**ЛАБОРАТОРНА РОБОТА №3**

***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.

**3.1 Теоретичні відомості**

Всі сутності в додатку прийнято виділяти в окремі моделі. Залежно від поставленого завдання і складності програми можна виділити різну кількість моделей. Так, в тестовому додатку з першої лабораторної використовувалися дві моделі - клас для книги і клас для покупки книги. Моделі представляють собою прості класи і розташовуються в проекті в каталозі Models. Моделі описують логіку даних. Наприклад, модель представляє книгу і її покупку:

public class Book

{

// ID книги

public int Id { get; set; }

// назва книги

public string Name { get; set; }

// автор книги

public string Author { get; set; }

// ціна

public int Price { get; set; }

}

public class Purchase

{

// ID покупки

public int PurchaseId { get; set; }

// ім’я і прізвище покупця

public string Person { get; set; }

// адрес покупця

public string Address { get; set; }

// ID книги

public int BookId { get; set; }

// дата покупки

public DateTime Date { get; set; }

}

Модель необов'язково складається тільки з властивостей, крім того, вона може мати конструктор, які-небудь допоміжні методи. Але головне не перевантажувати клас моделі і пам'ятати, що його призначення - описувати дані. Маніпуляції з даними і бізнес-логіка - це більше сфера контролера. Дані моделей зберігаються в базі даних. Щоб взаємодіяти з базою даних, дуже зручно користуватися фреймворком Entity Framework. Entity Framework підтримує підхід "Code first", який передбачає збереження або видалення інформації з БД на SQL Server без створення схеми бази даних або використання дизайнера в Visual Studo. Навпаки, 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. 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. Припустимо, створимо контекст даних для роботи з вищенаведеними моделями Book і Purchase:

public class BookContext : DbContext

{

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

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

}

За допомогою властивостей Books і Purchases ми отримуємо доступ до даних відповідних моделей, які зберігаються в базі даних.

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.

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 ()

**3.2. Хід роботи**

**Підключення до бази даних.** Щоб зберігати дані, нам потрібна база даних. Як правило, в якості бази даних використовується MS SQL Server, на прикладі якого ми і подивимося весь 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, 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; }

