

УДК 004.4

ББК 32.973.26 – 018.2.75

Р-58

*Рекомендовано Вченою радою факультету математики та інформатики
ДВНЗ «Прикарпатський національний університет імені Василя Стефаника»
як навчальний посібник для студентів математичних та технічних напрямів
підготовки (протокол № 4 від 28 листопада 2016р.).*

Рецензенти :

Мельничук С. І., доктор технічних наук, професор (Івано-Франківський національний університет нафти і газу)

Олійник А. П., доктор технічних наук, професор (Івано-Франківський національний університет нафти і газу)

Р-58 Навчальний посібник з курсу «Програмування мовою С#» /
В.А. Ровінський – Івано-Франківськ : 2016, Сімик. – 603с.

Посібник призначений для вивчення мови програмування С# на основі вже опанованої мови С++. Основна увага приділяється спеціальним аспектам застосування мови С# для випадків побудови Web-орієнтованого програмного забезпечення. Розглянуті: бібліотека MVC5, оптимальні способи взаємодії з базами даних, способи генерування Web-сторінок за допомогою бібліотеки Bootstrap, способи аутентифікації користувача за допомогою соціальних мереж, та способи автоматизації передачі даних з використанням різних комунікаційних каналів. Розгляд матеріалу здійснюється на основі розгляду практичних прикладів.

Для студентів технічних та математичних спеціальностей, які вивчають курси «Програмування. С#»

УДК 004.4

ББК 32.973.26 – 018.2.75

© В.А.Ровінський , 2016

Міністерство освіти і науки України
Прикарпатський національний університет ім. Василя Стефаника

В.А. Ровінський

Програмування мовою C#

навчальний посібник

Івано-Франківськ

2016

Передмова

Навчальний посібник написано на основі досвіду викладання курсу «Програмування мовою C#» на факультеті математики та інформатики Прикарпатського національного університету імені Василя Стефаника для студентів спеціальності «Інформатика».

В посібнику розглядаються основні елементи мови C#, які відрізняються від аналогічних засобів мов C/C++, тому такий посібник може бути ефективно використаний тільки після вивчення курсу «Програмування мовою C/C++». Розглядаються відносно нові елементи мови, такі як: делегати, лямбда-вирази, інтерфейси, регулярні вирази, індексатори та їх використання. Розглянуті також особливості об'єктно-орієнтованого програмування з використанням мови C#, такі як: запечатані класи, особливості успадкування класів, розподілені класи, тощо.

Основна увага зосереджена на способах побудови Web-орієнтованого програмного забезпечення з використанням мови C# та фреймворку MVC5. Розглянуті наступні практичні аспекти побудови такого програмного забезпечення: створення контролерів і представлень, моделі та їх функціонування у взаємодії з базами даних, моделі із багатокomпонентними зв'язками, питання маршрутизації, особливості використання метаданих та валідації моделей, фільтрація даних, використання JavaScript та AJAX, авторизація та аутентифікація в MVC 5, використання JQuery UI, Bootstrap та візуальних компонентів, кешування даних, як засіб прискорення функціонування програмного забезпечення.

Розглянуті також деякі питання технології розробки програмного забезпечення: засіб рефакторингу Resharper - його можливості, потреба та особливості його використання. Достатня увага приділена питанням основ автоматизованого тестування програмного забезпечення, як розвитку підходу Test Driven Development.

Доступний стиль викладу, орієнтований на розгляді прикладів дозволяє використовувати навчальний посібник самостійно при підготовці до екзаменів, та при самостійному вивченні курсу.

1	Основні конструкції C#.....	9
1.1	Принципи функціонування.....	9
1.1.1	Збирач сміття.....	9
1.1.2	Події.....	10
	Контрольні питання.....	10
1.2	Оператори та ключові слова мови.....	10
1.2.1	Ключові слова checked/unchecked.....	10
1.2.2	Ключове слово default.....	11
1.2.3	Ключове слово IS.....	13
1.2.4	Оператор ??.....	14
1.2.5	Оператор AS.....	14
1.2.6	Ключове слово base.....	15
1.2.7	Ключове слово virtual.....	15
1.2.8	Ключове слово params.....	15
1.2.9	Ключове слово ref.....	16
1.2.10	Ключове слово out.....	17
1.2.11	Оператор foreach.....	19
	Контрольні питання.....	20
1.3	Масиви.....	20
1.3.1	Одновимірні масиви.....	21
1.3.2	Багатовимірні масиви.....	22
1.3.3	Масиви масивів.....	23
1.3.4	Передача масивів за допомогою параметрів ref і out.....	26
	Контрольні питання.....	27
1.4	Класи, інтерфейси та ООП.....	28
1.4.1	Створення об'єктів.....	28
1.4.2	Наслідування (успадкування) класів.....	29
1.4.3	Модифікатор abstract.....	30
1.4.4	Метод override.....	32
1.4.5	Ключове слово virtual.....	34
1.4.6	Поліформізм.....	37
1.4.7	Віртуальні члени.....	40
1.4.8	Ключове слово sealed.....	42
1.4.9	Деструктори.....	45
1.4.10	Інтерфейси (interface).....	47
1.4.11	Ключове слово SET.....	49
1.4.12	Ключове слово GET.....	50
1.4.13	Асиметричні методи доступу.....	50
1.4.14	Індексатори.....	53
1.4.15	Розділювані типи.....	56
1.4.16	Розділювані методи.....	56
	Контрольні питання.....	57
1.5	Делегати.....	57
1.5.1	Основні принципи функціонування делегатів.....	57

1.5.2 Використання варіативності в делегатах.....	61
Контрольні питання.....	62
1.6 Лямбда-вирази.....	63
1.6.1 Основні положення.....	63
1.6.2 Лямбда-вирази.....	64
1.6.3 Лямбда-оператори.....	64
1.6.4 Лямбди зі стандартними операторами запитів.....	65
1.6.5 Вивід типу в лямбда-виразах.....	66
1.6.6 Область дії змінної в лямбда-виразах.....	66
Контрольні питання.....	68
1.7 Регулярні вирази.....	68
1.7.1 Елементи регулярних виразів.....	68
1.7.2 Використання регулярних виразів в C#.....	72
Контрольні питання.....	75
2 Візуальні програми.....	76
Контрольні питання.....	80
3 ASP.NET MVC.....	81
3.1 Початок роботи.....	81
3.1.1 Структура проекту MVC 5.....	84
Контрольні питання.....	85
3.2 Створення першої інтернет-програми на ASP.NET MVC 5.....	86
3.2.1 EntityFramework.....	88
3.2.2 CodeFirst.....	89
Контрольні питання.....	90
3.3 Створення контролера і представлень.....	90
3.3.1 Стилзація програми і майстер сторінки.....	93
3.3.2 Контролери.....	98
3.3.3 Методи дій та їх параметри.....	100
3.3.4 Передача даних в контролери і параметри.....	101
3.3.5 Отримання даних з контексту запиту.....	103
3.3.6 Результати дій.....	103
3.3.7 Вбудовані класи, похідні від ActionResult.....	105
3.3.8 ViewResult і генерація представлень.....	106
3.3.9 Передача даних з контролера в представлення.....	107
3.3.10 Переадресація і відправка кодів статусу і помилок.....	109
3.3.11 Відправка помилок і статусних кодів.....	110
3.3.12 Відправка файлів в ASP.NET MVC 5.....	111
3.3.13 Контекст запиту HttpContext. Cookie. Сесії.....	112
3.3.14 Відправка відповіді.....	113
3.3.15 Визначення користувача.....	114
3.3.16 Робота з Cookie.....	114
3.3.17 Сесії.....	115
3.3.18 Асинхронні методи.....	115
3.4 Представлення (View).....	118
3.4.1 Основні положення.....	118
3.4.2 Створення нового представлення.....	121

3.4.3	Шляхи до файлів представлень	123
3.4.4	Синтаксис Razor	123
3.4.5	Бібліотека представлень	123
3.4.6	Основи синтаксису Razor	124
3.4.7	Строго типізовані представлення	124
3.4.8	Майстер-сторінки	127
3.4.9	ViewStart	130
3.4.10	Часткові представлення	130
3.4.11	HTML-хелпери	132
3.4.12	Робота з формами	136
3.4.13	Строго типізовані хелпери	142
	Контрольні питання	144
3.5	Моделі	145
3.5.1	Підключення до бази даних	146
3.5.2	Шаблонні хелпери	153
3.5.3	Редагування моделі	154
3.5.4	Додавання та видалення моделі	155
3.5.5	Шаблони формування	159
3.5.6	Моделі зі складною структурою	163
3.5.7	Операції для роботи зі складними моделями	168
3.5.8	Моделі зі зв'язками «багато-до-багатьох»	172
3.5.9	Передача масивів і складних даних в контролер	178
3.5.10	Міграція баз даних	182
3.5.11	Створення блочних сторінок	186
3.5.12	Перевизначення шаблонів формування	189
3.5.13	Перевизначення шаблонів відображення і редагування	195
3.5.14	Фільтрація даних	199
3.5.15	Domain Model і View Model	202
	Контрольні питання	204
3.6	Маршрутизація	205
3.6.1	Визначення маршрутів	205
3.6.2	Робота з маршрутами	207
3.6.3	Зіставлення запитів з файлами на диску	209
3.6.4	Використання префіксів в рядку запиту	209
3.6.5	Порядок визначення нових маршрутів	210
3.6.6	Отримання переданих параметрів	211
3.6.7	Передача довільної кількості параметрів в запиті	211
3.6.8	Створення обмежень для маршрутів	212
3.6.9	Створення власних обмежень	213
3.6.10	Ігнорування запитів	214
3.6.11	Генерація вихідних адрес URL	215
3.6.12	Області	217
3.6.13	Генерація посилань в областях	220
3.6.14	Створення власного обробника маршрутів	221
3.6.15	Атрибути маршрутизації	222
3.6.16	Обмеження маршрутів	224

3.6.17	Значення за замовчуванням	224
3.6.18	Використання префіксів	225
3.6.19	Маршрутизація та вкладені ресурси	225
	Контрольні питання.	228
3.7	Метадані та валідація моделі	228
3.7.1	Анотації даних для відображення властивостей	228
3.7.2	Основи валідації	232
3.7.3	Атрибути валідації	234
3.7.4	Валідація моделі в контролері	237
3.7.5	Відображення помилок валідації.....	239
3.7.6	Створення власної логіки валідації	242
	Контрольні питання.	248
3.8	Фільтри	248
3.8.1	Фільтри аутентифікації.....	250
3.8.2	Фільтри авторизації	252
3.8.3	Фільтри виключень	256
3.8.4	Фільтри дій і результатів.....	259
3.8.5	Приклади фільтрів дій	262
3.8.6	Глобальні фільтри і перевизначення фільтрів	267
3.8.7	Додаткові вбудовані фільтри	269
	Контрольні питання.	270
3.9	Прив'язка моделі.....	270
3.9.1	Введення в прив'язку моделей	270
3.9.2	Вибіркова прив'язка	273
3.9.3	Явна прив'язка моделі.....	274
3.9.4	Постачальники значень	275
3.9.5	Створення прив'язки моделі	277
	Контрольні питання.	278
3.10	JavaScript і AJAX.....	279
3.10.1	Використання JavaScript.....	279
3.10.2	Вступ в AJAX	281
3.10.3	Ajax-форми	283
3.10.4	Параметри об'єкту AjaxOptions	285
3.10.5	AJAX-посилання	288
3.10.6	Формат JSON	289
3.10.7	AJAX-запити за допомогою jQuery.....	290
3.10.8	Приклад. Створення чату на AJAX і jQuery.....	291
	Контрольні питання.	300
3.11	OWIN і Katana	300
3.11.1	OWIN і Katana в проектах MVC 5.....	305
	Контрольні питання.	306
3.12	Авторизація та аутентифікація в MVC 5	307
3.12.1	ASP.NET Identity	307
3.12.2	Базові класи AspNet Identity.....	313
3.12.3	Авторизація через зовнішні сервіси.....	317
3.12.4	Перевизначення системи AspNet Identity	326

3.12.5	Робота з ролями в AspNet Identity	331
3.12.6	Аутентифікація форм.....	334
3.12.7	Використання об'єктів Claim	341
3.12.8	Налаштування валідації пароля в ASP.NET Identity	344
3.12.9	Валідація користувача в ASP.NET Identity.....	346
3.12.10	Підтвердження реєстрації по Email.....	348
3.12.11	Підтвердження Email в ASP.NET Identity 2.0	352
3.12.12	Підтвердження телефону по SMS в ASP.NET Identity 2.0....	355
3.12.13	Скидання пароля в ASP.NET Identity 2.0.....	362
3.12.14	Універсальні провайдери в MVC 5	365
3.12.15	Перевизначення універсальних провайдерів. Провайдер ролей 372	372
3.12.16	Створення програми з ASP.NET Identity «з нуля»	379
3.12.17	Реєстрація та створення користувачів в ASP.NET Identity...	382
3.12.18	Авторизація користувачів в ASP.NET Identity.....	385
3.12.19	Редагування і видалення користувачів	387
3.12.20	Додавання ролей в ASP.NET Identity.....	390
3.12.21	Аутентифікація OWIN і ClaimsIdentity.....	395
3.12.22	Авторизація на основі Claims	401
	Контрольні питання.	405
3.13	Бандли і мініфікація.....	405
3.13.1	Вступ до бандлів і мініфікацій	405
3.13.2	Створення бандлів. Бібліотеки з CDN	411
	Контрольні питання.	413
3.14	Bootstrap в ASP.NET MVC 5	414
3.14.1	Вступ до Bootstrap-у	414
3.14.2	Позиціювання елементів	417
3.14.3	Компоненти Bootstrap.....	420
3.14.4	Робота з формами в Bootstrap	425
3.14.5	Робота з таблицями.....	428
3.14.6	Модальні вікна	430
	Контрольні питання.	432
3.15	SignalR 2.....	433
3.15.1	Вступ до SignalR 2	433
3.15.2	Перша програма з SignalR 2.....	434
3.15.3	Persistent Connection API	441
3.15.4	Малювання на SignalR.....	446
3.15.5	Push-повідомлення	451
	Контрольні питання.	454
3.16	Робота з jQuery UI і візуальними елементами	454
3.16.1	jQuery UI і Autocomplete.....	454
3.16.2	Діалогові вікна і CRUD-інтерфейс.....	458
3.16.3	Валідація в діалогових вікнах.....	463
3.16.4	JqGrid в ASP.NET MVC.....	467
3.16.5	Посторінковий вивід в JqGrid.....	470
3.16.6	Пошук по таблиці в JqGrid.....	471

