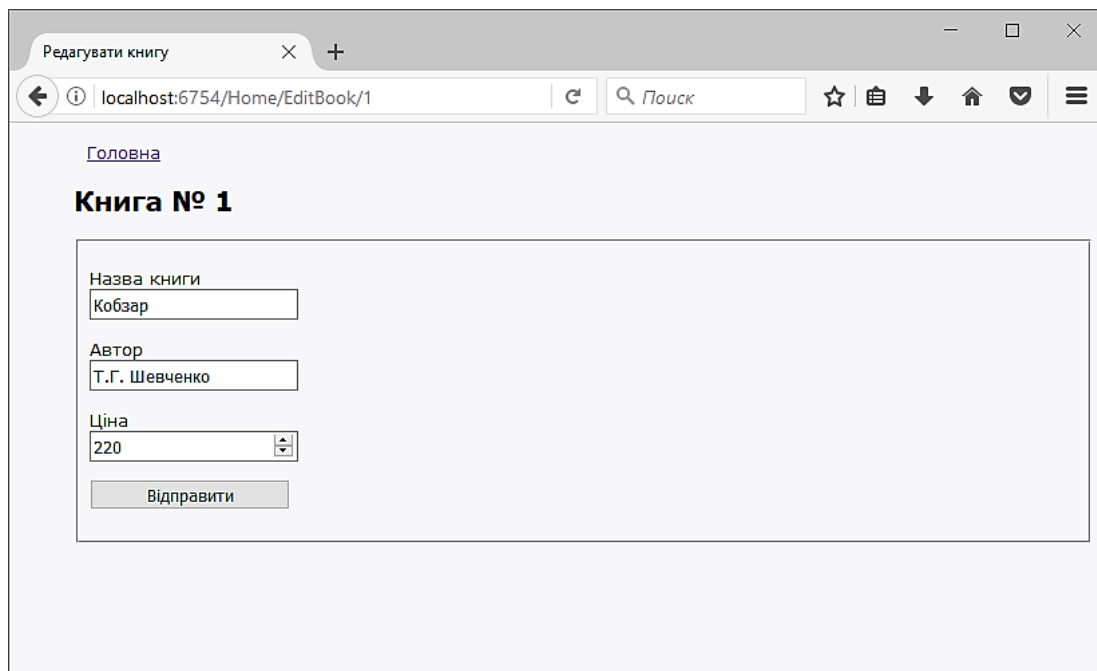
Тепер нам потрібен сам код збереження. Визначимо в контролері дію **EditBook**, яка буде обробляти POST-запити:

```
01 [HttpPost]
02 public ActionResult EditBook(Book book)
03 {
04     db.Entry(book).State = EntityState.Modified;
05     db.SaveChanges();
06     return RedirectToAction("Index");
07 }
```

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 перенаправляти 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*:



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

Додавання моделі

У попередніх прикладах ми розглянули, як редагувати модель. Продовжимо роботу з моделлю Book і тепер подивимося, як ми можемо її додавати і видаляти з БД. Для додавання моделі спочатку визначимо пару дій:

```
01 [HttpGet]
02 public ActionResult Create()
03 {
```

```
04     return View();
05 }
06 [HttpPost]
07 public ActionResult Create(Book book)
08 {
09     db.Books.Add(book);
10     db.SaveChanges();
11
12     return RedirectToAction("Index");
13 }
```

Перший метод повертає користувачеві представлення з формою для додавання, а другий – приймає дані цієї форми. Тепер створимо представлення.