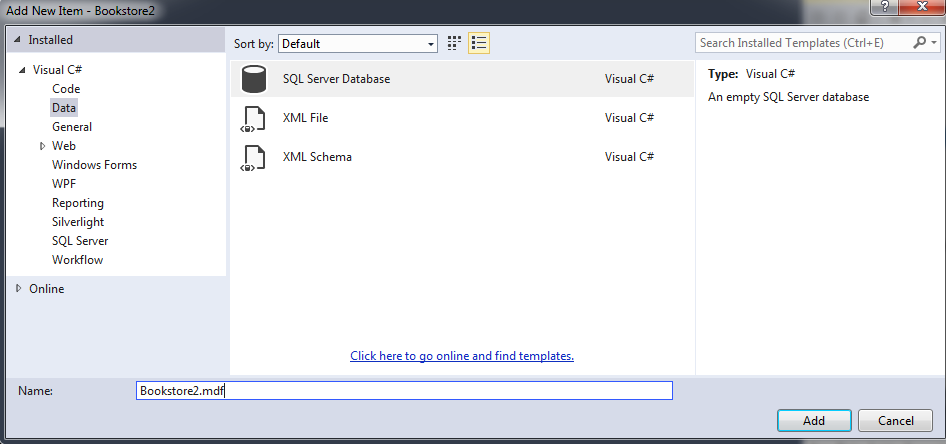
public string Address { get; set; }

public int BookId { get; set; }

public DateTime Date { get; set; }

}

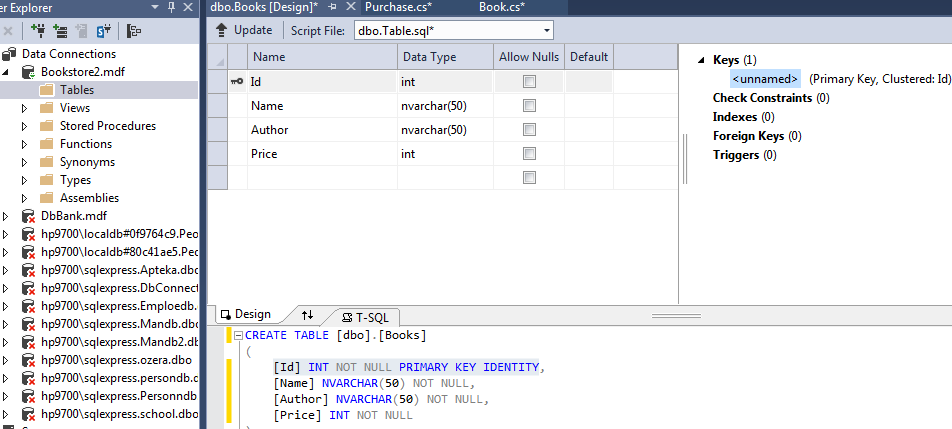
Для зберігання баз даних проекті у нас призначена папка App\_Data. Для цього натиснемо правою кнопкою миші на папку App\_Data і в контекстному меню виберемо Add-> New Item .... У вікні додавання нового елемента виберемо SQL Server Database і назвемо нову базу даних Bookstore2.mdf (рис. 3.1).



**Рис. 3.1- Створення бази даних**

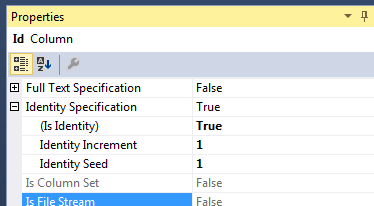
Ми можемо створити базу даних рівнозначним чином і на сервері. Після цього база даних додається в проект, і ми можемо побачити її в папці App\_Data. Тепер в оглядачі баз даних (вікно Database Explorer) ми можемо підключитися до неї і створити таблиці, які будуть зберігати дані. Розкриємо вузол Bookstore.mdf і знайдемо вузол Tables. Натиснемо на цей вузол правою кнопкою миші і в меню виберемо пункт Add New Table. І перед нами з'явиться вікно, в якому нам треба визначити назви і типи стовпців нової таблиці. За угодами про найменуваннях таблиці при роботі з Entity Framework повинні відповідати імені моделі. Тобто, так як наша модель називається Book, то таблиця буде називатися Books. А Entity Framework автоматично розпізнає, що таблиця Books відповідає класу Book.

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



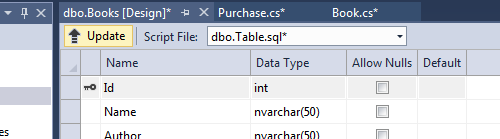
**Рис. 3.2 - Створення структури таблиці Books**

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



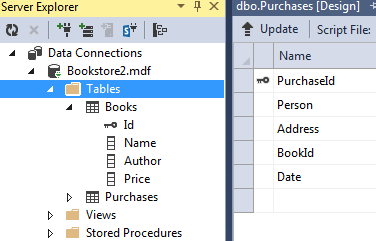
**Рис. 3.3 - Параметри для первинного ключа**

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



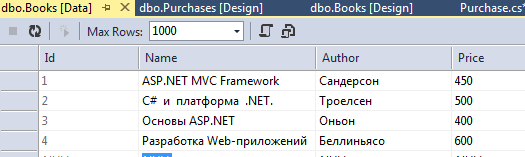
**Рис. 3.4 - Генерація таблиці**

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 BookContext2:

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

</connectionStrings>

</configuration>

Використання підстановки | DataDirectory | дозволяє опустити повний фізичний шлях до бази даних, яка зберігається в папці App\_Data.Тепер ми можемо отримати вміст таблиці Books в контролері Home:

public class HomeController : Controller

{

//

BookContext2 db = new BookContext2();

public ActionResult Index()

{

return View(db.Books);

}

}

І вивести дані в представленні Index.cshtml:

@using Bookstore2.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>

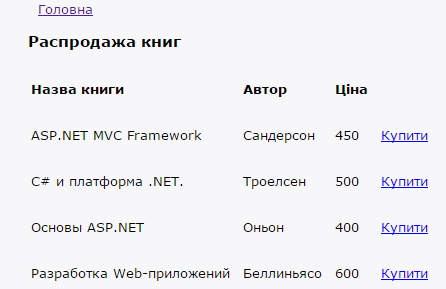
</tr>

}

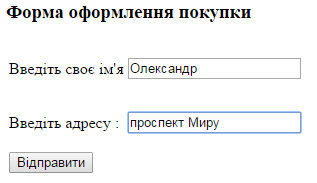
</table>

</div>

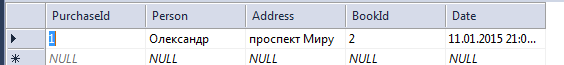
Результат роботи програми наведено на рисунках 3.7-3.9.



**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 Bookstore2.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 Bookstore2.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();

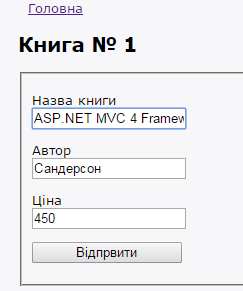
return RedirectToAction("Index");

}

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)

{

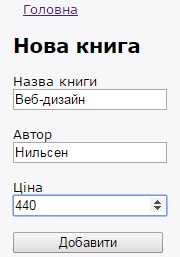
db.Entry (book) .State = EntityState.Added;

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 Bookstore2.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>

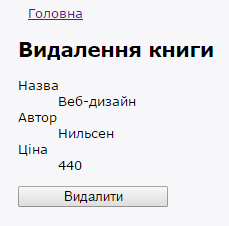
</dl>

@using (Html.BeginForm())

{

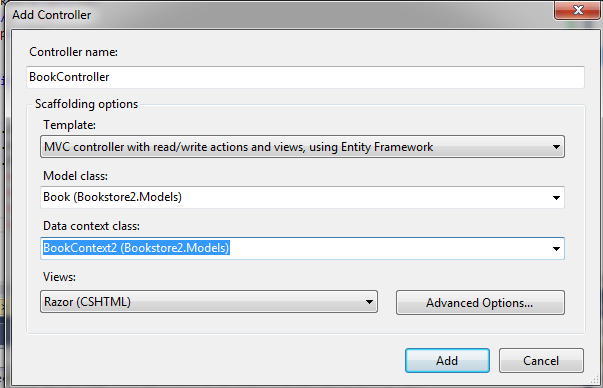
<input type="submit" value="Видалити" />

}



**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. *Для коректного застосування шаблонів формування необхідно, щоб використовувалася одна з зв'язок MVC 4+ EntityFramework 5 або MVC 5 + Entity Framework 6.* Отже, застосуємо шаблон формування. Додамо новий контролер. У вікні створення нового контролера встановимо наступні параметри (рис. 3.15).



**Рис. 3.15 – Шаблони формування**

У секції Template нам пропонується вибрати шаблон формування коду. За замовчуванням нам доступні наступні шаблони:

- Empty MVC Controller. Цей шаблон додає в папку Controllers клас, похідний від класу Controller з вказаним ім'ям. Єдина дія, яку виконує цей контролер - це дія Index, яка за замовчуванням не має ніякого функціоналу. Даний шаблон не створює представлень.

- MVC Controller with Empty Read / Write Actions. Цей шаблон додає в проект контролер, який містить методи Index, Details, Create, Edit і Delete. Однак ці методи фактично ніякої корисної роботи не виконують, і необхідно самим створити для них код та представлення для кожного з цих методів.

- MVC Controller with Read / Write Actions and Views, Using Entity Framework. Цей шаблон створює контролер з методами Index, Details, Create, Edit і Delete, а також всі необхідні представлення для цих дій і додає код для добування інформації з бази даних.

- Empty API Controller. Цей шаблон порожній контролер, похідний від класу ApiController. Клас ApiController був введений в MVC 4 і використовується в проектах типу Web API - свого роду веб-служб.

- API Controller with Empty Read / Write Actions. Цей шаблон додає в проект контролер, похідний від класу ApiController, який містить методи Index, Details, Create, Edit і Delete.