3.16.7	CRUD-операції в JqGrid	473
3.16.8	Посторінковий вивід	477
	Контрольні питання.	479
4	Засоби рефакторингу та інші технологічні моменти.	479
4.1	ReSharper	479
	Контрольні питання.	482
4.2	Тестування програмного забезпечення.....	482
4.2.1	Вступ в тестування. Test-Driven Development:	482
4.2.2	Створення проекту для юніт-тестів.....	484
4.2.3	Створення юніт-тестів	489
4.2.4	Слабо пов'язані об'єкти і тестування роботи з БД	493
4.2.5	Фреймворк Moq.....	497
4.2.6	Тестування створення моделі і переадресації.....	499
4.2.7	Управління життєвим циклом додатка	502
4.2.8	Модулі	507
4.2.9	Створення власного модуля.....	508
4.2.10	Взаємодія між модулями. Події модулів	510
4.2.11	HTTP-обробники	512
4.2.12	Перевизначення HTTP-обробника	516
4.2.13	Виконання дій по таймеру	517
	Контрольні питання.	518
4.3	Кешування в ASP.NET MVC	519
4.3.1	Атрибут OutputCache	519
4.3.2	Параметри кешування	521
4.3.3	Профілі кешування	523
4.3.4	Динамічне управління кешем	524
4.3.5	Кешування статичного контенту	526
4.3.6	MemoryCache	529
	Контрольні питання.	531
4.4	Впровадження залежностей	531
4.4.1	Інверсія управління:.....	531
4.4.2	IoC-контейнер Ninject.....	534
4.4.3	Інтерфейс IDependencyResolver.....	535
4.4.4	Впровадження залежностей з параметрами	538
4.4.5	IoC-контейнер Autofac.....	539
4.4.6	IoC-контейнер Castle Windsor.....	542
	Контрольні питання.	547
4.5	URL Rewriting.....	547
4.5.1	Вступ в URL Rewriting	547
4.5.2	Змінні сервера і зворотні посилання	551
4.5.3	Приклади правил URL Rewriting.....	553
	Контрольні питання.	555
4.6	Архітектура програмного забезпечення	555
4.6.1	Onion-архітектура.	555
4.6.2	Патерн Unit of Work.....	567
4.6.3	AutoMapper	572

4.6.4 Багаторівнева архітектура.....	577
4.6.5 Монолітна архітектура	579
4.6.6 Data Access Layer.....	583
4.6.7 Business Logic Layer.....	589
4.6.8 Presentation Layer.....	596
Контрольні питання.	601

Передмова

Навчальний посібник написано на основі досвіду викладання курсу «Програмування мовою C#» на факультеті математики та інформатики Прикарпатського національного університету імені Василя Стефаника для студентів спеціальності «Інформатика».

В посібнику розглядаються основні елементи мови C#, які відрізняються від аналогічних засобів мов C/C++, тому такий посібник може бути ефективно використаний тільки після вивчення курсу «Програмування мовою C/C++». Розглядаються відносно нові елементи мови, такі як: делегати, лямбда-вирази, інтерфейси, регулярні вирази, індексатори та їх використання. Розглянуті також особливості об'єктно-орієнтованого програмування з використанням мови C#, такі як: запечатані класи, особливості успадкування класів, розподілені класи, тощо.

Основна увага зосереджена на способах побудови Web-орієнтованого програмного забезпечення з використанням мови C# та фреймворку MVC5. Розглянуті наступні практичні аспекти побудови такого програмного забезпечення: створення контролерів і представлень, моделі та їх функціонування у взаємодії з базами даних, моделі із багатокomпонентними зв'язками, питання маршрутизації, особливості використання метаданих та валідації моделей, фільтрація даних, використання JavaScript та AJAX, авторизація та аутентифікація в MVC 5, використання JQuery UI, Bootstrap та візуальних компонентів, кешування даних, як засіб прискорення функціонування програмного забезпечення.

Розглянуті також деякі питання технології розробки програмного забезпечення: засіб рефакторингу Resharper - його можливості, потреба та особливості його використання. Достатня увага приділена питанням основ автоматизованого тестування програмного забезпечення, як розвитку підходу Test Driven Development.

Доступний стиль викладу, орієнтований на розгляді прикладів дозволяє використовувати навчальний посібник самостійно при підготовці до екзаменів, та при самостійному вивченні курсу.

1 Основні конструкції C#

1.1 Принципи функціонування.

Мова програмування C# є одним з продовжень мови C та C++ і розроблена на основі аналізу основних помилок, які найчастіше зустрічаються при розробці програмного забезпечення. Зроблена спроба за рахунок синтаксичних обмежень, та введення нових мовних конструкцій створити мову, яка буде зручною та зрозумілою при її використанні в широкомасштабних командних програмних проектах. Використання платформи .NET забезпечує певну можливість запуску програм, створених на C# на інших операційних системах.

.NET — програмна технологія, розроблена фірмою Microsoft як платформа для створення як звичайних, так і веб-програм. Багато в чому є продовженням ідей та принципів, покладених в технологію Java.

Одною з ідей .NET є сумісність служб, написаних різними мовами. Кожна бібліотека (збірка) в .NET має відомості про свою версію, що дозволяє усунути можливі конфлікти між різними версіями збірок.

.NET — крос-платформена технологія, в цей час існує реалізація для платформи Windows, FreeBSD і варіант технології для ОС Linux в проекті Mono, DotGNU.

.NET поділяється на дві основні частини— середовище виконання (по суті віртуальна машина) та інструментарій розробки.

Середовища розробки .NET-програм: Visual Studio.NET (C++, C#), SharpDevelop, Borland Developer Studio (Delphi, C#) і т. д. Такі програми також можна розроблювати в текстовому редакторі та використовувати консольний компілятор.

Як і технологія Java, середовище розробки .NET створює байт-код, призначений для виконання віртуальною машиною. Вхідна мова цієї машини в .NET називається CIL (Common Intermediate Language), також відома як MSIL (Microsoft Intermediate Language), або просто IL. Застосування байт-кода дозволяє отримати кросплатформеність на рівні скомпільованого проекту (в термінах .NET: *збірка*), а не на рівні початкового тексту, як, наприклад, в C. Перед запуском збірки в середовищі виконання (CLR) байт-код перетворюється вбудованим в середовище JIT-компілятором (just in time, компіляція на льоту) в машинні коди цільового процесора. Такий підхід має зменшену продуктивність на першому проході виконання програми. При продовженні виконання (коли однаковий фрагмент програми виконується багаторазово), швидкість виконання стає співрозмірним із швидкістю програм, написаних на C/C++.

1.1.1 Збирач сміття

Збирач сміття .NET Framework управляє виділенням і звільненням пам'яті для програми. При кожному створенні нового об'єкта середовище CLR виділяє пам'ять для об'єкту з керованої динамічно розподіленої пам'яті (купи). Поки в керованій купі є доступний адресний простір, середовище виконання продовжує виділяти простір для нових об'єктів. Але пам'ять має межі. У кінцевому рахунку, щоб звільнити деяку кількість пам'яті, збирач сміття повинен виконати

процедуру очищення. Механізм оптимізації збирача сміття визначає найкращий час для виконання збору, ґрунтуючись на аналізі процесу виділення пам'яті. У ході виконання очищення, збирач сміття відшукує в керованій купі об'єкти, які більш не використовуються додатком, та звільняє виділену для них пам'ять.

1.1.2 Події

Події дозволяють класу або об'єкту повідомляти інші класи чи об'єкти про виникнення будь-яких ситуацій. Клас, що відправляє (або викликає) подію, називається видавцем, а класи, приймаючі (або обробляючі) події, називаються передплатниками.

У C# в стандартному додатку Windows Forms або веб-додатку користувач підписується на події, що викликаються елементами управління, такими як кнопки і поля зі списками. Для перегляду подій, що публікуються елементом управління, і вибору деяких з них для обробки можна скористатися середовищем IDE (інтегроване середовище розробки) Visual C#. IDE автоматично додасть порожній метод обробника подій і код, необхідний для підписки на подію.

Контрольні питання.

1. Для чого використовується збирач сміття?
2. Які переваги і недоліки використання байт-кодової моделі виконання програми?

1.2 Оператори та ключові слова мови

Оператори мови C# в основному співпадають з операторами C++, тому розглянемо основні оператори мови, відмінні від мови C++.

1.2.1 Ключові слова **checked/unchecked**

Ключове слово **checked** використовується для явного включення перевірки переповнення при виконанні арифметичних операцій і перетворень з даними цілого типу.

Типово вираз, що містить тільки константні значення, викликає помилку компілятора в тому випадку, якщо результат його обчислення виходить за допустимі межі значень кінцевого типу. Якщо вираз містить одне або кілька неконстантних значень, компілятор не виконує перевірку переповнення. Обчислення виразу, присвоєного змінної `i2` в наступному прикладі, не викликає помилку компілятора:

```
int ten = 10;  
int i2 = 2147483647 + ten;
```

```
// By default, the overflow in the previous statement also does  
// not cause a run-time exception. The following line displays  
// -2,147,483,639 as the sum of 2,147,483,647 and 10.
```

```
Console.WriteLine(i2);
```

За замовчуванням ці неконстантні вирази також не перевіряються на переповнення під час виконання, і вони не створюють винятків переповнення. У попередньому прикладі як суми двох позитивних цілих чисел виводиться значення -2147483639.

Перевірку переповнення можна включити за допомогою параметрів компілятора, налаштування середовища або використання ключового слова **checked**. У наступних прикладах демонструється використання вираження **checked** або блоку **checked** для виявлення переповнення, що виникає в результаті попереднього додавання під час виконання програми. В обох прикладах створюється виняток переповнення:

```
// Checked expression.  
Console.WriteLine(checked(2147483647 + ten));  
  
// Checked block.  
checked  
{  
    int i3 = 2147483647 + ten;  
    Console.WriteLine(i3);  
}
```

Ключове слово **unchecked** використовується для усунення перевірки переповнення при виконанні арифметичних операцій і перетворень з даними цілого типу.

Якщо в контексті, що не перевіряється, результатом виконання виразу є значення, що виходить за допустимі межі значень кінцевого типу, то переповнення НЕ позначається. Обчислення у наведеному нижче прикладі виконується в блоці або вираженні **unchecked**, тому факт перевищення результатом максимального значення цілого числа ігнорується і змінної **int1** присвоюється значення -2147483639:

```
unchecked  
{  
    int1 = 2147483647 + 10;  
}  
int1 = unchecked(ConstantMax + 10);
```

1.2.2 Ключове слово default

В універсальних класах і методах однією з існуючих проблем є призначення параметризованому типу **T** значення за замовчуванням, якщо заздалегідь невідомі наступні моменти:

- Чи є **T** покажчиковим типом або типом значення.

- Чи є T типом значення, або чи буде він числовим значенням або структурою.

При заданій змінній t параметризованого типу T оператор `t = null` дійсний, якщо тільки T є покажчиковим типом, а `t = 0` буде справедливо тільки для числових типів значень, але не для структур. Рішенням є використання ключового слова `default`, яке поверне значення `null` для посилальних типів і `0` - для числових типів значень. Для структур воно поверне значення кожного члена зі значенням `0` або `null` залежно від їх типу. Для типів, що допускають значення `null`, ключове слово `default` поверне значення `System.Nullable <T>`, яке ініціалізується як будь-яка структура.

```
namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            // Test with a non-empty list of integers.
            GenericList<int> gll = new GenericList<int>();
            gll.AddNode(5);
            gll.AddNode(4);
            int intVal = gll.GetLast();
            // The following line displays 5.
            System.Console.WriteLine(intVal);

            // Test with a non-empty list of strings.
            GenericList<string> gll3 = new GenericList<string>();
            gll3.AddNode("five");
            gll3.AddNode("four");
            string sVal = gll3.GetLast();
            // The following line displays five.
            System.Console.WriteLine(sVal);
        }
    }

    // T is the type of data stored in a particular instance of GenericList.
    public class GenericList<T>
    {
        private class Node
        {
            // Each node has a reference to the next node in the list.
            public Node Next;
            // Each node holds a value of type T.
            public T Data;
        }

        // The list is initially empty.
        private Node head = null;
    }
}
```

```

// Add a node at the beginning of the list with t as its data value.
public void AddNode(T t)
{
    Node newNode = new Node();
    newNode.Next = head;
    newNode.Data = t;
    head = newNode;
}

// The following method returns the data value stored in the last // node in
// the list. If the list is empty, the default value for type T is
// returned.
public T GetLast()
{
    // The value of temp is returned as the value of the method.
    // The following declaration initializes temp to the appropriate
    // default value for type T. The default value is returned if the
    // list is empty.
    T temp = default(T); // - застосування default коли тип невідомий
    Node current = head;
    while (current != null)
    {
        temp = current.Data;
        current = current.Next;
    }
    return temp;
}
}

```

1.2.3 Ключове слово IS

Перевіряє сумісність об'єкта з заданим типом. Наприклад, в наступному коді визначається, чи є об'єкт екземпляром типу MyObject або типу, похідного від MyObject:

```

if (obj is MyObject)
{
}

```

Якщо заданий вираз відмінний від NULL і заданий об'єкт також може бути приведений до заданого типу не викликаючи винятків, вираз **is** приймає значення true.