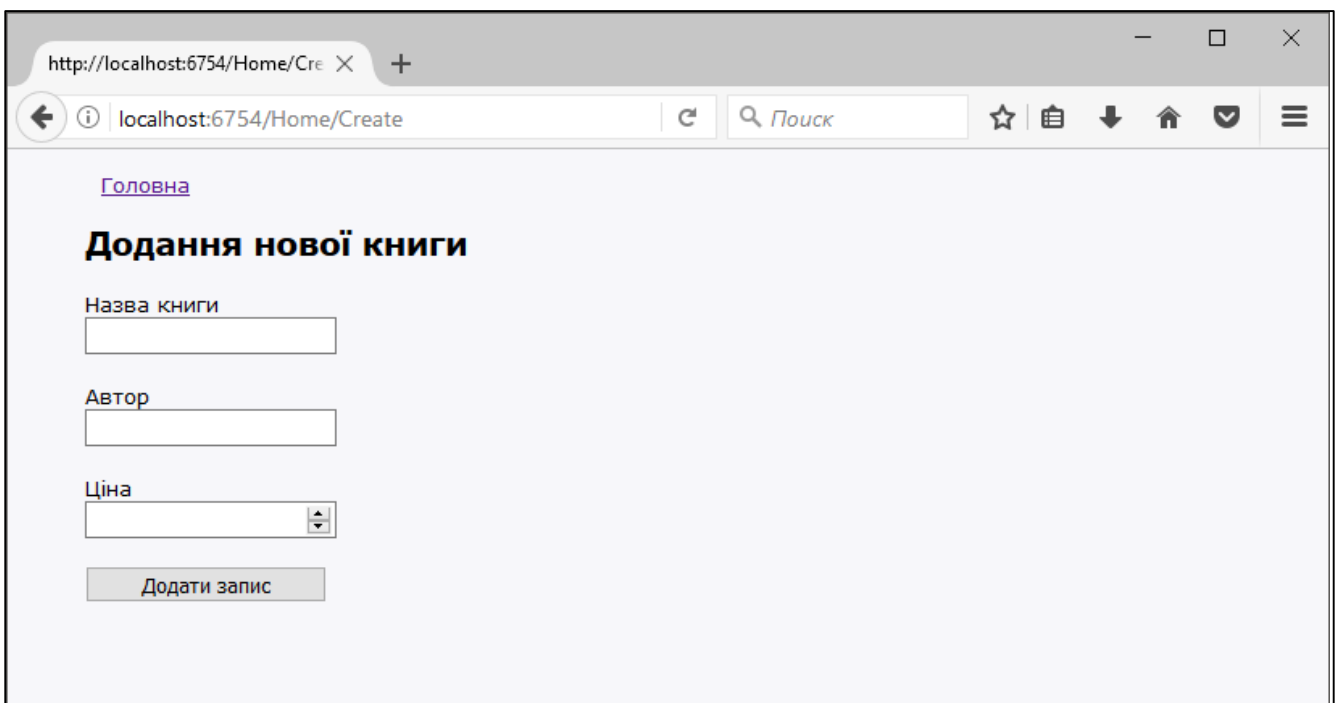
Представлення буде виглядати так:

```
01 @model BookStore.Models.Book
02
03 @{
04     Layout = "~/Views/Shared/_Layout.cshtml";
05 }
06
07 <h2>Новая книга</h2>
08
09 @using (Html.BeginForm())
10 {
11     @Html.LabelFor(model => model.Name, "Назва книги")
12     <br />
13     @Html.EditorFor(model => model.Name)
14     <br /><br />
15     @Html.LabelFor(model => model.Author, "Автор")
16     <br />
17     @Html.EditorFor(model => model.Author)
18     <br /><br />
19     @Html.LabelFor(model => model.Price, "Ціна")
20     <br />
21     @Html.EditorFor(model => model.Price)
22     <br /><br />
23     <input type="submit" value="Додати запис" />
24 }
```

Для отримання моделі **book** задіюють **Create** метод *db.Books.Add (book)* буде встановлювати значення **Added** як стану моделі. Тому метод *db.SaveChanges*

