**ЛАБОРАТОРНА РОБОТА №1**

***Тема.*** Створення примітивного додатка на ASP.NET MVC 4.

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

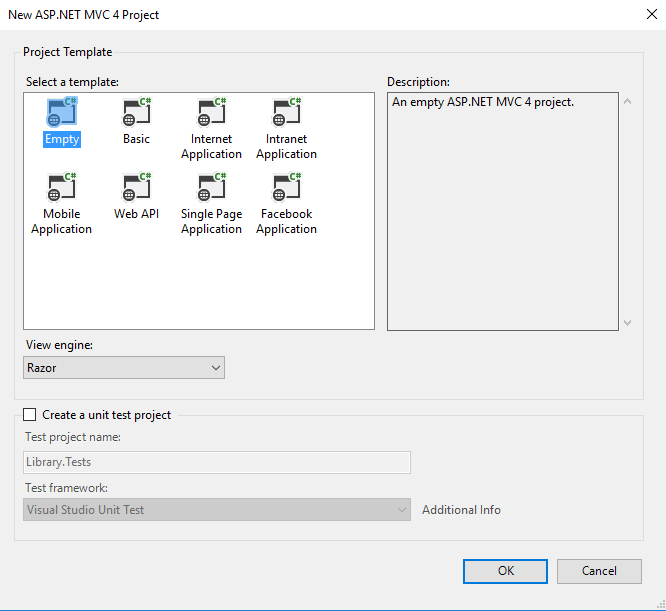
**1.1 Завдання до лабораторної роботи**

Розробити web – додаток, що буде представляти примітивну емуляцію роботи бібліотеки, в якому ми зможемо переглядати список книг і оформляти видачу.

**1.2. Хід роботи**

Створимо перший примітивний додаток на MVC 4, який повинен дати деяке початкове представлення роботи MVC. Як шаблон проекту ми виберемо шаблон Empty і створимо міні-додаток з нуля. Додаток буде представляти примітивну емуляцію книжкового магазину, в якому ми зможемо переглядати список книг і оформляти покупку.

Відкриємо Visual Studio, в меню виберемо File -> New Project .. і серед шаблонів проектів виберемо шаблон ASP.NET MVC 4 Web Application і назвемо новий проект, наприклад, Library. Потім у вікні створення нового проекту ASP.NET MVC 4 Web Application виберемо шаблон Empty (рис.1.4).



**Рис. 1.1 - Вікно вибору шаблону нового додатка**

Після цього буде створений порожній проект, в якому ще нічого немає. Нам належить наповнити його.

**Створення моделей.** Для нашого завдання ми можемо виділити дві області даних: інформація про книгу та інформація, пов'язана з оформленням покупки. Відповідно, ми створимо дві моделі. Отже, в нашому додатку вже визначена папка *Models.* У ній будуть знаходитися наші моделі. Натиснемо на цю папку правою кнопкою миші і в меню виберемо *Add-> Class* .... Назвемо новий клас або модель *Book* і додамо в нього код, що описує модель книги:

namespace Library.Models

{

public class Book

{

// ID книги

public int Id { get; set; }

// назва книги

public string Name { get; set; }

// автор книги

public string Author { get; set; }

// кількість

public int Quantity { get; set; }

} }

Таким же чином створимо клас для оформлення покупки - клас Purchase:

public class Purchase

{

// ID покупки

public int PurchaseId { get; set; }

// ім’я і прізвище покупця

public string Person { get; set; }

// адреса покупця

public string Address { get; set; }

// ID книги

public int BookId { get; set; }

// дата покупки

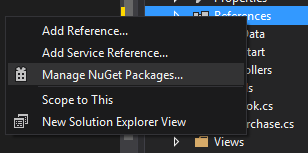
public DateTime Date { get; set; }

}

Умовності при створенні моделей. Оскільки база даних і відповідна їй модель повинна мати первинний ключ (Primary Key), то ми його повинні визначити. В даному випадку ми використовували визначення ключа за назвою властивості. Для цього модель повинна містити властивість з ім'ям Імя\_моделіId або просто Id. Так, у нас в моделі Purchase визначено властивість PurchaseId, тобто дана властивість є первинним ключем.