Ключове слово **is** викликає попередження під час компіляції, якщо відомо, що вираз завжди буде мати значення true або false. Оператор **is** перевантажити не можна. Оператор **is** розглядає тільки перетворення показчиків, упаковки/перетворення і розпакування/перетворення. Інші перетворення не враховуються.

1.2.4 Оператор ??

Оператор ?? називається оператором null-об'єднання і використовується для вказівки значень за замовчуванням для типів значень і покажчикових типів, що допускають значення null. Повертає лівий операнд, якщо операнд НЕ дорівнює null; в іншому випадку - повертає правий операнд.

```
class NullCoalesce
{
    static int? GetNullableInt()
    {
        return null;
    }

    static string GetStringValue()
    {
        return null;
    }

    static void Main()
    {
        // ?? operator example.
        int? x = null;

        // y = x, unless x is null, in which case y = -1.
        int y = x ?? -1;

        // Assign i to return value of method, unless
        // return value is null, in which case assign
        // default value of int to i.
        int i = GetNullableInt() ?? default(int);

        string s = GetStringValue();
        // ?? also works with reference types.
        // Display contents of s, unless s is null,
        // in which case display "Unspecified".
        Console.WriteLine(s ?? "Unspecified");
    }
}
```

1.2.5 Оператор AS

Оператор **as** подібний оператору приведення типів. Однак якщо перетворення неможливо, то **as** повертає **null** замість виклику винятку. Наступний приклад:

expression as type

Цей код еквівалентний наступному виразу, за винятком того, що змінна expression обчислюється тільки один раз.

expression is type ? (type)expression : (type)null

Зверніть увагу, що оператор **as** виконує тільки перетворення покажчиків, подібні перетворення і перетворення упаковка-перетворення. Оператор **as** не може виконувати інші перетворення, наприклад задане користувачем перетворення.

1.2.6 Ключове слово **base**

Ключове слово **base** використовується для доступу до членів базового класу з похідного класу:

- Виклик методу базового класу, який був перевизначений іншим методом.
- Визначення конструктора базового класу, який повинен викликатися при створенні екземплярів похідного класу.

Детальніше про нього - в розділі **Поліморфізм**.

1.2.7 Ключове слово **virtual**

Ключове слово **virtual** використовується для зміни оголошень методів, властивостей, індексаторів і подій і забезпечує дозвіл їх перевизначення в похідному класі. Детальніше про нього - в розділі **Класи**.

1.2.8 Ключове слово **params**

Ключове слово **params** дозволяє визначити параметр методу, який приймає аргумент, в якому кількість аргументів є змінним. В оголошенні методу після ключового слова **params** додаткові параметри не допускаються, і в оголошенні методу допускається тільки одне ключове слово **params**.

```
public class MyClass
{
    public static void UseParams(params int[] list)
    {
        for (int i = 0 ; i < list.Length; i++)
        {
            Console.Write(list[i] + " ");
        }
        Console.WriteLine();
    }

    public static void UseParams2(params object[] list)
    {
        for (int i = 0 ; i < list.Length; i++)
        {
            Console.Write(list[i] + " ");
        }
        Console.WriteLine();
    }

    static void Main()
```

```

{
    UseParams(1, 2, 3);
    UseParams2(1, 'a', "test");

    // An array of objects can also be passed, as long as
    // the array type matches the method being called.
    int[] myarray = new int[3] {10,11,12};
    UseParams(myarray);
}
}
/*
Output:
1 2 3
1 a test
10 11 12
*/

```

1.2.9 Ключове слово ref

Ключове слово **ref** використовується для передачі аргументів за покажчиком. В результаті всі зміни параметра в методі будуть відображені у змінній при передачі елемента управління назад в викликаючий метод. Приклад:

```

class RefExample
{
    static void Method(ref int i)
    {
        i = 44;
    }
    static void Main()
    {
        int val = 0;
        Method(ref val);
        // val is now 44
    }
}

```

Аргумент, що передається в параметр **ref**, спочатку слід ініціалізувати. У цьому полягає відмінність від **out**, аргументи якого не вимагають явної ініціалізації перед передачею. Хоча **ref** і **out** по-різному обробляються під час виконання програми, в режимі компіляції вони управляються однаково. Отже, якщо один метод приймає аргумент **ref**, а інший - **out**, вони не можуть бути перевантажені. Ці два методи ідентичні, наприклад при компіляції, тому даний код скомпільовано не буде:

```

class CS0663_Example
{
    // Compiler error CS0663: "Cannot define overloaded
    // methods that differ only on ref and out".

```

```

    public void SampleMethod(out int i) { }
    public void SampleMethod(ref int i) { }
}

```

Однак перевантаження можливе, якщо один метод приймає аргумент **ref** або **out**, а інший не приймає жодного, як показано в наступному прикладі.

```

class RefOverloadExample
{
    public void SampleMethod(int i) { }
    public void SampleMethod(ref int i) { }
}

```

Властивості не є змінними і тому не можуть бути передані в якості параметрів **ref**.

Передача типів значень за посиланням, є корисною, однак для передачі покажчикових типів також буде доцільним використання **ref**. Це дозволяє викликати методи змінювати об'єкт, на який вказує посилання, оскільки сама посилання передається за посиланням. Наступний приклад показує, що при передачі покажчикowego типу як параметр **ref** може бути змінений сам об'єкт.

```

class RefExample2
{
    static void Method(ref string s)
    {
        s = "changed";
    }
    static void Main()
    {
        string str = "original";
        Method(ref str);
        Console.WriteLine(str);
    }
}
// Output: changed

```

1.2.10 Ключове слово out

Ключове слово **out** використовується для передачі аргументів за покажчиком. Воно схоже на ключове слово **ref**, за винятком того, що **ref** вимагає ініціалізації змінної перед її передачею. Для роботи з параметром **out** визначення методу і викликаючий метод повинні явно використовувати ключове слово **out**. Наприклад:

```

class OutExample
{
    static void Method(out int i)
    {
        i = 44;
    }
    static void Main()
    {

```

```

    int value;
    Method(out value);
    // value is now 44
}

```

Незважаючи на те, що змінні, передані в якості аргументів **out** можуть не ініціалізуватися перед передачею, викликаючий метод повинен присвоїти значення перед поверненням методу. Ключові слова **ref** і **out** по-різному обробляються під час виконання, але в режимі компіляції вони рівноцінні. Отже, якщо один метод приймає аргумент **ref**, а інший - **out**, вони не можуть бути перевантажені. Ці два методи ідентичні, наприклад, при компіляції, тому даний код скомпільовано не буде:

```

class CS0663_Example
{
    // Compiler error CS0663: "Cannot define overloaded
    // methods that differ only on ref and out".
    public void SampleMethod(out int i) { }
    public void SampleMethod(ref int i) { }
}

```

Однак перевантаження можливе, якщо один метод приймає аргумент **ref** або **out**, а інший не приймає жодного, як показано в наступному прикладі:

```

class OutOverloadExample
{
    public void SampleMethod(int i) { }
    public void SampleMethod(out int i) { i = 5; }
}

```

Властивості не є змінними і тому не можуть бути передані в якості параметрів **out**.

Оголошення методу **out** використовується тоді, коли необхідно, щоб метод повертав кілька значень. Метод, який використовує параметр **out**, як і раніше має доступ до змінних як повертається тип (див. return), але може також повертати один або кілька об'єктів викликаючому методу в якості параметрів **out**. У цьому прикладі використовується **out** для повернення трьох змінних з одним викликом методу. Зверніть увагу, що третьому аргументу починається значення 0. Це дозволяє методам повертати значення на вибір:

```

class OutReturnExample
{
    static void Method(out int i, out string s1, out string s2)
    {
        i = 44;
        s1 = "I've been returned";
        s2 = null;
    }
    static void Main()

```

```

{
    int value;
    string str1, str2;
    Method(out value, out str1, out str2);
    // value is now 44
    // str1 is now "I've been returned"
    // str2 is (still) null;
}
}

```

1.2.11 Оператор foreach

Оператор `foreach` фактично є оператором циклу і повторює групу вкладених операторів для кожного елемента масиву або колекції об'єктів, що реалізують інтерфейс `System.Collections.IEnumerable` або `System.Collections.Generic.IEnumerable <T>`. Оператор `foreach` використовується для ітерації колекції з метою отримання необхідної інформації, проте його не слід використовувати для додавання або видалення елементів вихідної колекції щоб уникнути непередбачених побічних ефектів. Якщо потрібно додати або видалити елементи вихідної колекції, слід використовувати цикл `for`. Оператори тіла циклу продовжують виконуватися для кожного елемента масиву або колекції. Після завершення ітерації всіх елементів колекції управління переходить до наступного оператора після блоку `foreach`. У будь-якій точці блоку `foreach` можна розірвати цикл за допомогою ключового слова `break` або перейти до наступної ітерації в циклі за допомогою ключового слова `continue`. Цикл `foreach` також може бути розірваний за допомогою операторів `goto`, `return` або `throw`.

```

class ForEachTest
{
    static void Main(string[] args)
    {
        int[] fibarray = new int[] { 0, 1, 2, 3, 5, 8, 13 };
        foreach (int i in fibarray)
        {
            System.Console.WriteLine(i);
        }
    }
}
/*
Output:
0
1
2
3
5
8
13
*/

```

Контрольні питання.

1. Для чого застосовуються ключові слова “checked/unchecked” ?
2. Для чого застосовується ключове слово “default”?
3. Для чого застосовується ключове слово “is” ?
4. Для чого застосовується оператор “??” ?
5. Для чого застосовується оператор “AS” ?
6. Для чого застосовується ключове слово “base” ?
7. Для чого застосовується ключове слово “virtual” ?
8. Для чого застосовується ключове слово “params” ?
9. Для чого застосовується ключове слово “ref” ?
10. Для чого застосовується ключове слово “out” ?
11. Для чого застосовується оператор “foreach” ?

1.3 Масиви

Масив - це структура даних, що містить кілька змінних одного типу. Масиви оголошуються з наступним типом.

```
type [] arrayName;
```

У наступному прикладі показано створення одновимірних, багатовимірних масивів і масивів масивів.

```
class TestArraysClass
{
    static void Main()
    {
        // Declare a single-dimensional array
        int[] array1 = new int[5];

        // Declare and set array element values
        int[] array2 = new int[] { 1, 3, 5, 7, 9 };

        // Alternative syntax
        int[] array3 = { 1, 2, 3, 4, 5, 6 };

        // Declare a two dimensional array
        int[,] multiDimensionalArray1 = new int[2, 3];

        // Declare and set array element values
        int[,] multiDimensionalArray2 = { { 1, 2, 3 }, { 4, 5, 6 } };

        // Declare a jagged array
```

```

int[][] jaggedArray = new int[6][];

// Set the values of the first array in the jagged array structure
jaggedArray[0] = new int[4] { 1, 2, 3, 4 };
}
}

```

Масив може бути одномірним, багатовимірним або масивом масивів. Значення за замовчуванням числових елементів масиву задано рівним нулю, а елементи посилань мають значення NULL. Невирівняний масив є масивом масивів і тому його елементи є покажчиковими типами і ініціалізуються значенням null. Індксація масивів починається з нуля: масив з елементами n індексується від 0 до n-1. Елементи масиву можуть бути будь-яких типів, включаючи тип масиву. Типи масиву є посилальними типами, похідними від абстрактного базового типу Array. Оскільки цей тип реалізує IEnumerable і IEnumerable<T>, в C# у всіх масивах можна використовувати ітерацію foreach.

У мові C# масиви є **об'єктами**, а не просто суміжними адресованими областями пам'яті, як в C і C++. Array є абстрактним базовим типом для всіх типів масивів. Можна використовувати властивості та інші члени класу, які має Array. У прикладі використовується властивість Length для отримання довжини масиву. У наступному коді довжина масиву numbers, що дорівнює 5, присвоюється змінній lengthOfNumbers:

```

int[] numbers = { 1, 2, 3, 4, 5 };
int lengthOfNumbers = numbers.Length;

```

Клас System.Array дозволяє використовувати багато інших корисних методів і властивостей для виконання сортування, пошуку і копіювання масивів.

У цьому прикладі властивість Rank використовується для відображення числа вимірів масиву.

```

class TestArraysClass
{
    static void Main()
    {
        // Declare and initialize an array:
        int[,] theArray = new int[5, 10];
        System.Console.WriteLine("The array has {0} dimensions.", theArray.Rank);
    }
}
// Output: The array has 2 dimensions.