() згенерує вираз **INSERT** для вставки моделі в таблицю. Тобто метод **Create** ми могли б переписати таким чином:

```
01 [HttpPost]
02 public ActionResult Create(Book book)
03 {
04     db.Entry(book).State = EntityState.Added;
05     db.SaveChanges();
06
07     return RedirectToAction("Index");
08 }
```



Видалення моделі

Тепер найважливіша частина – видалення моделі. Навіть не в плані реалізації, скільки в плані безпеки. Додамо просту дію, яка видаляє модель з бази даних:

```
01 public ActionResult Delete(int id)
02 {
03     Book b = db.Books.Find(id);
04     if (b != null)
```



```
05     {
06         db.Books.Remove(b);
07         db.SaveChanges();
08     }
09     return RedirectToAction("Index");
10 }
```

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-запит **DELETE**. Але ми можемо самі вказати статус явно:

```
01 public ActionResult Delete(int id)
02 {
03     Book b = new Book { Id = id };
04     db.Entry(b).State = EntityState.Deleted;
05     db.SaveChanges();
06
07     return RedirectToAction("Index");
08 }
```

Подібний підхід має один плюс – ми уникаємо першого запиту до бд, який у нас був виражений як *Book b = db.Books.Find (id)* ;. Тобто замість двох запитів до БД тепер у нас тільки один. В цілому подібний метод на видалення має один мінус в плані безпеки.

Припустимо, нам прийшов електронний лист, в який була вложена картинка за допомогою тега:

```
<img src = "http: // адреса_нашого_сайту / Home / Delete / 1" />
```

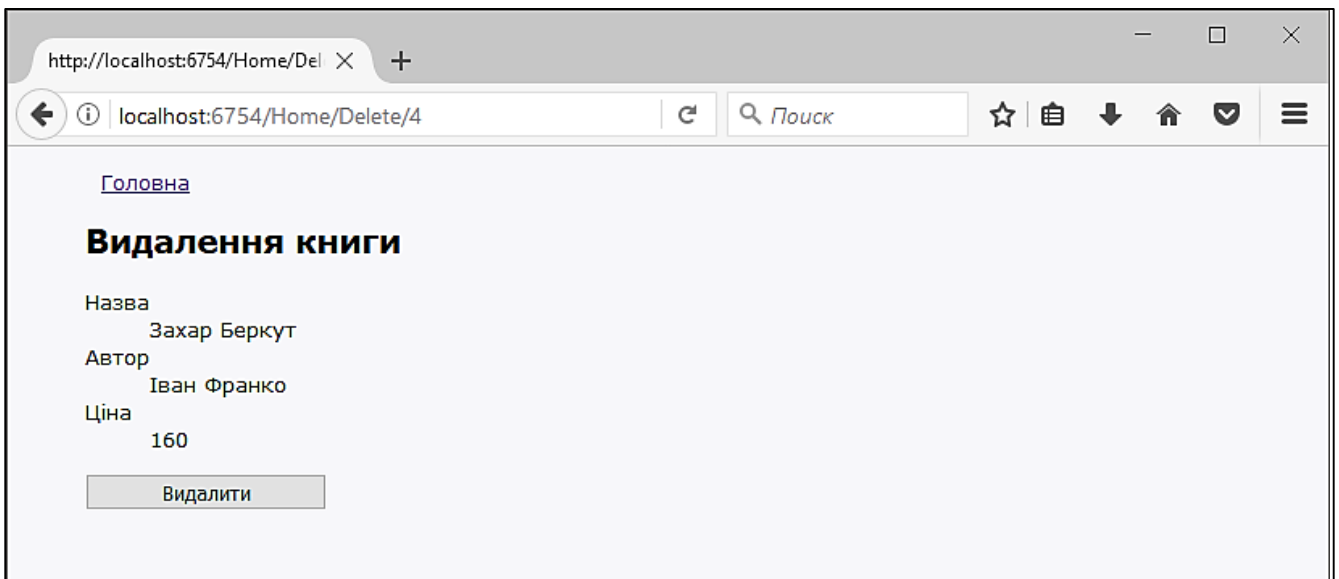
В результаті при відкритті листа 1-ший запис в таблиці може бути видалений. Вразливість відноситься не тільки до листів, але й може проявлятися і в інших місцях, але сенс один – **GET-запит** для методу **Delete** несе потенційну вразливість. Тому переробимо метод наступним чином:

```
01 [HttpGet]
02 public ActionResult Delete(int? id)
03 {
04     if (id == null)
05     {
06         return HttpNotFound();
07     }
08     Book b = db.Books.Find(id);
09     if (b == null)
10     {
11         return HttpNotFound();
12     }
13     return View(b);
14 }
15 [HttpPost, ActionName("Delete")]
16 public ActionResult DeleteConfirmed(int? id)
17 {
18     if (id == null)
19     {
20         return HttpNotFound();
21     }
22     Book b = db.Books.Find(id);
23     if (b == null)
24     {
25         return HttpNotFound();
26     }
27     db.Books.Remove(b);
28     db.SaveChanges();
29     return RedirectToAction("Index");
30 }
```