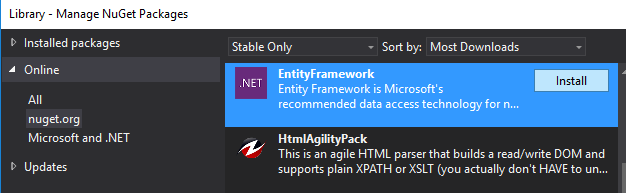
Створення контексту даних і EntityFramework. Для більш зручного доступу до даних рекомендується використовувати фреймворк Entity Framework, хоча його використання необов'язково і цілком залежить від уподобань розробника. Цей фреймворк дозволяє абстрагуватися від структури конкретної бази даних і вести всі операції з даними через модель.

Оскільки ми створили проект по шаблону Empty, який практично не містить жодних файлів, то нам треба буде додати в проект файли EntityFramework для взаємодії з БД. Якби ми створювали проект по іншому шаблоном, наприклад, Basic або Internet Application, то бібліотека EntityFramework вже була б додана за замовчуванням. Отже, додамо в проект підтримку EntityFramework і для цього натиснемо правою кнопкою миші в структурі проекту на вузол References (який містить підключення в проект бібліотеки) і в меню виберемо *Manage NuGet Packages* ...



**Рис. 1.2 - Підключення в проект бібліотеки**

У вікні управління пакетами NuGet в правому верхньому куті введіть у поле пошуку Entity і натисніть Enter. Після цього всі знайдені пакети, які мають відношення до запиту, будуть відображені в середньому стовпці, як показано на рисунку 1.6.



**Рис. 1.3 - Вікно управління пакетами NuGet**

Потім встановлюємо пакет, і після того, як закінчиться установка, ми можемо використовувати всі переваги EntityFramework для управлінням даними. Після установки створимо контекст даних. Він потрібен для полегшення доступу до БД на основі деякої моделі. Додамо в папку Models новий клас *BookContext* з наступним змістом:

using System.Data.Entity;

namespace Library.Models

{

public class BookContext : DbContext

{

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

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

}

Щоб створити контекст, нам треба наслідувати новий клас від класу *DbContext*. Властивості типу public *DbSet <Book> Books {get; set; }* Допомагають отримувати набір даних певного типу (в даному випадку типу *Book*). У результаті EntityFramework сам візьме на себе відповідальність по створенню бази даних. Тобто в даному випадку ми будемо використовувати так званий підхід Code First - у нас є моделі, і по них фреймворк буде створювати таблиці в базі даних.

Нарешті, вкажемо рядок підключення. Відкриємо файл web.config, знайдемо секцію configSections і після неї вставимо секцію connectionStrings:

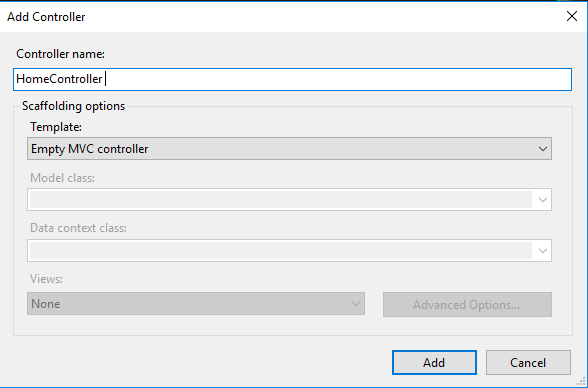
<connectionStrings>

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

</connectionStrings>

Тим самим ми визначаємо шлях до бази даних, яка потім буде створюватися. Вираз | DataDirectory | представляє заступник, який вказує, що база даних буде створюватися в проекті в папці App\_Data.

**Створення контролера і представлень.** Оскільки з моделями ми завершили роботу, перейдемо до найбільш важливої частини додатки - до контролера. Для контролерів в проекті вже є папка *Controllers*. Натисніть на неї правою кнопкою миші і в меню виберіть *Add-> Controller* .... З'явиться діалогове вікно створення нового контролера (рис. 1.7). Встановіть в якості імені контролера HomeController і натисніть *Add* (*Додати*)



**Рис. 1.4 - Діалогове вікно створення нового контролера**

У вас буде створений наступний контролер:

public class HomeController : Controller

{

// GET: /Home/

public ActionResult Index()

{

return View();

}

}

Контролер по суті і є головна ланка додатка, яка пов'язує модель і користувальницький інтерфейс. Зверніть увагу, що контролер - це звичайний клас, який наслідується від базового класу Controller. Він поки має єдиний метод Index, який повертає деякий результат методу View () - майбутнє представлення. І якщо ми зараз запустимо додаток, то отримаємо помилку про те, що представлення для даної дії не визначене. Але перед тим як додати представлення і приступити до створення користувальницького інтерфейсу змінимо метод Index наступним чином:

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 об'єкти в динамічну властивість Books в ViewBag

ViewBag.Books = books;

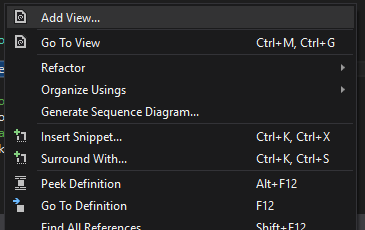
// Повертаємо представлення

return View();

}

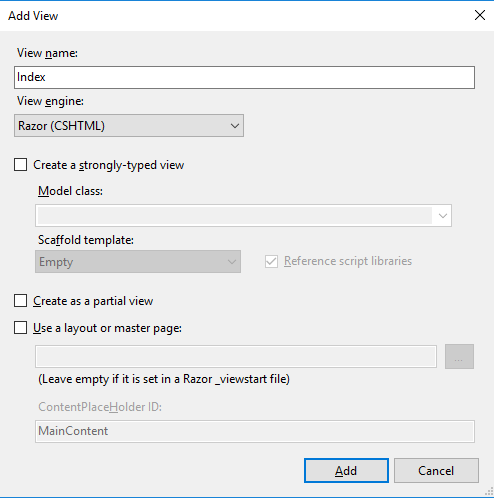
}

Насамперед, зверніть увагу, що, так як у нас моделі знаходяться в іншому просторі імен, хоча і в одному проекті, то ми його повинні імпортувати. Потім ми створюємо контекст даних. Використовуючи властивість Books, ми отримуємо з бази даних список об'єктів Book. Далі ми створюємо об'єкт Books в об'єкті ViewBag і присвоюємо йому цей список. Об'єкт ViewBag є таким об'єктом, який передається в представлення. І ми можемо визначити в цьому об'єкті будь-яку змінну і передати їй деяке значення, а потім в представленні витягти це значення. А тепер створимо саме представлення. Наведемо курсор на метод Index і натиснемо правою кнопкою миші. Нам відобразиться меню (рис.1.8), в якому виберемо *Add View* ....



**Рис. 1.5 - Меню *Add View***

Відкриється вікно додавання представлення, де буде запропоновано вибрати ім'я представлення і ще ряд параметрів. Оскільки у нас ще немає майстер-сторінок в проекті, знімемо галочку з прапорця Use a layout or masterpage: і натиснемо *Add* (рис.1.9).



**Рис. 1.6 - Вікно додавання представлення**

Visual Studio автоматично відкриє створене представлення Index.cshtml. По суті воно порожнє і містить мінімальний код розмітки на мові html. Змінимо його таким чином:

@{

Layout = null;

}

<!DOCTYPE html>

<html>

<head>

<meta name="viewport" content="width=device-width" />

<title>Бібліотека</title>

</head>

<body>

<div>

<h3> Бібліотека</h3>

<table>

<tr>

<td><p>Назва книги</p></td>

<td><p>Автор</p></td>

<td><p> Кількість днів оренди </p></td>

<td></td>

</tr>

@foreach (var b in ViewBag.Books)

{

<tr>

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

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

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

<td><p>

<a href="/Home/Rent/@b.Id">Орендувати</a></p></td>

</tr>

}

</table>

</div>

</body>

</html>

Тут ми створюємо таблицю, в якій будуть розташовуватися наші дані про книги. Особливий інтерес представляє конструкція *foreach (var b in ViewBag.Books)*. Ця конструкція використовує синтаксис движка представлення Razor (а саме його ми вибрали при створенні представлення). Після символу @ згідно синтаксису ми можемо використовувати вирази коду мовою C #. У циклі отримуємо властивості кожного елемента і поміщаємо її значення в комірку таблиці. Посилання в кінці - <a href="/Home/Buy/@b.Id">Купити</a> означатиме адресу, за якою буде розміщуватися форма оформлення покупки обраної книги.

**Основи маршрутизації.** Щоб викликати контролер HomeController або відправити йому запит, нам треба вказати в рядку запиту його ім'я - Home. Крім того, після імені контролера нам треба через слеш вказати дію або метод контролера, до якого відправляється запит. За замовчуванням при запуску проекту або при зверненні до сайту система MVC буде викликати дію Index контролера HomeController, якщо ми не вкажемо інший маршрут за замовчуванням в параметрах маршрутизації. Шлях /Home/Buy/@b.Id означає, що ми будемо звертатися до методу Buy контролера HomeController. Але перед тим як створити цей метод, наповнимо додаток даними.