```

1.3.1 Одновимірні масиви

Можна оголосити масив з п'яти цілих чисел, як показано в наступному прикладі:

```

int[] array = new int[5];

```


Масив містить елементи із `array[0]` по `array[4]`. Оператор **new** служить для створення масиву і ініціалізації елементів масиву зі значеннями за замовчуванням. У даному прикладі елементи масиву ініціалізуються значенням 0. Масив, в якому зберігаються рядкові елементи, можна оголосити таким же чином. Приклад:

```
string[] stringArray = new string[6];
```

Масив можна ініціалізувати при оголошенні. У цьому випадку специфікація рангу не потрібна, оскільки вона вже надана за кількістю елементів у списку ініціалізації:

```
int[] array1 = new int[] { 1, 3, 5, 7, 9 };
```

Рядковий масив можна ініціалізувати таким же чином. Нижче наведено оголошення рядкового масиву, в якому кожен елемент ініціалізується назвою дня:

```
string[] weekDays = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
```

Можна оголосити змінну масиву без ініціалізації, але при присвоєнні масиву цієї змінної потрібно використовувати оператор **new**.

```
int[] array3;  
array3 = new int[] { 1, 3, 5, 7, 9 }; // OK  
//array3 = {1, 3, 5, 7, 9}; // Error
```

Розглянемо наступні оголошення масиву:

```
SomeType[] array4 = new SomeType[10];
```

Результат цього оператора залежить від того, чи є `SomeType` типом значення або посилальним типом. Якщо це тип значення, оператор створює масив з 10 екземплярів типу `SomeType`. Якщо `SomeType` - контрольний тип, оператор створює масив з 10 елементів, Кожен з яких ініціалізується нульової посиланням.

1.3.2 Багатовимірні масиви

Масиви можуть мати кілька вимірів. Наприклад, наступне оголошення створює двовимірний масив з чотирьох рядків і двох стовпців:

```
int[,] array = new int[4, 2];
```

А наступне оголошення створює тривимірний масив з кількістю елементів 4,2 і 3:

```
int[, ] array1 = new int[4, 2, 3];
```

Масив можна ініціалізувати при оголошенні, як показано в наступному прикладі:

```
int[,] array2D = new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };  
int[, ] array3D = new int[, ] { { { 1, 2, 3 } }, { { 4, 5, 6 } } };
```

Можна також ініціалізувати масив, не зазначаючи його розмірність:

```
int[,] array4 = { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
```

Якщо потрібно створити змінну масиву без ініціалізації, то необхідно використовувати оператор new, щоб присвоїти масив змінної. Наприклад:

```
int[,] array5;  
array5 = new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } }; // OK  
//array5 = {{1,2}, {3,4}, {5,6}, {7,8}}; // Error
```

Можна також присвоїти значення елементу масиву, наприклад:

```
array5[2, 1] = 25;
```

У наступному прикладі коду змінні масивів ініціалізується значеннями за замовчуванням (за винятком масивів масивів):

```
int[,] array6 = new int[10, 10];
```

1.3.3 Масиви масивів

Масив масивів - це масив, елементи якого самі є масивами. Елементи масиву масивів можуть мати різні розміри і вимірювання. Масиви масивів іноді також називаються "невіривняні масивами". У наступних прикладах показано, як виконується оголошення, ініціалізація і доступ до масивів масивів. Нижче показано оголошення одновимірного масиву, що включає три елементи, кожен з яких є одновимірним масивом цілих чисел.

```
int[][] jaggedArray = new int[3][];
```

Перед використанням jaggedArray його елементи потрібно ініціалізувати. Зробити це можна таким чином.

```
jaggedArray[0] = new int[5];  
jaggedArray[1] = new int[4];  
jaggedArray[2] = new int[2];
```

Кожен елемент являє собою одномірний масив цілих чисел. Перший елемент масиву складається з п'яти цілих чисел, другий - з чотирьох і третій - з двох.

Для заповнення елементів масиву значеннями можна також використовувати ініціалізатори, при цьому розмір масиву знати не потрібно. Наприклад:

```
jaggedArray[0] = new int[] { 1, 3, 5, 7, 9 };  
jaggedArray[1] = new int[] { 0, 2, 4, 6 };  
jaggedArray[2] = new int[] { 11, 22 };
```

Також масив можна ініціалізувати шляхом оголошення:

```
int[][] jaggedArray2 = new int[][]  
{  
    new int[] {1,3,5,7,9},  
    new int[] {0,2,4,6},  
    new int[] {11,22}  
};
```

Також можна використовувати скорочену форму. Зверніть увагу, що при ініціалізації елементів оператор **new** опускати не можна, так як ініціалізації за замовчуванням для цих елементів не існує.

```
int[][] jaggedArray3 =  
{  
    new int[] {1,3,5,7,9},  
    new int[] {0,2,4,6},  
    new int[] {11,22}  
};
```

Невирівняні масив є масивом масивів, і тому його елементи є посилальними типами і ініціалізуються значенням null. Доступ до окремих елементів масиву виконується наступним чином:

```
// Assign 77 to the second element ([1]) of the first array ([0]):  
jaggedArray3[0][1] = 77;
```

```
// Assign 88 to the second element ([1]) of the third array ([2]):  
jaggedArray3[2][1] = 88;
```

Масиви масивів можна змішувати з багатовимірними масивами. Нижче показано оголошення і ініціалізація одновимірного масиву масивів, що складається з двовимірних елементів різного розміру:

```
int[][,] jaggedArray4 = new int[3][,]  
{  
    new int[,] { {1,3}, {5,7} },  
    new int[,] { {0,2}, {4,6}, {8,10} },  
};
```

```
new int[,] { {11,22}, {99,88}, {0,9} }
};
```

Доступ до окремих елементів виконується як показано в прикладі нижче, де відображено значення елемента [1,0] першого масиву (значення 5).

```
System.Console.Write("{0}", jaggedArray4[0][1, 0]);
```

Метод Length повертає число масивів, що містяться в масиві масивів. Наприклад, якщо оголосити попередній масив, ми отримаємо наступне:

```
System.Console.WriteLine(jaggedArray4.Length);
```

повертає значення 3.

У цьому прикладі виконується створення масиву, елементи якого самі є масивами. Кожен елемент масиву має власний розмір:

```
class ArrayTest
{
    static void Main()
    {
        // Declare the array of two elements:
        int[][] arr = new int[2][];

        // Initialize the elements:
        arr[0] = new int[5] { 1, 3, 5, 7, 9 };
        arr[1] = new int[4] { 2, 4, 6, 8 };

        // Display the array elements:
        for (int i = 0; i < arr.Length; i++)
        {
            System.Console.Write("Element({0}): ", i);

            for (int j = 0; j < arr[i].Length; j++)
            {
                System.Console.Write("{0}{1}", arr[i][j], j ==
                    (arr[i].Length - 1) ? "" : " ");
            }
            System.Console.WriteLine();
        }
        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}

/* Output:
Element(0): 1 3 5 7 9
Element(1): 2 4 6 8
*/
```

1.3.4 Передача масивів за допомогою параметрів **ref** і **out**

Як і всі параметри **out**, параметр **out** типу масиву перед використанням необхідно присвоїти:

```
static void TestMethod1(out int[] arr)
{
    arr = new int[10]; // definite assignment of arr
}
```

Як і всі параметри **ref**, параметр **ref** типу масиву повинен бути певним чином присвоєний. Параметр **ref** типу масиву можна змінювати як результат виклику. Наприклад, масиву можна присвоїти значення **null**, або ж його можна ініціалізувати в інший масив. Приклад:

```
static void TestMethod2(ref int[] arr)
{
    arr = new int[10]; // arr initialized to a different array
}
```

У наступному прикладі масив `theArray` оголошений в викликаючому методі (метод `Main`) і ініціалізований в методі `FillArray`. Потім елементи масиву повертаються викликаючому методу і відображаються:

```
class TestOut
{
    static void FillArray(out int[] arr)
    {
        // Initialize the array:
        arr = new int[5] { 1, 2, 3, 4, 5 };
    }

    static void Main()
    {
        int[] theArray; // Initialization is not required

        // Pass the array to the callee using out:
        FillArray(out theArray);

        // Display the array elements:
        System.Console.WriteLine("Array elements are:");
        for (int i = 0; i < theArray.Length; i++)
        {
            System.Console.Write(theArray[i] + " ");
        }

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
```

```

/* Output:
  Array elements are:
  1 2 3 4 5
*/

```

У цьому прикладі масив theArray ініціалізований в викликаючому методі(метод Main) і підставляється в метод FillArray за допомогою параметра **ref**. Деякі з елементів масиву оновлюються в методі FillArray. Потім елементи масиву повертаються викликаючому і відображаються:

```

class TestRef
{
    static void FillArray(ref int[] arr)
    {
        // Create the array on demand:
        if (arr == null)
        {
            arr = new int[10];
        }
        // Fill the array:
        arr[0] = 1111;
        arr[4] = 5555;
    }

    static void Main()
    {
        // Initialize the array:
        int[] theArray = { 1, 2, 3, 4, 5 };

        // Pass the array using ref:
        FillArray(ref theArray);

        // Display the updated array:
        System.Console.WriteLine("Array elements are:");
        for (int i = 0; i < theArray.Length; i++)
        {
            System.Console.Write(theArray[i] + " ");
        }

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}

/* Output:
  Array elements are:
  1111 2 3 4 5555
*/

```

Контрольні питання.

1. Як оголошуються одновимірні масиви?

2. Як оголошуються багатовимірні масиви?
3. Чим масиви масивів відрізняються від багатовимірних масивів?
4. Як здійснюється передача масивів «в» та «з» методів?

1.4 Класи, інтерфейси та ООП

Оскільки об'єктно-орієнтоване програмування (ООП) на сучасному етапі є основним способом написання програм великого розміру, розглянемо його більш детально. На відміну від C++, в C# заборонене множинне наслідування класів, натомість введено поняття інтерфейсів, які допускають множинне наслідування, проте не можуть мати своїх реалізацій.

Клас - це логічна структура, яка дозволяє створювати свої власні користувацькі типи шляхом групування змінних інших типів, методів і подій. Він визначає дані і поведінку типу. Якщо клас не оголошений статичним, то клієнтський код може його використовувати, створивши об'єкти або екземпляри, призначені змінній. Зміна залишається в пам'яті, поки всі посилання на неї не вийдуть з області видимості. У цей час середовище CLR позначає її придатною для збирача сміття. Якщо клас оголошується статичним, то в пам'яті залишається тільки одна копія і клієнтський код може отримати до неї доступ тільки за допомогою самого класу, а не змінною екземпляра. На відміну від структур класи підтримують **успадкування**, фундаментальну характеристику об'єктно - орієнтованого програмування.

Класи оголошуються за допомогою ключового слова `class`, як показано в наступному прикладі:

```
public class Customer
{
    //Fields, properties, methods and events go here...
}
```

Ключовим словом `class` задається рівень доступу. Оскільки в даному випадку використовується `public`, будь-хто може створювати об'єкти з цього класу. Ім'я класу вказується після ключового слова `class`. Інша частина визначення є тілом класу, в якому задаються дані і поведінка. Поля, властивості, методи і події в класі позначаються терміном члени класу.

1.4.1 Створення об'єктів

Клас і об'єкт - це різні поняття, хоча в деяких випадках вони взаємозамінні. Клас визначає тип об'єкта, але не сам об'єкт. Об'єкт - це конкретна сутність, заснована на класі і іноді може бути названа екземпляром класу.

Об'єкти можна створювати за допомогою ключового слова `new`, за яким слідує ім'я класу, на якому буде заснований об'єкт:

```
Customer object1 = new Customer();
```

При створенні екземпляра класу показчик на цей об'єкт передається програмісту. У попередньому прикладі `object1` є показником на об'єкт, базований на `Customer`. Це й показчик вказує на новий об'єкт, але не містить дані цього об'єкта. Фактично, можна створити посилання на об'єкт без створення самого об'єкта:

```
Customer object2;
```

Створення таких показчиків, які не вказують на об'єкт, не рекомендується, так як спроба доступу до об'єкта по такому показчику призведе до збою під час виконання. Однак такий показчик можна зробити вказуючим на об'єкт, створивши новий об'єкт або призначивши її існуючому об'єкту:

```
Customer object3 = new Customer();  
Customer object4 = object3;
```

У даному коді створюються два показчики на об'єкти, які вказують на один об'єкт. Тому будь-які зміни об'єкта, виконані за допомогою `object3`, будуть видимі при подальшому використанні `object4`. Оскільки на об'єкти, базовані на класах, вказують показчики, класи іноді називають показчиковими типами.

1.4.2 Наслідування (успадкування) класів

Наслідування виконується за допомогою утворення похідних, тобто клас оголошується за допомогою базового класу, від якого він успадковує дані і поведінку (методи-тобто фактично функції). Базовий клас задається додаванням після імені похідного класу двокрапки й назви класу:

```
public class Manager : Employee  
{  
    // Employee fields, properties, methods and events are inherited  
    // New Manager fields, properties, methods and events go here...  
}
```

Коли клас оголошується від базового типу, він успадковує всі члени базового класу, **за винятком конструкторів**.

На відміну від C++, клас в C# може безпосередньо бути утвореним тільки від одного базового класу. Однак, оскільки базовий клас може сам успадковуватись від іншого класу, клас може побічно успадковувати кілька базових класів. Крім того, клас може безпосередньо реалізувати декілька інтерфейсів.

Клас може бути **абстрактним**. Абстрактний клас містить абстрактні методи, які мають визначення сигнатури (оголошення), але не мають реалізації.

Створювати реалізацію абстрактних класів не можна. Вони можуть використовуватися тільки за допомогою похідних від них класів, що реалізують абстрактні методи. І навпаки, **запечатаний** клас не дозволяє іншим класам бути від нього похідними. Визначення класів можна розділити між різними вихідними файлами.

1.4.3 Модифікатор **abstract**

Модифікатор **abstract** вказує, що реалізація змінюваного об'єкта є неповною або відсутня. Модифікатор **abstract** може використовуватися з класами, методами, властивостями, індексаторами та подіями. Модифікатор **abstract** в оголошенні класу вказує, що клас призначений тільки для використання в якості базового класу для інших класів. Члени, помічені як абстрактні або включені до абстрактний клас, повинні бути реалізовані за допомогою класів, похідних від абстрактних класів.