Тепер замість одного методу Delete аж два. Атрибут *ActionName("Delete")* вказує, що метод **DeleteConfirmed** буде сприйматися як дія Delete. Перший метод передає модель, що видаляється в представлення. А вже в представленні за допомогою натискання кнопки ми зможемо підтвердити видалення. І id піде другим методом за запитом **POST**. Таким чином, ми уникнемо небезпеки у GET-запиті. Представлення виглядає так:

```
01 @{
02     Layout = "~/Views/Shared/_Layout.cshtml";
03 }
04 @model BookStore.Models.Book
05 <h2>Видалення книги</h2>
```

```
06 <dl>
07     <dt>Назва</dt>
08     <dd>
09         @Html.DisplayFor(model => model.Name)
10     </dd>
11
12     <dt>Автор</dt>
13     <dd>
14         @Html.DisplayFor(model => model.Author)
15     </dd>
16
17     <dt>Ціна</dt>
18     <dd>
19         @Html.DisplayFor(model => model.Price)
20     </dd>
21 </dl>
22
23 @using (Html.BeginForm())
24 {
25     <input type="submit" value="Видалити" />
26 }
```



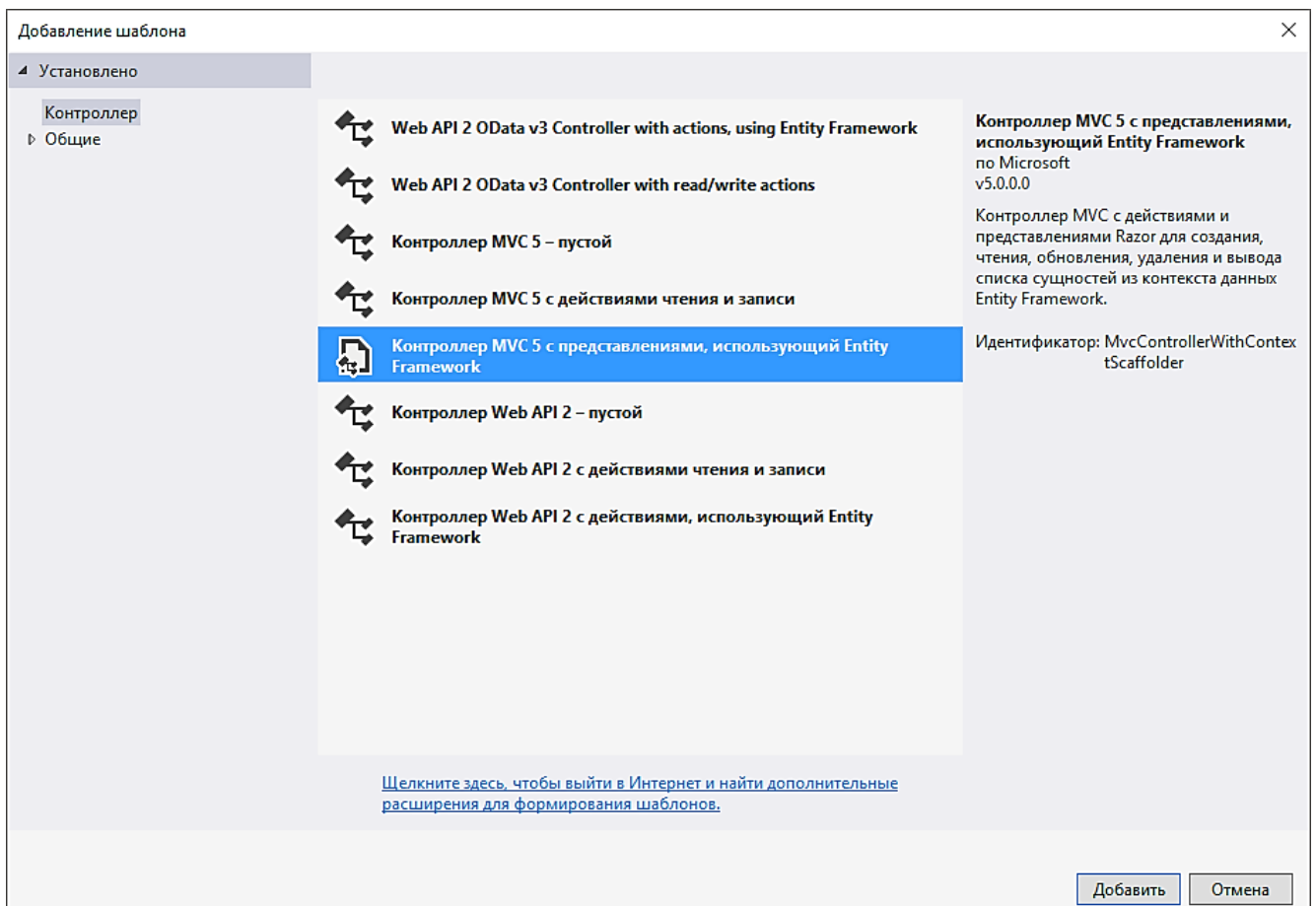
Шаблони формування

Оскільки найчастіше 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 за заданою моделлю

і контексту даних сформувати всю необхідну розмітку для представлень і контролера, за допомогою яких можна керувати записами в БД.

Тут треба зразу додати, що для коректного застосування шаблонів формування треба, щоб використовувалася одна із зв'язок MVC 4 + EntityFramework 5 або MVC 5 + Entity Framework 6.