**Дані для моделей за замовчуванням.** Ми можемо вже зараз запустити наш додаток, однак крім рядка із заголовками в таблиці ми нічого не побачимо, тому що у нас немає даних. Дану задачу можливо вирішити двома способами: додати в базу даних MSSQL деякі дані, або ініціалізувати початкові дані для моделей в коді. Виберемо другий варіант. Для цього в папку Models додамо новий клас BookDbInitializer і додамо в нього наступний код:

using System.Data.Entity;

namespace BookStore.Models

{

public class BookDbInitializer:DropCreateDatabaseAlways<BookContext>

{

protected override void Seed(BookContext db)

{

db.Books.Add(new Book { Name = " І спалахне полум'я ", Author = " С. Коллінз ", Quantity = 10 });

db.Books.Add(new Book { Name = " Дівчина в поїзді ", Author = " П. Хокінс ", Quantity = 15 });

db.Books.Add(new Book { Name = " Марсіянин ", Author = " Е. Вір ", Quantity = 20 });

base.Seed(db);

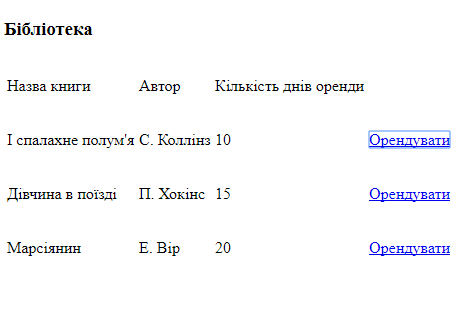
} }

}

За допомогою даного класу ми створюємо три об'єкти Book і додаємо їх в базу даних. Щоб цей клас заробив, нам треба не тільки його оголосити, але і запустити. Всі початкові налаштування і конфігурації у нас зберігаються у файлі Global.asax. Відкриємо його і додамо в нього в метод Application\_Start, який відпрацьовує при старті програми, наступний рядок:

Database.SetInitializer(new BookDbInitializer());

Все, тепер, якщо ми запустимо додаток, то побачимо, що в таблиці містяться раніше визначені нами дані (рис.1.10).



**Рис. 1.7 – Результат роботи додатка**

І якщо ми відкриємо папку проекту і зайдемо в ній в каталог App\_Data, то знайдемо щойно створену базу даних Bookstore.mdf, яка і буде зберігати таблиці з даними.

Зараз нам треба створити метод Buy, який буде відповідати за обробку введення користувача при оформленні покупки. Додамо в контролер HomeController наступні два методи:

[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 те що відвідали нашу бібліотеку!";

}

По суті тут ми визначили одну дію *Buy*, проте в одному випадку вона виконується при отриманні запиту *GET*, а в другому випадку - при отриманні запиту *POST*, що ми і визначили за допомогою атрибутів [HttpGet] і [HttpPost]. Метод *public ActionResult Rent (int id)* бере *id* обраної книги і повертає відповідне представлення. Метод *public string Rent (Purchase purchase)* приймає передану йому в запиті POST модель purchase і додає її в базу даних. В кінці повертаємо рядок повідомлення. Додамо представлення *Buy*. Для цього натиснемо на метод *public ActionResult Rent (int id)* правою кнопкою і додамо в проект нове представлення. Змінимо код нового представлення наступним чином:

@{

Layout = null;

}

<!DOCTYPE html>

<html>

<head>

<meta name="viewport" content="width=device-width" />

<title>Оренда</title>

</head>

<body>

<div>

<h3>Форма оформлення оренди</h3>

<form method="post">

<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>

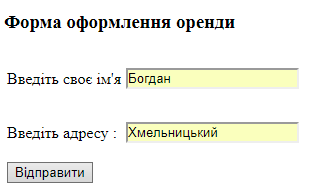
</div>

</body>

</html>

Тут ми створили форму введення даних. При переході по посиланню "/ Home / Rent / 2" контролер буде отримувати запит до дії Buy. І так як такий запит являє тип GET, користувачеві буде повертатися форма. Після заповнення форми і натиснення на кнопку форма буде оброблятися запитом POST, так як ми його визначили в рядку <form method="post">. Контролер знову буде отримувати запит до методу Rent, тільки тепер буде вибиратися для обробки запиту метод *public string Buy (Purchase purchase)*.