У цьому прикладі клас **Square** повинен надати реалізацію **Area**, оскільки він є похідним від **ShapesClass**:

```
abstract class ShapesClass
{
    abstract public int Area();
}

class Square : ShapesClass
{
    int side = 0;

    public Square(int n) // Конструктор
    {
        side = n;
    }

    // Area метод потрібний для уникнення помилки компіляції.
    public override int Area()
    {
        return side * side;
    }

    static void Main()
    {
        Square sq = new Square(12);
        Console.WriteLine("Площа квадрату = {0}", sq.Area());
    }
}
```

Далі представлені деякі особливості абстрактних класів.

- Створювати екземпляри абстрактного класу не можна.
- Абстрактні класи можуть містити абстрактні методи і методи доступу.

- Абстрактний клас з модифікатором **sealed** визначити не можна. Модифікатор **sealed** забороняє успадковувати клас, у той час як модифікатор **abstract** вказує, що клас повинен мати похідні класи.
- Неабстрактний клас, який є похідним від абстрактного, повинен містити фактичні реалізації всіх успадкованих абстрактних методів і методів доступу.
- Щоб вказати відсутність реалізації в методі чи властивості, скористайтеся модифікатором **abstract** в оголошенні методу або властивості.
- Абстрактний метод є неявним віртуальним методом.
- Оголошення абстрактних методів допускаються тільки в абстрактних класах.
- Оскільки оголошення абстрактного методу не надає фактичної реалізації, тіло методу відсутнє; оголошення методу просто закінчується крапкою з комою, а після сигнатури ставити фігурні дужки {} не потрібно. наприклад:

```
public abstract void MyMethod();
```

- Реалізація надається методом перевизначення **override**, який є членом неабстрактного класу. Використання статичних або віртуальних модифікаторів в оголошенні абстрактного методу є неприпустимим.
- Дія абстрактних властивостей аналогічно абстрактним методам, за винятком відмінностей в синтаксисі оголошень і викликів.
- Використання модифікатора **abstract** в статичній властивості є неприпустимим.
- Абстрактна успадкована властивість може бути перевизначена в похідному класі за рахунок включення оголошення властивості, що використовує модифікатор перевизначення.

Абстрактний клас повинен надавати реалізацію для всіх членів інтерфейсу. Абстрактний клас, який реалізує інтерфейс, може відображати методи інтерфейсу в абстрактних методах. Наприклад:

```
interface InterfaceTest
{
    void M();
}
abstract class C : InterfaceTest
{
    public abstract void M();
}
```

У наступному прикладі клас `DerivedClass` є похідним від абстрактного класу `BaseClass`. Абстрактний клас містить абстрактний метод `AbstractMethod` і дві абстрактні властивості - `X` і `Y`.

```

abstract class BaseClass // Abstract class
{
    protected int _x = 100;
    protected int _y = 150;
    public abstract void AbstractMethod(); // Abstract method
    public abstract int X { get; }
    public abstract int Y { get; }
}

class DerivedClass : BaseClass
{
    public override void AbstractMethod()
    {
        _x++;
        _y++;
    }

    public override int X // overriding property
    {
        get
        {
            return _x + 10;
        }
    }

    public override int Y // перевизначена властивість
    {
        get
        {
            return _y + 10;
        }
    }

    static void Main()
    {
        DerivedClass o = new DerivedClass();
        o.AbstractMethod();
        Console.WriteLine("x = {0}, y = {1}", o.X, o.Y);
    }
}
// Output: x = 111, y = 161

```

1.4.4 Метод **override**

Метод **override** забезпечує нову реалізацію члена, успадкованого від базового класу. Метод, перевизначений оголошенням **override**, називається **перевизначеним базовим методом**. Перевизначений базовий метод повинен мати ту ж сигнатуру, що й метод **override**.

Невіртуальний або статичний метод не можна перевизначити. Перевизначений базовий метод повинен мати тип **virtual**, **abstract** або **override**.

Оголошення **override** не може змінити доступність методу **virtual**. Методи **override** і **virtual** повинні мати однаковий модифікатор рівня доступу.

Модифікатори **new**, **static** і **virtual** не можна використовувати для зміни методу **override**. Переобумовлене оголошення властивості повинне задавати такі ж модифікатор рівня доступу, тип та ім'я, які є у успадкованої властивості, а перевизначена властивість повинна мати тип **virtual**, **abstract** або **override**.

У наступному прикладі визначається базовий клас з ім'ям **Employee** і похідний клас з ім'ям **SalesEmployee**. Клас **SalesEmployee** включає додаткову властивість **salesbonus**, для використання якого перевизначається метод **CalculatePay**.

```
class TestOverride
{
    public class Employee
    {
        public string name;

        // Basepay is defined as protected, so that it may be
        // accessed only by this class and derived classes.
        protected decimal basepay;

        // Constructor to set the name and basepay values.
        public Employee(string name, decimal basepay)
        {
            this.name = name;
            this.basepay = basepay;
        }

        // Declared virtual so it can be overridden.
        public virtual decimal CalculatePay()
        {
            return basepay;
        }
    }

    // Derive a new class from Employee.
    public class SalesEmployee : Employee
    {
        // New field that will affect the base pay.
        private decimal salesbonus;

        // The constructor calls the base-class version, and
        // initializes the salesbonus field.
        public SalesEmployee(string name, decimal basepay,
            decimal salesbonus) : base(name, basepay)
        {
            this.salesbonus = salesbonus;
        }
    }
}
```

```

// Override the CalculatePay method
// to take bonus into account.
public override decimal CalculatePay()
{
    return basepay + salesbonus;
}
}

static void Main()
{
    SalesEmployee employee1 = new SalesEmployee("Alice",
        1000, 500);
    Employee employee2 = new Employee("Jack", 1100);

    Console.WriteLine("Employee4 " + employee1.name +
        " earned: " + employee1.CalculatePay());
    Console.WriteLine("Employee4 " + employee2.name +
        " earned: " + employee2.CalculatePay());
}
}
/*
Output:
Employee4 Alice earned: 1500
Employee4 Jack earned: 1100
*/

```

1.4.5 Ключове слово **virtual**

Ключове слово **virtual** використовується для зміни оголошень методів, властивостей, індексаторів і подій і забезпечує дозвіл їх перевизначення в похідному класі. Наприклад, цей метод може бути перевизначений будь-яким похідним класом:

```

public virtual double Area()
{
    return x * y;
}

```

При виклику віртуального методу тип об'єкта під час виконання програми перевіряється на перевизначення члена. Викликається перевизначення члена в найдальшому класі. Це може бути початковий член, якщо ніякий похідний клас не виконував перевизначення цього члена.

Типово методи не є віртуальними. Невіртуальний метод не можна перевизначити. Модифікатор **virtual** не можна використовувати з модифікаторами **static**, **abstract**, **private** або **override**. У наступному прикладі показана віртуальна властивість:

```

class MyBaseClass
{
    // virtual auto-implemented property. Overrides can only

```

```

// provide specialized behavior if they implement get and set accessors.
public virtual string Name { get; set; }

// ordinary virtual property with backing field
private int num;
public virtual int Number
{
    get { return num; }
    set { num = value; }
}
}

class MyDerivedClass : MyBaseClass
{
    private string name;

    // Override auto-implemented property with ordinary property
    // to provide specialized accessor behavior.
    public override string Name
    {
        get
        {
            return name;
        }
        set
        {
            if (value != String.Empty)
            {
                name = value;
            }
            else
            {
                name = "Unknown";
            }
        }
    }
}
}

```

Дія віртуальних властивостей аналогічна до абстрактних методів, за винятком відмінностей в синтаксисі оголошень і викликів. Використання модифікатора **virtual** в статичному властивості є неприпустимим. Віртуальна успадкованв властивість може бути перевизначена в похідному класі шляхом включення оголошення властивості, яка використовує модифікатор **override**.

У наступному прикладі клас **Shape** містить дві координати **x**, **y** і віртуальний метод **Area()**. Різні класи фігур, такі як **Circle**, **Cylinder** і **Sphere**, успадковують клас **Shape**, і для кожної фігури обчислюється площа поверхні. Кожен похідний клас має власну реалізацією перевизначеного методу **Area ()**.

Зверніть увагу, що успадковані класи Circle, Sphere і Cylinder використовують конструктор, який ініціалізує базовий клас так, як показано в наступному оголошенні:

```
public Cylinder(double r, double h): base(r, h) {}
```

Наступна програма обчислює і відображає відповідну площу для кожної фігури, шляхом виклику відповідної реалізації методу Area (), який відповідає своєму об'єкту.

```
class TestClass
{
    public class Shape
    {
        public const double PI = Math.PI;
        protected double x, y;
        public Shape()
        {
        }
        public Shape(double x, double y)
        {
            this.x = x;
            this.y = y;
        }

        public virtual double Area()
        {
            return x * y;
        }
    }

    public class Circle : Shape
    {
        public Circle(double r) : base(r, 0)
        {
        }

        public override double Area()
        {
            return PI * x * x;
        }
    }

    class Sphere : Shape
    {
        public Sphere(double r) : base(r, 0)
        {
        }

        public override double Area()
```

```

    {
        return 4 * PI * x * x;
    }
}

class Cylinder : Shape
{
    public Cylinder(double r, double h) : base(r, h)
    {
    }

    public override double Area()
    {
        return 2 * PI * x * x + 2 * PI * x * y;
    }
}

static void Main()
{
    double r = 3.0, h = 5.0;
    Shape c = new Circle(r);
    Shape s = new Sphere(r);
    Shape l = new Cylinder(r, h);
    // Display results:
    Console.WriteLine("Area of Circle = {0:F2}", c.Area());
    Console.WriteLine("Area of Sphere = {0:F2}", s.Area());
    Console.WriteLine("Area of Cylinder = {0:F2}", l.Area());
}
}
/*
Output:
Area of Circle = 28.27
Area of Sphere = 113.10
Area of Cylinder = 150.80
*/

```

1.4.6 Поліформізм

Поліформізм - це грецьке слово, що означає "наявність багатьох форм". Це поняття має два аспекти, що розрізняються. Під час виконання об'єкти похідного класу можуть розглядатися як об'єкти базового класу в таких місцях програми, як параметри методу і колекції масивів. При цьому оголошений тип об'єкта більше не ідентичний його типу під час виконання програми. Базові класи можуть визначати і реалізовувати віртуальні методи, а похідні класи можуть перевизначати їх. Це означає, що вони надають свої власні визначення і реалізацію. Під час виконання, коли клієнтський код викликає метод, середовище CLR шукає дійсний тип об'єкта і викликає саме це перевизначення віртуального методу. Таким чином, у вихідному коді можна викликати метод в базовому класі і викликати метод з версією похідного класу.

Віртуальні методи дозволяють єдиним чином працювати з групами пов'язаних об'єктів. Наприклад, є програма малювання, яке дає можливість користувачеві створювати на поверхні для відображення різні геометричні форми. Під час компіляції невідомо, які конкретні форми буде створювати користувач. Але програма повинна враховувати всі типи створюваних форм, і повинна оновлювати їх у відповідь на дії миші користувача. Поліморфізм можна використовувати для вирішення цієї проблеми у двох основних етапах.

- Створення ієрархії класів, в якій клас кожної конкретної форми проводиться від загального базового класу.
- Використання віртуального методу для виклику відповідного методу в якому-небудь похідному класі одним викликом методу базового класу.

Для цього необхідно створити базовий клас, (наприклад **Shape**), і похідні класи, наприклад **Rectangle**, **Circle** і **Triangle**. Клас Shape повинен містити віртуальний метод, названий Draw, який буде перевизначений в кожному похідному класі для малювання конкретної форми, яку представляє клас. В створеному об'єкті List <Shape> можна додати коло, трикутник і прямокутник. Для відновлення поверхні відображення можна використовувати цикл foreach для ітерації списку і виклику методу Draw на кожному об'єкті Shape в списку. Хоча кожен об'єкт у списку має оголошений тип Shape, буде викликатися саме тип часу виконання (перевизначена версія методу в кожному похідному класі).

```
public class Shape
{
    // A few example members
    public int X { get; private set; }
    public int Y { get; private set; }
    public int Height { get; set; }
    public int Width { get; set; }

    // Virtual method
    public virtual void Draw()
    {
        Console.WriteLine("Performing base class drawing tasks");
    }
}

class Circle : Shape
{
    public override void Draw()
    {
        // Code to draw a circle...
        Console.WriteLine("Drawing a circle");
        base.Draw();
    }
}

class Rectangle : Shape
{

```

```

    public override void Draw()
    {
        // Code to draw a rectangle...
        Console.WriteLine("Drawing a rectangle");
        base.Draw();
    }
}
class Triangle : Shape
{
    public override void Draw()
    {
        // Code to draw a triangle...
        Console.WriteLine("Drawing a triangle");
        base.Draw();
    }
}

class Program
{
    static void Main(string[] args)
    {
        // Polymorphism at work #1: a Rectangle, Triangle and Circle
        // can all be used wherever a Shape is expected. No cast is
        // required because an implicit conversion exists from a derived
        // class to its base class.
        System.Collections.Generic.List<Shape> shapes = new
System.Collections.Generic.List<Shape>();
        shapes.Add(new Rectangle());
        shapes.Add(new Triangle());
        shapes.Add(new Circle());

        // Polymorphism at work #2: the virtual method Draw is
        // invoked on each of the derived classes, not the base class.
        foreach (Shape s in shapes)
        {
            s.Draw();
        }

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

```

```

/* Output:
Drawing a rectangle
Performing base class drawing tasks
Drawing a triangle
Performing base class drawing tasks
Drawing a circle

```

*/

У С# кожен тип є поліморфним, оскільки всі типи, включаючи користувацькі типи, успадковуються від Object.