- API Controller with Read / Write Actions and Views, Using Entity Framework. Цей шаблон, також як і у випадку зі звичайним контролером, створює контролер, тільки в даному випадку похідний від класу ApiController, який містить методи Index, Details, Create, Edit і Delete, а також всі необхідні представлення для цих дій.

У нашому випадку ми вибираємо пункт MVC Controller with Read / Write Actions and Views, Using Entity Framework.

Далі нам буде запропоновано вибрати в полі Model class модель, на основі якої будуть створюватися представлення. Виберемо створену раніше модель Book (або якусь іншу наявну модель). На завершення в поле 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. Встановивши всі необхідні параметри, натиснемо кнопку Add, і в проект буде додано новий контролер. А в папці Views/Book ми побачимо всі необхідні представлення з усім необхідним кодом, який тепер нам не треба набирати вручну. Можемо запустити проект і перейти в адресному рядку браузера до нашого контролера, щоб переконатися, що все працює як треба.

**3.3 Завдання до лабораторної роботи**

1. Розробити web – додаток, що буде представляти примітивну емуляцію магазину по продажу комп’ютерної техніки, в якому необхідно реалізувати операції - редагування, видалення, додавання.

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

3. Розробити web – додаток, що буде представляти примітивну емуляцію довідкового бюро залізничного вокзалу по продажу білетів, в якому необхідно реалізувати операції - редагування, видалення, додавання.

4. Розробити web – додаток, що буде представляти примітивну емуляцію складу по продажу товару сільськогосподарської продукції, в якому необхідно реалізувати операції - редагування, видалення, додавання.

5. Розробити web – додаток, що буде представляти примітивну емуляцію довідкового бюро автовокзалу по продажу білетів, в якому необхідно реалізувати операції - редагування, видалення, додавання..

6. Розробити web – додаток, що буде представляти примітивну емуляцію салону по продажу мобільної техніки, в якому необхідно реалізувати операції - редагування, видалення, додавання.

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

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

9. Розробити web – додаток, що буде представляти примітивну емуляцію роботи банка по наданню кредитів, в якому необхідно реалізувати операції - редагування, видалення, додавання.

10. Розробити web – додаток, що буде представляти примітивну емуляцію салону по продажу автомобілів, в якому необхідно реалізувати операції - редагування, видалення, додавання.

11. Розробити web – додаток, що буде представляти примітивну емуляцію роботи кінотеатрів міста, в якому необхідно реалізувати операції - редагування, видалення, додавання.

12. Розробити web – додаток, що буде представляти примітивну емуляцію регістрації ДТП, в якому необхідно реалізувати операції - редагування, видалення, додавання.

13. Розробити web – додаток, що буде представляти примітивну емуляцію регістрацію порушення правил дорожнього руху робітниками ДАІ. Необхідно реалізувати операції - редагування, видалення, додавання.

14. Розробити web – додаток, що буде представляти примітивну емуляцію виставлення оцінок за результатами екзаменаційної сесії. Необхідно реалізувати операції - редагування, видалення, додавання.

15. Розробити web – додаток, що буде представляти примітивну емуляцію виконаних дипломних робіт студентами. Необхідно реалізувати операції - редагування, видалення, додавання.

**3.4. Контрольні питання**

1. Моделі в ASP.NET MVC 4
2. Взаємодія з базою даних
3. Контекст даних в ASP.NET MVC 4
4. Шаблонні хелпери
5. Шаблонні хелпери для певної моделі
6. Процес створення БД і підключення до неї
7. Визначення рядка підключення
8. Файл Web.config
9. Використання підстановки | DataDirectory |
10. Редагування моделі
11. Видалення моделі
12. Вставка моделі
13. Хелпер Html.HiddenFor
14. Get і POST-запити
15. Рядок db.Entry(book).State = EntityState.Modified
16. Рядок db.Entry (book) .State = EntityState.Added
17. Рядок db.Entry(b).State = EntityState.Deleted
18. Get і POST-запити при видалені моделі
19. Шаблони формування

**Література:** [1,3,4,7,8,9].