Щоб використовувати дану функціональність, додамо новий контролер, натиснувши правою кнопкою миші на папку **Controllers** і виберемо **Додати -> Контролер...** У вікні, що з'явилося буде запропоновано вибрати шаблон контролера:



До MVC-проектів відносяться тільки три шаблони із списку:

– **MVC 5 Controller – Empty**. Цей шаблон додає у папку Controllers пустий контролер, який має один метод Index. Даний шаблон не створює представлень.

– **MVC 5 Controller with read/write actions.** Цей шаблон додає в проект контролер, який містить методи Index, Details, Create, Edit, Delete але ці методи не містять ніякої логіки роботи з базою даних і нам необхідно самим створювати для них бізнес-логіку та представлення.

– **MVC 5 Controller with views, using Entity Framework.** Це вже цікавіший шаблон, який створює контролер з методами Index, Details, Create, Edit, Delete а також всі необхідні представлення для цих методів та додає код для роботи з базою даних.

Тому у нашому випадку вибираємо шаблон **MVC 5 Controller with views, using Entity Framework.**

Після вибору шаблону нам відкриється вікно додання нового контролера, в якому нам необхідно додати деякі налаштування:

Controller name: ім'я контролера, встановимо його як BookController;

Use async controller actions: генерувати автоматично методи контролера асинхронними. Виберемо цю функцію;

Model class: модель, на основі якої будуть створюватися представлення. Виберемо створену раніше модель Book (або якусь іншу наявну модель).