1.4.7 Віртуальні члени

Якщо похідний клас успадковується від базового класу, то він набуває всіх методів, полів, властивостей та подій базового класу. Проектувальник похідного класу може вибрати з можливих варіантів:

- перевизначити віртуальні члени в базовому класі;
- успадковувати найближчі методи базового класу без його перевизначення;
- визначити нову не віртуальну реалізацію цих членів, яка приховує реалізації базового класу.

Похідний клас може перевизначити член базового класу, якщо тільки член базового класу оголошений як віртуальний або абстрактний. Похідний член повинен використовувати ключове слово **override**, щоб явно вказати, що метод повинен брати участь у віртуальному виклику. Наприклад:

```
public class BaseClass
{
    public virtual void DoWork() { }
    public virtual int WorkProperty
    {
        get { return 0; }
    }
}
public class DerivedClass : BaseClass
{
    public override void DoWork() { }
    public override int WorkProperty
    {
        get { return 0; }
    }
}
```

Поля не можуть бути віртуальними. Віртуальними можуть бути тільки методи, властивості, події та індексатори. Якщо в похідному класі віртуальний метод перевизначається, то цей член викликається навіть у тому випадку, якщо доступ до екземпляра цього класу здійснюється як до екземпляра базового класу. Наприклад:

```
DerivedClass B = new DerivedClass();
B.DoWork(); // Calls the new method.
```

```
BaseClass A = (BaseClass)B;
A.DoWork(); // Also calls the new method.
```

Віртуальні методи і властивості дають можливість похідним класам розширювати базовий клас, без необхідності використання реалізації методу базового класу.

Якщо необхідно, щоб похідний член мав те ж ім'я, що і член базового класу, але не потрібно, щоб він брав участь у віртуальному виклику, можна використовувати ключове слово **new**. Ключове слово **new** розташовується перед типом члена класу, який повинен бути заміщений. Наприклад:

```
public class BaseClass
{
    public void DoWork() { WorkField++; }
    public int WorkField;
    public int WorkProperty
    {
        get { return 0; }
    }
}

public class DerivedClass : BaseClass
{
    public new void DoWork() { WorkField++; }
    public new int WorkField;
    public new int WorkProperty
    {
        get { return 0; }
    }
}
```

До прихованих членів базового класу можна продовжувати звертатися з клієнтського коду, привівши екземпляр похідного класу до екземпляру базового класу. Наприклад:

```
DerivedClass B = new DerivedClass();
B.DoWork(); // Calls the new method.

BaseClass A = (BaseClass)B;
A.DoWork(); // Calls the old method.
```

Віртуальні члени залишаються віртуальними без будь-яких обмежень щодо кількості класів, оголошених між віртуальним членом і класом, в якому він був спочатку оголошений. Якщо в класі А оголошується віртуальний член і клас В є похідним від класу А, а клас С є похідним від класу В, то клас С успадковує віртуальний член і забезпечує можливість його перевизначення незалежно від того, чи було в класі В оголошено перевизначення цього члена. Наприклад :

```
public class A
{
```

```

    public virtual void DoWork() { }
}
public class B : A
{
    public override void DoWork() { }
}

```

1.4.8 Ключове слово **sealed**

Похідний клас може припинити успадкування, оголосивши перевизначення як **sealed**. Для цього потрібно додати ключове слово **sealed** перед ключовим словом **override** в оголошенні члена класу. У коді нижче наведений приклад:

```

public class C : B
{
    public sealed override void DoWork() { }
}

```

У попередньому прикладі метод `DoWork` більше не є віртуальним для будь-якого класу, похідного від `C`. Він як і раніше є віртуальним для екземплярів `C`, навіть якщо вони перетворюються на тип `B` або `A`. Запечатані методи можуть бути замінені похідними класами за допомогою ключового слова **new**, як показано в наступному прикладі:

```

public class D : C
{
    public new void DoWork() { }
}

```

У цьому випадку при виклику методу `DoWork` для типу `D` з використанням змінної типу `D` викликається новий метод `DoWork`. Якщо для доступу до екземпляра типу `D` використовується змінна типу `C`, `B` або `A`, то виклик методу `DoWork` буде підкорятися правилам віртуального наслідування, направляючи виклики реалізації методу `DoWork` в класі `C`.

Похідний клас з заміщеним або перевизначеним методом або властивістю зберігає доступ до методу або властивості базового класу за допомогою ключового слова **base**. У коді нижче наведено приклад:

```

public class Base
{
    public virtual void DoWork() { /* ... */ }
}
public class Derived : Base
{
    public override void DoWork()
    {
        //Perform Derived's work here
    }
}

```

```

    //...
    // Call DoWork on base class
    base.DoWork();
}
}

```

Доступ до базового класу дозволений тільки в конструкторі, методі примірника чи методі доступу екземпляра.

Ключове слово **base** використовується для доступу до членів базового класу з похідного класу:

- Виклик методу базового класу, який був перевизначений іншим методом.
- Визначення конструктора базового класу, який повинен викликатися при створенні екземплярів похідного класу.

Використання ключового слова **base** в статичному методі є неприпустимим. Базовий клас, до якого здійснюється доступ, є базовим класом, заданим в оголошенні класу. Наприклад, якщо вказати `class ClassB : ClassA`, члени `ClassA` будуть доступні з `ClassB`, незалежно від базового класу `ClassA`.

У наступному прикладі метод `GetInfo` присутній як в базовому класі `Person`, так і в похідному класі `Employee`. За рахунок використання ключового слова `base` можна викликати метод `GetInfo` базового класу з похідного класу.

```

public class Person
{
    protected string ssn = "333-55-6677";
    protected string name = "John M. Olson";

    public virtual void GetInfo()
    {
        Console.WriteLine("Name: {0}", name);
        Console.WriteLine("SSN: {0}", ssn);
    }
}

class Employee : Person
{
    public string id = "BBC567EFG";
    public override void GetInfo()
    {
        // Calling the base class GetInfo method:
        base.GetInfo();
        Console.WriteLine("Employee ID: {0}", id);
    }
}

class TestClass
{
    static void Main()
    {

```

```

        Employee E = new Employee();
        E.GetInfo();
    }
}
/*
Output
Name: John M. Olson
SSN: 333-55-6677
Employee ID: BBC567EFG
*/

```

У наступному прикладі показано, як задати конструктор базового класу, що викликається при створенні примірників похідного класу:

```

public class BaseClass
{
    int num;

    public BaseClass()
    {
        Console.WriteLine("in BaseClass()");
    }

    public BaseClass(int i)
    {
        num = i;
        Console.WriteLine("in BaseClass(int i)");
    }

    public int GetNum()
    {
        return num;
    }
}

public class DerivedClass : BaseClass
{
    // This constructor will call BaseClass.BaseClass()
    public DerivedClass() : base()
    {
    }

    // This constructor will call BaseClass.BaseClass(int i)
    public DerivedClass(int i) : base(i)
    {
    }

    static void Main()
    {
        DerivedClass md = new DerivedClass();
    }
}

```

```

        DerivedClass md1 = new DerivedClass(1);
    }
}
/*
Output:
in BaseClass()
in BaseClass(int i)
* /

```

1.4.9 Деструктори

Деструктори використовуються для знищення екземплярів класів.

У структурах визначення деструкторів неможливо. Вони застосовуються тільки в класах. Клас може мати тільки один деструктор. Деструктори не можуть успадковуватися або перевантажуватися. Деструктори неможливо викликати. Вони запускаються автоматично. Деструкція не приймає модифікатори і не має параметрів. Наприклад, наступна інструкція є оголошенням деструктора класу Car:

```

class Car
{
    ~Car() // destructor
    {
        // cleanup statements...
    }
}

```

Деструктор неявним чином викликає метод Finalize для базового класу об'єкта. Отже, попередній код деструктора неявним чином перетвориться в наступний код:

```

protected override void Finalize()
{
    try
    {
        // Cleanup statements...
    }
    finally
    {
        base.Finalize();
    }
}

```

Це означає, що метод Finalize викликається рекурсивно для всіх екземплярів ланцюжка спадкування починаючи з найдалшого і закінчуючи найпершим.

Порожні деструктори використовувати не слід. Якщо клас містить деструктор, то в черзі методу Finalize створюється запис. При виклику деструктора викликається збирач сміття, який виконує обробку черги. Якщо деструктор порожній, це призводить лише до непотрібного зниження продуктивності.

Програміст не може керувати моментом виклику деструктора, тому що момент виклику визначається збирачем сміття. Збирач сміття перевіряє наявність об'єктів, які більше не використовуються програмою. Якщо він вважає, що який-небудь об'єкт вимагає знищення, то він викликає деструктор (за його наявності) та звільняє пам'ять, використовувану для зберігання цього об'єкта. Деструктори також викликаються при виході з програми. Існує можливість примусово виконати збірку сміття, викликавши метод `Collect`, але в більшості випадків цього слід уникати, оскільки це може призвести до проблем з продуктивністю.

У цілому, мова `C#` не вимагає управління пам'яттю в тій мірі, в якій це потрібно у разі розробки коду мовою, яка не розрахована на середовище виконання із збіркою сміття. Це пов'язано з тим, що збирач сміття платформи `NET Framework` неявним чином керує виділенням і вивільненням пам'яті для об'єктів. Однак при інкапсуляції програмою некерованих ресурсів, наприклад вікон, файлів і мережевих підключень, для вивільнення цих ресурсів слід використовувати деструктори. Якщо об'єкт вимагає знищення, то збирач сміття запускає виконання методу `Finalize` цього об'єкта.

У разі, коли програмою використовується цінний зовнішній ресурс, також рекомендується забезпечити спосіб вивільнення цього ресурсу явним чином, перш ніж збирач сміття звільнить цей об'єкт. Це виконується шляхом реалізації методу **`Dispose`** інтерфейсу **`IDisposable`**, який виконує необхідну очистку для об'єкта. Це може значно підвищити продуктивність програми. Навіть у випадку використання даного явного управління ресурсами деструктор стає резервним засобом очищення ресурсів, якщо виклик методу `Dispose` не буде виконаний.

У наступному прикладі створюються три класи, що утворюють ланцюжок успадкування. Клас `First` є базовим, клас `Second` є похідним від класу `First`, а клас `Third` є похідним від класу `Second`. Всі три класи мають деструктори. У методі `Main ()` створюється екземпляр найдальшого в ланцюжку успадкування класу. При виконанні програми зверніть увагу, що відбувається автоматичний виклик деструкторів всіх трьох класів по порядку від самого далекого до першого в ланцюжку наслідування.

```
class First
{
    ~First()
    {
        System.Diagnostics.Trace.WriteLine("First's destructor is called.");
    }
}

class Second : First
{
    ~Second()
    {
        System.Diagnostics.Trace.WriteLine("Second's destructor is called.");
    }
}
```

```

class Third : Second
{
    ~Third()
    {
        System.Diagnostics.Trace.WriteLine("Third's destructor is called.");
    }
}

class TestDestructors
{
    static void Main()
    {
        Third t = new Third();
    }
}

/* Output (to VS Output Window):
   Third's destructor is called.
   Second's destructor is called.
   First's destructor is called.
*/

```

1.4.10 Інтерфейси (interface)

Інтерфейс містить тільки описи методів, властивостей, подій або індексаторів. Реалізація членів виконується в класі або в структурі, що реалізує інтерфейс, як показано в наступному прикладі:

```

interface ISampleInterface
{
    void SampleMethod();
}

class ImplementationClass : ISampleInterface
{
    // Explicit interface member implementation:
    void ISampleInterface.SampleMethod()
    {
        // Method implementation.
    }

    static void Main()
    {
        // Declare an interface instance.
        ISampleInterface obj = new ImplementationClass();

        // Call the member.
        obj.SampleMethod();
    }
}

```

```
}  
}
```

Інтерфейс може бути членом простору імен або класу і містити описи наступних членів:

- методи
- властивості
- індексатори
- події

У наступному прикладі демонструється реалізація інтерфейсу. У цьому прикладі інтерфейс містить оголошення властивостей, а клас містить реалізацію. Інтерфейс здатний наслідувати від одного або декількох базових інтерфейсів. Якщо в списку базових типів міститься базовий клас і інтерфейси, то базовий клас повинен стояти в списку на першому місці. Клас, який реалізує інтерфейс, може явно реалізовувати члени цього інтерфейсу. Явно реалізований член можна викликати тільки через екземпляр інтерфейсу, але не через екземпляр класу.

```
interface IPoint  
{  
    // Property signatures:  
    int x { get; set; }  
    int y { get; set; }  
}  
  
class Point : IPoint  
{  
    // Fields:  
    private int _x;  
    private int _y;  
  
    // Constructor:  
    public Point(int x, int y)  
    {  
        _x = x;  
        _y = y;  
    }  
  
    // Property implementation:  
    public int x {  
        get  
        {  
            return _x;  
        }  
  
        set  
        {  
            _x = value;  
        }  
    }  
}
```

```

public int y
{
    get
    {
        return _y;
    }
    set
    {
        _y = value;
    }
}
}

class MainClass
{
    static void PrintPoint(IPoint p)
    {
        Console.WriteLine("x={0}, y={1}", p.x, p.y);
    }

    static void Main()
    {
        Point p = new Point(2, 3);
        Console.Write("My Point: ");
        PrintPoint(p);
    }
}
// Output: My Point: x=2, y=3

```

1.4.11 Ключове слово SET

Ключове слово **set** визначає метод доступу у властивості або індексатора, який встановлює значення даних властивості елемента індексатора. Далі наведено приклад методу доступу **set** для властивості Seconds.

```

class TimePeriod
{
    private double _seconds;
    public double Seconds
    {
        get { return _seconds; }
        set { _seconds = value; }
    }
}

```

Далі наведено приклад методу доступу **set** в властивості яка автоматично реалізується:

```

class TimePeriod2

```

```
{
    public double Hours { get; set; }
}
```

1.4.12 Ключове слово GET

Ключове слово **get** визначає метод доступу у властивості або індексатора, який отримує значення даних властивості елемента індексатора.

```
class TimePeriod
{
    private double _seconds;
    public double Seconds
    {
        get { return _seconds; }
        set { _seconds = value; }
    }
}
```

Далі наведено приклад методу доступу **get** в автоматично реалізованій властивості.

```
class TimePeriod2
{
    public double Hours { get; set; }
}
```

1.4.13 Асиметричні методи доступу

Частини властивості **get** і **set** або індексатора називаються методами доступу. За замовчуванням ці методи доступу володіють тією ж видимістю або рівнем доступу, властивим властивості або індексатора, якому вони належать. Однак іноді має сенс обмежити доступ до одного з цих методів доступу. Як правило, для цього обмежується доступність методу доступу **set** і зберігається відкрита доступність методу доступу **get**. Наприклад:

```
private string name = "Hello";

public string Name
{
    get
    {
        return name;
    }
    protected set
    {
        name = value;
    }
}
```

У цьому прикладі властивість з ім'ям **Name** визначає метод доступу **get** і **set**. Метод доступу **get** отримує рівень доступності самого властивості - в цьому

випадку це **public**, а метод доступу **set** обмежується явним чином шляхом застосування модифікатора доступу **protected** до самого методу доступу.