Зверніть увагу на поля введення *<input type = "text" name = "Person"/>* і <input type = "text" name = "Address" />. Їх властивості name відповідають іменам властивостей моделі Purchase. При натисканні кнопки і відправлення запиту передаються значення цих полів. Система MVC, використовуючи угоди за замовчуванням, вважає ці значення значеннями відповідних властивостей моделі. можемо його запустити. Перейдіть на сторінку оформлення замовлення, заповніть поля та натисніть кнопку "Відправити" (рис.1.11).





**Рис. 1.8- Форма оформлення покупки**

Стилізація додатка. Тепер в додатку не вистачає однієї деталі - стилізації. Зробимо якусь простеньку стилізацію. Для цього нам треба визначити файл стилів для нашого застосування. Всі файли стилів прийнято зберігати в папці Content, тому спочатку додамо в проект папку Content. Для цього натиснемо на назві проекту правою кнопкою миші і в меню виберемо *Add* (*Додати*) -> *New Folder* (*Нова папка*).

Тепер в нову папку додамо файл стилів. Назвемо його *Site.css*. Для цього натиснемо правою кнопкою миші на папку Content і в меню виберемо *Add* (*Додати*) -> *New Item* (*Новий елемент*). Потім в списку шаблонів знайдемо шаблон Style Sheet і дамо новому файлу ім'я Site.css. Тепер додамо в файл Site.css такий зміст:

body {

font-size: 13px;

font-family: 'Open Sans', Arial, Helvetica, Sans-Serif;

background-color: #f7f7fa;

padding-left:40px;

}

a{

display: inline-block;

text-decoration: none;

background-color: #f0f0f2;

color: #333;

border-radius: 10px;

padding: 10px 15px;

}

nav{

display: block;

}

h3 {

font-size:22px;

}

.menu {

padding-left:0px;

}

.menu ul {

list-style:none;

}

.menu li {

display:inline;

}

.menu a:hover {

color:red;

}

table {

vertical-align:middle;

text-align:left;

vertical-align: middle;

text-align: left;

background-color: #d4d9ff;

border-radius: 10px;

padding: 20px 30px;

}

.header {

font-weight:bold;

}

td {

padding-right:10px;

}

input {

width: 150px;

}

Клас .menu в даному випадку буде служити в якості класу для навігаційного меню на сайті. Тепер ми можемо використовувати стилі. Оскільки наші представлення по суті звичайні веб-сторінки, ми можемо як і на будь-якій сторінці html додати в секції head посилання на файл стилів:

<head>

<meta name="viewport" content="width=device-width" />

<link type="text/css" rel="stylesheet" href="../../Content/Site.css" />

</head>

Якщо в додатку одне представлення або якщо різні представлення використовують різні стилі та інші ресурси, то такий підхід виправданий. Однак у нашому випадку у нас є два представлення, у яких має бути одне меню і які повинні бути однаково стилізовані. У цьому випадку дуже зручно використовувати майстер-сторінки. Майстер-сторінка задає єдиний шаблон для інших представлень. Отже, створимо майстер-сторінку. По суті це просте представлення. Спочатку додамо в папку Views проекту нову папку Shared. Після цього в папку Shared додамо нове представлення. Назвемо його \_Layout.cshtml. Додамо в файл \_Layout.cshtml наступний текст:

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8" />

<title>@ViewBag.Title</title>

<link href="@Url.Content("~/Content/Site.css")" rel="stylesheet" type="text/css" />

</head>

<body>

<nav>

<ul class="menu">

<li>@Html.ActionLink("Головна", "Index", "Home")</li>

</ul>

</nav>

@RenderBody()

</body>

</html>

Тут ми використовували інший спосіб підключення стилів - ми використовували метод Url.Content, в який передали шлях до файлу. Обидва способи - і цей, і використаний раніше рівноправні. Потім ми створили меню сайту за допомогою методів Html.ActionLink, який генерує елемент-посилання і приймає назву посилання, метод контролера і ім'я контролера. Наприкінці йде метод RenderBody (), при виклику ресурсу в цю частину майстер-сторінки і будуть підставлятися вміст інших представлень. Тепер адаптуємо представлення до використання їх майстер-сторінкою. Змінимо представлення Index.cshtml наступним чином:

@{

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

}

<div>

<h3>Розпродаж книг</h3>

<table>

<tr>

<td><p>Назва книги</p></td>

<td><p>Автор</p></td>

<td><p> Кількість днів оренди </p></td>

<td></td>

</tr>

@foreach (var b in ViewBag.Books)

{

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

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

<td><p><a href="/Home/Rent/@b.Id">Оренда</a></p></td>

</tr>

}

</table>

</div>

Також змінимо представлення Buy.cshtml:

@{

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

}

<div>

<h3>Форма оформлення покупки</h3>

<form method="post">

<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>

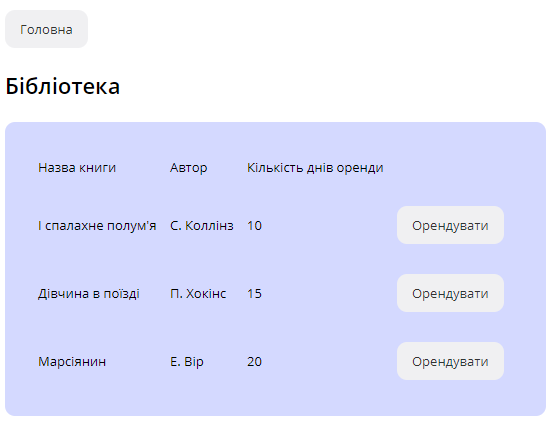
<tr><td><input type="submit" value="Відправити" /> </td><td></td></tr>

</table>

</form>

</div>

Сутність змін полягає в тому, що в секції *Layout = "~ / Views / Shared / \_Layout.cshtml";* ми вказуємо шлях до майстер-сторінки. Після цього нам не потрібні секції head і body. Запустивши додаток, ми побачимо, що тепер до нашого сайту застосована стилізація (рис.1.12).



**Рис. 1.9- Стилізація додатка**

**1.3. Контрольні питання**

1. ASP.NET MVC являє собою платформу для створення веб-додатків з використанням патерну (шаблону) MVC ( model - view - controller ).

2. Цей шаблон передбачає поділ системи на три взаємопов'язані частини: модель даних, вигляд (інтерфейс користувача) та модуль керування. Застосовується для відокремлення даних (моделі) від інтерфейсу користувача (вигляду) так, щоб зміни інтерфейсу користувача мінімально впливали на роботу з даними, а зміни в моделі даних могли здійснюватися без змін інтерфейсу користувача.

Мета шаблону — гнучкий дизайн програмного забезпечення, який повинен полегшувати подальші зміни чи розширення програм, а також надавати можливість повторного використання окремих компонентів програми. Крім того використання цього шаблону у великих системах сприяє впорядкованості їхньої структури і робить їх більш зрозумілими за рахунок зменшення складності.

3. Контролер ( Controller ) - обробляє команди користувача, визначає Моделі для роботи і пов'язує її з Представленням. Контролер (controller) представляє клас, з якого власне і починається робота програми. Цей клас забезпечує зв'язок між моделлю і представленням. Отримуючи дані що вводяться користувачем, контролер виходячи з внутрішньої логіки при необхідності звертається до моделі і генерує відповідне подання.

4. Представлення (View) - відповідає за користувальницький інтерфейс, відображає дані, отримані від Моделі. Представлення (view) - це власне візуальна частина або користувальницький інтерфейс програми - наприклад, html - сторінка, через яку користувач, що зайшов на сайт, взаємодіє з веб –додатком.

5. Модель ( Model ) - надає дані для Представлень у відповідь на запити Контролера, містить бізнес- логіку програми. Модель ( model ) являє набір класів, що описують логіку використовуваних даних .

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

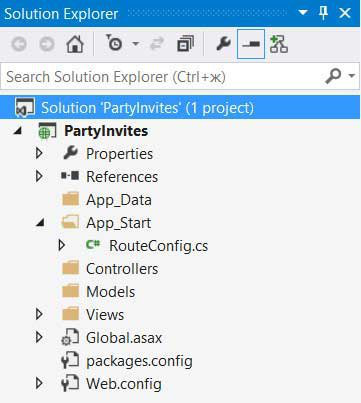
MVC Controller with Empty Read / Write Actions. Цей шаблон додає в проект контролер, який містить методи Index, Details, Create, Edit і Delete. Однак ці методи практично ніякої корисної роботи не виконують, і необхідно самим створити для них код та подання для кожного з цих методів.