Data context class: треба вибрати клас контексту даних, за допомогою якого ми отримуємо всі дані моделі з БД. У нашому випадку – це раніше створений клас BookContext.

Generate view: генерує представлення до створених методів контролера. При виборі цієї опції зразу ж будуть доступні дві інші опції;

Reference script libraries: підключення до представлень бібліотеки JQuery та інших файлів JavaScript;

Use a layout page: представлення, що генеруються будуть використовувати майстер-сторінку.

Добавление контроллера

Класс модели: Book (BookingAppStore.Models)

Класс контекста данных: BookContext (BookingAppStore.Models) +

☒ Использование асинхронных действий контроллера

Представления:

☒ Создать представления

☒ Библиотеки сценариев

☒ Использовать страницу макета:

~/Views/Shared/_Layout.cshtml ...

(Оставить пустым, если значение задано в файле Razor _viewstart)

Имя контроллера: BookController

Добавить Отмена

Встановивши всі необхідні параметри, натиснемо кнопку **Add**, і в проект буде додано новий контролер. Він буде виглядати приблизно так:

```
01 using System;
02 using System.Collections.Generic;
03 using System.Data;
04 using System.Data.Entity;
05 using System.Linq;
06 using System.Web;
07 using System.Web.Mvc;
08 using BookStoreApp.Models;
09
10 namespace BookStoreApp.Controllers
11 {
12     public class BookController : Controller
13     {
14         private BookContext db = new BookContext();
15
16         //
17         // GET: /Book/
18
19         public ActionResult Index()
20         {
21             return View(db.Books.ToList());
22         }
23     }
```