Використання модифікаторів доступу для властивостей або індексаторів підпорядковується таким умовам:

- Модифікатори доступу не можна використовувати в інтерфейсі або в явній реалізації члена інтерфейсу.
- Модифікатори доступу можна використовувати тільки в тому випадку, якщо як властивість, так і індексатор мають модифікатори доступу **set** і **get**. У цьому випадку, модифікатор дозволено в одному з двох методів доступу.
- Якщо властивість або індексатор має модифікатор **override**, модифікатор доступу повинен відповідати методу доступу перевизначеного методу доступу, якщо він існує.
- Рівень доступності в методі доступу повинен бути більш обмежуючим, ніж рівень доступності в самому властивості або індексатора.

При перевизначенні властивості або індексатора перевизначені методи доступу повинні бути доступні перевизначеному коду. Крім того, рівень доступності властивості / індексатора, а також методів доступу повинен перехоплювати відповідну властивість / індексатор і методи доступу. Наприклад:

```
public class Parent
{
    public virtual int TestProperty
    {
        // Notice the accessor accessibility level.
        protected set { }

        // No access modifier is used here.
        get { return 0; }
    }
}

public class Kid : Parent
{
    public override int TestProperty
    {
        // Use the same accessibility level as in the overridden accessor.
        protected set { }

        // Cannot use access modifier here.
        get { return 0; }
    }
}
```

При використанні методу доступу для реалізації інтерфейсу, метод доступу не може мати модифікатор доступу. Однак, якщо інтерфейс реалізується за допомогою одного методу доступу, такого як **get**, інший метод доступу може мати модифікатор доступу, так як показано у наступному прикладі:

```

public interface ISomeInterface
{
    int TestProperty
    {
        // No access modifier allowed here
        // because this is an interface.
        get;
    }
}

public class TestClass : ISomeInterface
{
    public int TestProperty
    {
        // Cannot use access modifier here because
        // this is an interface implementation.
        get { return 10; }

        // Interface property does not have set accessor,
        // so access modifier is allowed.
        protected set {}
    }
}

```

При використанні модифікатора доступу в методі доступу, цей модифікатор визначає домен доступності методу доступу. Якщо модифікатор доступу в методі доступу не використовується, домен доступності методу доступу визначається рівнем доступності властивості або індексатора.

У наступному прикладі містяться три класи: `BaseClass`, `DerivedClass` і `MainClass`. У `BaseClass`, `Name` і `Id` для обох класів є дві властивості. Приклад демонструє, як властивість `Id` в `DerivedClass` можна приховати за допомогою властивості `Id` в `BaseClass`, якщо використовується обмежувальний модифікатор доступу, такий як **protected** або **private**. Таким чином, при присвоєнні значень цій властивості, замість цього викликається властивість `BaseClass`. Заміна модифікатора доступу модифікатором **public** зробить властивість доступним. Приклад також демонструє, що обмежує модифікатор доступу, такий як **private** або **protected**, в методі доступу **set** властивості `Name` в `DerivedClass` забороняє доступ до методу доступу і створює помилку при його присвоєнні.

```

namespace AccessibilityDomainNamespace
{
    public class T1
    {
        public static int publicInt;
        internal static int internalInt;
        private static int privateInt = 0; // CS0414

        public class M1
        {

```

```

    public static int publicInt;
    internal static int internalInt;
    private static int privateInt = 0; // CS0414
}

private class M2
{
    public static int publicInt = 0;
    internal static int internalInt = 0;
    private static int privateInt = 0; // CS0414
}

class MainClass
{
    static void Main()
    {
        // Access is unlimited:
        T1.publicInt = 1;

        // Accessible only in current assembly:
        T1.internalInt = 2;

        // Error: inaccessible outside T1:
        // T1.myPrivateInt = 3;

        // Access is unlimited:
        T1.M1.publicInt = 1;

        // Accessible only in current assembly:
        T1.M1.internalInt = 2;

        // Error: inaccessible outside M1:
        // T1.M1.myPrivateInt = 3;

        // Error: inaccessible outside T1:
        // T1.M2.myPublicInt = 1;

        // Error: inaccessible outside T1:
        // T1.M2.myInternalInt = 2;

        // Error: inaccessible outside M2:
        // T1.M2.myPrivateInt = 3;
    }
}

```

1.4.14 Індексатори

Індексатори є синтаксичним інструментом, що дозволяють створювати клас, структуру або інтерфейс, доступ до якого клієнтські програми отримують,

як до масиву. Найчастіше індексатори реалізуються в типах, головна мета яких - інкапсуляція внутрішньої колекції або масиву. Наприклад, припустимо, що є клас з ім'ям "TempRecord", що представляє набір температур за шкалою Фаренгейта, отриманих в 10 різних моментів протягом 24 годин. Клас містить масив з ім'ям "temps" типу " float ", що представляє температури, і DateTime, що представляє дату реєстрації температур. Шляхом впровадження в цей клас індексатора клієнти отримають доступ до температур в екземплярі TempRecord за допомогою виразу float temp = tr[4], а не float temp = tr.temps[4]. Використання індексатора не тільки спрощує синтаксис для клієнтських програм, але і робить клас і його призначення інтуїтивно зрозумілими для інших розробників.

Щоб оголосити індексатор для класу або структури, можна використати ключове слово **this**, як показано в наступному прикладі:

```
public int this[int index] // Indexer declaration
{
    // get and set accessors
}
```

Тип індексатора і типи його параметрів повинні мати принаймні такий же рівень доступності, як і сам індексатор.

Опис індексатора складається з кількості і типів його формальних параметрів. У опис не включається тип індексатора або імена формальних параметрів. Якщо в одному класі оголошено кілька індексаторів, у них повинні бути різні описи.

Значення індексатора не класифікується як змінна, тому не допускається передача значення індексатора як параметра **ref** або **out**.

Щоб надати індексатора ім'я, яке можна використовувати в інших мовах, використовуйте в оголошенні атрибут **name**.

Приклад :

```
[System.Runtime.CompilerServices.IndexerName("TheItem")]
public int this [int index] // Indexer declaration
{
}
```

Цей індексатор матиме ім'я **TheItem**. Якщо атрибут імені не надано, використовується ім'я за замовчуванням **Item**.

У наступному прикладі показано, як оголосити закрите поле масиву temps і індексатор. Індексатор забезпечує прямий доступ до примірника tempRecord[i]. В якості альтернативи застосуванню індексатора можна оголосити масив як член типу public здійснювати прямий доступ до його членів tempRecord.temps[i].

Зверніть увагу, що при обчисленні доступу індексатора, наприклад, в інструкції Console.WriteLine викликається метод доступу get. Таким чином, якщо не існує методу доступу get, відбувається помилка часу компіляції.

```
class TempRecord
{
```

```

// Array of temperature values
private float[] temps = new float[10] { 56.2F, 56.7F, 56.5F, 56.9F, 58.8F, 61.3F, 65.9F, 62.1F, 59.2F,
57.5F };

// To enable client code to validate input
// when accessing your indexer.
public int Length
{
    get { return temps.Length; }
}

// Indexer declaration.
// If index is out of range, the temps array will throw the exception.
public float this[int index]
{
    get
    {
        return temps[index];
    }

    set
    {
        temps[index] = value;
    }
}
}

class MainClass
{
    static void Main()
    {
        TempRecord tempRecord = new TempRecord();
        // Use the indexer's set accessor
        tempRecord[3] = 58.3F;
        tempRecord[5] = 60.1F;

        // Use the indexer's get accessor
        for (int i = 0; i < 10; i++)
        {
            System.Console.WriteLine("Element #{0} = {1}", i, tempRecord[i]);
        }

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}

```

1.4.15 Розділювані типи

Визначення розділюваних типів дозволяють розділювати визначення для класів, структур і інтерфейсів на кілька файлів. Наприклад:

В файлі File1.cs:

```
namespace PC
{
    partial class A
    {
        int num = 0;
        void MethodA() {}
        partial void MethodC();
    }
}
```

У файлі File2.cs продовжуємо оголошення:

```
namespace PC
{
    partial class A
    {
        void MethodB(){}
        partial void MethodC() {}
    }
}
```

Поділ типу класу, структури або інтерфейсу на декілька файлів може стати в нагоді при роботі з великими проектами або з автоматично створеним кодом, наприклад з кодом Конструктор Windows Forms. Розділюваний тип може містити розділювані методи.

1.4.16 Розділювані методи

Опис розділюваного методу визначається в одній частині розділюваного типу, а реалізація визначається в іншій частині типу. Колективні методи дозволяють конструкторам класів надавати захоплювачі методів (аналогічні обробникам подій), які при необхідності можуть бути реалізовані розробниками. Якщо реалізація не здійснена розробником програми, компілятор видаляє опис під час компіляції. До розділюваних методів застосовуються такі умови:

- Описи в обох частинах розділюваних типів повинні збігатися.
- Метод повинен повертати значення void.
- Модифікатори або атрибути неприпустимі. Колективні методи (неявно) є закритими.

У наведеному нижче прикладі показано розділюваний метод, визначений у двох частинах розділюваного класу.

```

namespace PM
{
    partial class A
    {
        partial void OnSomethingHappened(string s);
    }

    // Наступна частина може бути в іншому файлі:
    partial class A
    {
        // Comment out this method and the program
        // will still compile.
        partial void OnSomethingHappened(String s)
        {
            Console.WriteLine("Something happened: {0}", s);
        }
    }
}

```

Контрольні питання.

1. Чим клас відрізняється від об'єкта?
2. Як здійснюється наслідування класів?
3. Для чого використовується модифікатор “abstract”?
4. Для чого використовується метод “override”?
5. Для чого використовується ключове слово “virtual”?
6. Реалізація поліморфізму в C#.
7. Застосування віртуальних методів.
8. «Запечатані» класи.
9. Використання деструкторів.
10. Відмінності інтерфейсів від класів.
11. Використання ключових слів “GET/SET”.
12. Як використовуються індексатори?
13. Як можуть бути використані розділювані типи і методи?

1.5 Делегати

1.5.1 Основні принципи функціонування делегатів

Делегат - це тип, який визначає сигнатуру методу. При створенні екземпляра делегата можна пов'язати цей екземпляр з будь-яким методом з сумісною сигнатурою. Метод можна запустити (або викликати) за допомогою екземпляра делегата.

Делегати використовуються для передачі методів в якості аргументів до інших методів. Обробники подій - це не що інше, як методи, що викликаються за допомогою делегатів. Ви створюєте свій метод, і клас, як наприклад, елемент керування Windows, може викликати ваш метод при виникненні певної події. У наступному прикладі показано оголошення делегата:

```
public delegate int PerformCalculation(int x, int y);
```

Делегату можна призначити будь-який метод з будь-якого доступного класу або структури, відповідний сигнатурі делегата, яка складається з типу значення, що повертається і параметрів. Цей метод має бути статичним методом або методом екземпляра. Це дозволяє програмно змінювати виклики методу, а також включати новий код в існуючі класи. Якщо сигнатура делегата відома, то можна призначити власний метод.

У контексті перевантаження методу сигнатура методу не містить значення, що повертається. Проте в контексті делегатів сигнатура методу містить повертається значення. Іншими словами, метод повинен мати те ж значення, що повертається, що і делегат.

Завдяки можливості посилатися на метод як на параметр делегатів оптимально підходять для завдання функцій зворотного виклику. Наприклад, посилання на метод, що порівнює два значення, які можна передати в якості аргументу алгоритмом сортування. Оскільки код порівняння знаходиться в окремій процедурі, алгоритм сортування може бути написаний більш загальним чином.