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 подання для цих дій і додає код для добування інформації з бази даних. Що дуже корисно, тому що нам не треба писати відповідний код, як в попередньому розділі.

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 подання для цих дій.

7. 

8. Даний фреймворк дозволяє абстрагуватися від структури конкретної бази даних і вести всі операції з даними через модель.

9. Встановити початковий стан об'єкта шляхом ініціалізації атрибутів об'єкта та визначити інваріант класу.

10. Рядок підключення (англ. Connection string) це рядок, яка містить інформацію, необхідну ODBC драйверу / OLE DB провайдера для підключення до джерела даних. Як джерело даних можуть виступати бази даних, файли з даними.

Рядок підключення складається з набору пар "ключове слово" - "значення", між якими ставлять крапку з комою (;). Кожне ключове слово пов'язане з відповідним йому значенням знаком рівності (Приклад: Key1 = Value1; Key2 = Value2; Key3 = Value3;).

11. ViewBag - це оболонка навколо ViewData, яка відкриває словник ViewData як об'єкт dynamic.

12. Маршрутизація ASP.NET дозволяє використовувати URL-адреси, що не співпадають з певними файлами на веб-вузлі. Оскільки URL-адреса не співпадає з файлом, можна використовувати в веб-додатку URL-адреси, що описують дії користувача, внаслідок чого вони більш зрозумілі для користувачів. В додатку ASP.NET, що не використовує маршрутизацію, входящий запит URL-адреса звичайно співпадає з фізичною файлом на диску, наприклад ASPX-файлу. Наприклад, запрос адреса http: //server/application/Products.aspx? Id = 4 сопоставляется файлу Products.aspx, містить вміст коду та розмітку для відображення результату запиту в веб-оглядачі. Веб-сторінка використовує значення в рядку запросу id = 4, щоб визначити, який тип вмісту слід відобразити, проте це значення насправді не буде мати сенс для користувача. При маршрутизації ASP.NET задаються шаблони URL-адрес, що містять розташування заповнювачів для значень, що використовуються при обробці URL-запитів. Під час виконання частини URL-адреси, наступні за іменем програми, розбиваються на окремі значення залежно від заданого шаблону URL-адреси. Наприклад, при запросі адреса http: // server / application / Продукти / шоу / напої маршрутний аналізатор може передавать значення Продукти, шоу та напої в обработчик для запроса. У випадку, якщо запит не контролюється механізмом маршрутизації URL-адрес, фрагмент / Продукти / шоу / напої інтерпретується як шлях до файлу в додатку. Можна також використовувати шаблоны URL-адресов для програмного створення URL-адресов, відповідних маршрутам. Це дозволяє централізувати логіку створення гіперсила в додатку ASP.NET.

13. За замовчуванням в папці Models визначена одна модель - AccountModel, яка представляє окремий обліковий запис.

14. Файл global.asax дозволяє записувати оброблювачі подій, реагуючи на глобальні події. Користувачі не можуть запрашивать файл global.asax напрямую. Файл global.asax виконує свій код автоматично у відповідь на деякі події програми.

15. Файл Site.css використовується для задання стилів. Даний файл створюється за замовчуванням. Підключається за допомогою строки <link type="text/css" rel="stylesheet" href="../../Content/Site.css" /> в секції head.

16. Майстер-сторінки використовуються для створення однакового, уніфікованого вигляду сайту. По суті майстер-сторінки - це ті ж самі уявлення, але дозволяють включати в себе інші уявлення. Наприклад, можна визначити на майстер-сторінці загальні для всіх інших уявлень елементи, а також підключити загальні стилі і скрипти. У підсумок нам не доведеться на кожному окремому поданні прописувати шлях до файлів стилів, а потім при необхідності його змінювати. А використовувані на майстер-сторінках наповнювачі або плейсхолдери дозволять вставити на їх місце інші уявлення.

За замовчуванням при створенні нового проекту ASP.NET MVC 5 в проект вже додається майстер-сторінка під назвою \_Layout.chtml, яку можна знайти в каталозі Views / Shared.

17. Метод RenderBody() викликає ресурс в частину майстра-сторінки, куди підставляється вміст інших представлень.

**Висновок:** На даній лабораторній роботія я створив примітивний додаток на MVC 4 і ознайомився з роботою ASP & MVC.