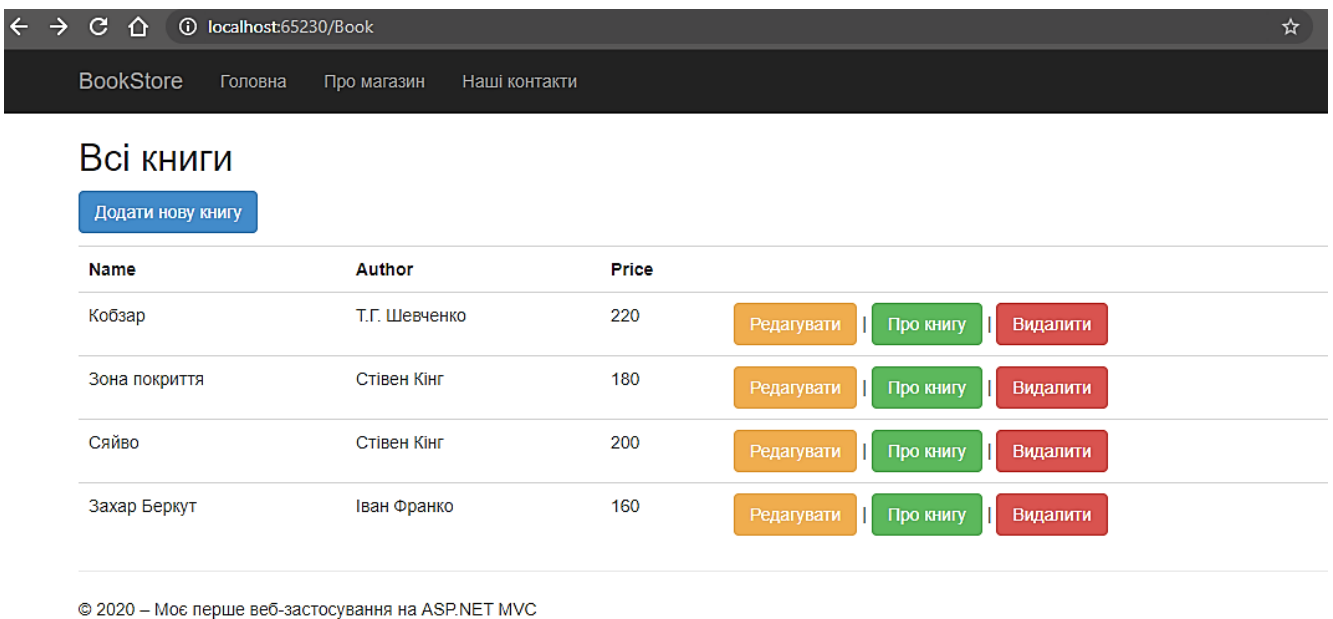
```
24      //
25      // GET: /Book/Details/5
26
27      public ActionResult Details(int id = 0)
28      {
29          Book book = db.Books.Find(id);
30          if (book == null)
31          {
32              return HttpNotFound();
33          }
34          return View(book);
35      }
36
37      //
38      // GET: /Book/Create
39
40      public ActionResult Create()
41      {
42          return View();
43      }
44
45      //
46      // POST: /Book/Create
47
48      [HttpPost]
49      public ActionResult Create(Book book)
50      {
51          if (ModelState.IsValid)
52          {
53              db.Books.Add(book);
54              db.SaveChanges();
55              return RedirectToAction("Index");
56          }
57
58          return View(book);
59      }
60
61      //
62      // GET: /Book/Edit/5
63
64      public ActionResult Edit(int id = 0)
65      {
66          Book book = db.Books.Find(id);
67          if (book == null)
68          {
69              return HttpNotFound();
70          }
71          return View(book);
```

```
72     }
73
74     //
75     // POST: /Book/Edit/5
76
77     [HttpPost]
78     public ActionResult Edit(Book book)
79     {
80         if (ModelState.IsValid)
81         {
82             db.Entry(book).State = EntityState.Modified;
83             db.SaveChanges();
84             return RedirectToAction("Index");
85         }
86         return View(book);
87     }
88
89     //
90     // GET: /Book/Delete/5
91
92     public ActionResult Delete(int id = 0)
93     {
94         Book book = db.Books.Find(id);
95         if (book == null)
96         {
97             return HttpNotFound();
98         }
99         return View(book);
100    }
101
102    //
103    // POST: /Book/Delete/5
104
105    [HttpPost, ActionName("Delete")]
106    public ActionResult DeleteConfirmed(int id)
107    {
108        Book book = db.Books.Find(id);
109        db.Books.Remove(book);
110        db.SaveChanges();
111        return RedirectToAction("Index");
112    }
113
114    protected override void Dispose(bool disposing)
115    {
116        db.Dispose();
117        base.Dispose(disposing);
118    }
119 }
```



```
120 | }
```

А в папці Views / Book ми побачимо всі необхідні представлення з усім необхідним кодом, який тепер нам не треба набирати вручну. ми можемо запустити проект і перейти в адресному рядку браузера до нашого контролера, щоб переконатися, що все працює як треба.



Name	Author	Price	
Кобзар	Т.Г. Шевченко	220	Редагувати Про книгу Видалити
Зона покриття	Стівен Кінг	180	Редагувати Про книгу Видалити
Сяйво	Стівен Кінг	200	Редагувати Про книгу Видалити
Захар Беркут	Іван Франко	160	Редагувати Про книгу Видалити

© 2020 – Моє перше веб-застосування на ASP.NET MVC

Завдяки шаблонам формування ми можемо не думати про створення коду для стандартних операцій. Нам залишається після генерації коду лише змінити автоматично згенеровані назви на свої (наприклад, назва сторінки, посилання які автоматично генеруються і т.д.).

Завдання

Реалізувати вище наведені приклади до функціоналу свого веб-додатку, розробленого на попередніх практичних роботах.