Делегат - це тип, який безпечно інкапсулює метод, тобто його дія схожа з покажчиком на функції в C і C++. На відміну від покажчиків функцій в C делегати об'єктно-орієнтовані, строго типізовані і безпечні. Тип делегата задається його ім'ям. У наступному прикладі оголошується делегат з ім'ям Del, який може інкапсулювати метод, що використовує як аргумент значення string і повертає значення void:

```
public delegate void Del(string message);
```

Об'єкт делегата зазвичай створюється зазначенням імені методу, для якого делегат буде служити оболонкою, або за допомогою анонімного методу. Після створення екземпляра делегата виклик методу, виконаний у делегатіві, передається делегатом в цей метод. Параметри, передані делегату викликаючим об'єктом, передаються в метод, а значення яке повертається методом (при його наявності) повертається делегатом в викликаючий об'єкт. Ця процедура називається викликом делегата. Делегат, для якого створено екземпляр, можна викликати так, ніби це був метод, для якого створюється оболонка. Наприклад:

```
// Create a method for a delegate.
public static void DelegateMethod(string message)
{
    System.Console.WriteLine(message);
}

// Instantiate the delegate.
Del handler = DelegateMethod;

// Call the delegate.
handler("Hello World");
```

Типи делегатів є похідними від класу `Delegate` в платформі `.NET Framework`. Типи делегатів є запечатаними - від них не можна успадковувати, а від `Delegate` не можна створювати похідні користувача класи. Оскільки створений екземпляр делегата є об'єктом, його можна передавати як параметр або призначати властивості. Це дозволяє методам приймати делегат в якості параметра або викликати делегат згодом. Ця процедура називається асинхронним зворотним викликом і зазвичай використовується для повідомлення викликаючого об'єкта про завершення тривалої операції. Коли делегат використовується таким чином, коду, що використовує делегат, не потрібні відомості про реалізацію використовуваного методу. Дані функціональні можливості схожі на можливості, що надаються інтерфейсами інкапсуляції.

Зворотний виклик також часто використовується для завдання спеціального методу порівняння та передачі цього делегата в метод сортування. Це дозволяє зробити код викликає об'єкта частиною алгоритму сортування. У наступному прикладі метод використовує тип `Del` як параметр:

```
public void MethodWithCallback(int param1, int param2, Del callback)
{
    callback("The number is: " + (param1 + param2).ToString());
}
```

Потім можна передати створений раніше делегат в даний метод:

```
MethodWithCallback(1, 2, handler);
```

і отримати наступні вихідні дані у вікні консолі :

The number is : 3

При використанні делегата в якості абстракції методом `MethodWithCallback` не потрібно виконувати безпосередній виклик консолі, тобто його можна створювати без урахування консолі. Метод `MethodWithCallback` просто готує рядок і передає її в інший метод. Це дуже зручно, так як делегованих метод може використовувати будь-яку кількість параметрів.

Якщо делегат створений для забезпечення оболонки для методу примірнику, то цей делегат посилається і на екземпляр, і на метод. Делегат не має відомостей про тип екземпляра, крім отриманих з методу, для якого він забезпечує оболонку, тому делегат може посилатися на будь-який тип об'єкта, якщо для цього об'єкта є метод, відповідний сигнатурі делегата. Якщо делегат створений для забезпечення оболонки для статичного методу, то цей делегат посилається лише на метод. Розглянемо наступні оголошення:

```
public class MethodClass
{
    public void Method1(string message) { }
```

```
public void Method2(string message) { }  
}
```

Разом з розглянутим раніше статичним методом `DelegateMethod` у нас є три методи, для яких можна створити оболонку за допомогою примірника `Del`.

При виклику делегат може викликати відразу кілька методів. Це називається багатоадресністю. Щоб додати до списку методів делегата (список виклику) додатковий метод, необхідно просто додати два делегати з допомогою оператора додавання або призначення додавання ("`+`" або "`+=`"). наприклад:

```
MethodClass obj = new MethodClass();  
Del d1 = obj.Method1;  
Del d2 = obj.Method2;  
Del d3 = DelegateMethod;
```

```
//Both types of assignment are valid.  
Del allMethodsDelegate = d1 + d2;  
allMethodsDelegate += d3;
```

На даному етапі список виклику делегата `allMethodsDelegate` містить три методи - `Method1` , `Method2` і `DelegateMethod` . Три вихідних делегата `d1` , `d2` і `d3` залишаються без змін. При виклику `allMethodsDelegate` всі три методи викликаються по порядку. Якщо делегат використовує параметр, переданий по покажчику , цей покажчик передається після кожного з трьох методів , а всі зміни одного з методів стають видимі в наступному методі . Якщо будь-який з методів викликає неперехвачений виняток, то цей виняток передається в об'єкт що викликає делегат, а наступні методи у списку виклику не викликаються . Якщо делегат має значення, що повертається і/або вихідні параметри , він повертає повертається значення і параметри останнього викликаного методу. Для видалення методу зі списку виклику слід використовувати оператор декремента або призначення декремента ("`-`" або "`-=`") . наприклад:

```
//remove Method1  
allMethodsDelegate -= d1;  
  
// copy AllMethodsDelegate while removing d2  
Del oneMethodDelegate = allMethodsDelegate - d2;
```

Оскільки типи делегата є похідними від `System.Delegate`, в делегатові можна викликати методи і властивості, що задаються даним класом. Наприклад, щоб визначити число методів в списку виклику делегата, можна використовувати код:

```
int invocationCount = d1.GetInvocationList().GetLength(0);
```

Делегати, в списку виклику яких знаходяться кілька методів, є похідним від `MulticastDelegate` , що є підкласом `System.Delegate` . Наведений вище код

працює в будь-якому з випадків , оскільки обидва класи підтримують GetInvocationList .

Групові делегати часто використовуються при обробці подій. Об'єкти джерел подій відправляють повідомлення об'єктам одержувачів, зареєстрованим для отримання даної події. Щоб зареєструватися для отримання події, об'єкт одержувача створює метод, призначений для обробки цієї події, потім створює делегат для цього методу і передає його на джерело події. Коли відбувається подія, джерело викликає делегата. Після цього делегат викликає в об'єкті одержувача метод обробки події , надавши йому дані події . Тип делегата для даної події задається джерелом події. Призначення порівняння делегатів двох різних типів під час компіляції, викличе помилку компіляції . Якщо примірники делегата статично відносяться до типу System.Delegate, то порівняння припустиме, але поверне значення false під час виконання. Наприклад:

```
delegate void Delegate1();
delegate void Delegate2();

static void method(Delegate1 d, Delegate2 e, System.Delegate f)
{
    // Compile-time error.
    //Console.WriteLine(d == e);

    // OK at compile-time. False if the run-time type of f
    // is not the same as that of d.
    System.Console.WriteLine(d == f);
}
```

1.5.2 Використання варіативності в делегатах

При присвоєнні методу делегату коваріація і контраваріація забезпечують гнучкість зіставлення типу делегата з сигнатурою методу. Коваріація дозволяє методу мати тип значення, ступінь успадкування якого більша, ніж вказано в делегаті. Контраваріація дозволяє використовувати метод з типами параметрів, ступінь успадкування яких менше, ніж у типу делегата:

Коваріація. В даному прикладі демонструється використання делегатів з методами, типи повертаються значень яких є похідними від типу значення, що повертається в сигнатурі делегата. Тип даних, що повертається DogsHandler, є типом Dogs, похідним від заданого в делегаті типу Mammals:

```
class Mammals{}
class Dogs : Mammals{}

class Program
{
    // Define the delegate.
    public delegate Mammals HandlerMethod();

    public static Mammals MammalsHandler()
```



```

{
    return null;
}

public static Dogs DogsHandler()
{
    return null;
}

static void Test()
{
    HandlerMethod handlerMammals = MammalsHandler;
    // Covariance enables this assignment.
    HandlerMethod handlerDogs = DogsHandler;
}
}

```

Контрваріація. У даному прикладі демонструється використання делегатів з методами, параметри типу яких є базовими типами з типом параметра сигнатури делегата. Контраваріація дозволяє використовувати один обробник подій замість декількох. Наприклад, можна створити обробник подій, що приймає параметр введення EventArgs, і використовувати його з подією Button.MouseClick, яке відправляє тип MouseEventArgs як параметр, а також з подією TextBox.KeyDown, яке відправляє параметр KeyEventArgs.

```

// Event handler that accepts a parameter of the EventArgs type.
private void MultiHandler(object sender, EventArgs e)
{
    label1.Text = System.DateTime.Now.ToString();
}

public Form1()
{
    InitializeComponent();

    // You can use a method that has an EventArgs parameter,
    // although the event expects the KeyEventArgs parameter.
    this.button1.KeyDown += this.MultiHandler;

    // You can use the same method
    // for an event that expects the MouseEventArgs parameter.
    this.button1.MouseClick += this.MultiHandler;
}

```

Контрольні питання.

1. Принципи функціонування делегатів.
2. Принципи коваріації та контрваріації при використанні делегатів.

1.6 Лямбда-вирази

1.6.1 Основні положення

Лямбда-вираз - це анонімна функція, яку можна використовувати для створення типів делегатів або дерев виразів. За допомогою лямбда-виразів можна написати локальні функції, які потім можна передавати в інші функції в якості аргументів чи повертати з них у якості значення. Лямбда-вирази особливо корисні для написання виразів запитів LINQ.

Для створення лямбда-вирази можна визначити вхідні параметри (якщо такі є) зліва лямбда-оператора `=>` і помістити блок висловів чи виписки на інший бік. Наприклад, лямбда-вираз `x => x * x` приймає параметр з ім'ям `x` і повертає значення `x`, зведене до квадрату. Можна призначити цей вираз до типу делегата, як показано в наступному прикладі:

```
using System.Linq.Expressions;
```

```
namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            Expression<del> myET = x => x * x;
        }
    }
}
```

Оператор `=>` має той же пріоритет, що і оператор присвоювання (`=`) і є право-асоціативним. Лямбда-оператори використовуються в запитах LINQ з синтаксисом на основі методів в якості аргументів методів стандартних операторів запиту, таких як `Where`.

При використанні синтаксису на основі методів для виклику методу `Where` класу `Enumerable` (як це відбувається в LINQ to Objects і LINQ to XML) параметром є тип делегата `System.Func<T, TResult>`. Лямбда-вираз - це найбільш зручний спосіб створення такого делегата. При виклику такого ж методу, наприклад, у класі `System.Linq.Queryable` (як це відбувається в LINQ to SQL) типом параметра буде дерево виразів `System.Linq.Expressions.Expression<Func>`, де `Func` - це будь-які делегати `Func` з числом вхідних параметрів не більше шістнадцяти. При цьому лямбда-вирази являють собою найшвидший спосіб побудови такого дерева виразів. Лямбда-вирази дозволяють викликам `Where` виглядати однаково, хоча, насправді, об'єкти, створені з лямбда-виразів, будуть мати різні типи.

Зверніть увагу: у наведеному вище прикладі сигнатура делегата має один неявний вхідний параметр типу `int` і повертає значення типу `int`. Лямбда-вираз можна перетворити в делегат відповідного типу, оскільки він також має один параметр введення (`x`) і повертає значення, яке компілятор може неявно

перетворити в тип `int`. Делегат, що викликається за допомогою параметра введення 5, повертає результат 25.

Лямбда-вирази не дозволені в лівій частині оператора `is` або `as`. Всі обмеження, застосовувані до анонімних методів, застосовуються також і до лямбда-виразів.

1.6.2 Лямбда-вирази

Лямбда-вирази широко використовуються при створенні таких конструкцій як **дерева виразів**. Вирази-лямбда повертають результат і мають наступну основну форму:

```
(input parameters) => expression
```

Якщо лямбда має тільки один вхідний параметр, дужки можна не ставити, у всіх інших випадках вони обов'язкові. Якщо вхідних параметрів два і більше, то вони розділяються комами і розміщуються в дужках:

```
(x, y) => x == y
```

Іноді компілятору буває важко або навіть неможливо вивести типи вхідних параметрів. У цьому випадку типи можна вказати в явному вигляді, як показано в наступному прикладі:

```
(int x, string s) => s.Length > x
```

Відсутність вхідних параметрів задається порожніми дужками:

```
() => SomeMethod()
```

Зверніть увагу на попередній приклад: тіло лямбда-виразу може складатися з виклику методу. Однак при створенні дерев виразів, які будуть використовуватися в іншому домені, наприклад в SQL Server, в лямбда-виразах не слід використовувати виклики методів. Ці методи не мають змістового значення поза контекстом середовища виконання .NET.

1.6.3 Лямбда-оператори

Лямбда-оператор (або операторна лямбда) нагадує лямбда-вираз, за винятком того, що оператор береться у фігурні дужки:

```
(input parameters) => {statement;}
```

Тіло лямбда-оператора може складатися з будь-якої кількості операторів; проте на практиці зазвичай використовується не більше двох-трьох:

```
delegate void TestDelegate(string s);
```

```
...
```

```
TestDelegate myDel = n => { string s = n + " " + "World"; Console.WriteLine(s); };
myDel("Hello");
```

Лямбда-оператори, як і анонімні методи, не можуть використовуватися для створення дерев виразів.

1.6.4 Лямбди зі стандартними операторами запитів

Багато стандартних операторів запитів мають вхідний параметр, тип якого належить сімейству універсальних методів-делегатів `Func <T, TResult>`. Делегати `Func <T, TResult>` використовують параметри типу для визначення кількості та типу вхідних параметрів, а також тип значення делегата. Делегати `Func` дуже корисні для інкапсуляції користувальницьких виразів, які застосовуються до кожного елементу в наборі вихідних даних. Як приклад розглянемо наступний тип делегату:

```
public delegate TResult Func<TArg0, TResult>(TArg0 arg0)
```

Можна створити екземпляр цього делегата як `Func <int,bool> myFunc`, де `int` - тип вхідного параметра, а `bool` - тип значення. Значення, що повертається завжди вказується в останньому параметрі типу. `Func <int, string, bool>` визначає делегат з двома вхідними параметрами, типи яких - `int` і `string`, і типом значення, що повертається `bool`. Наступний делегат `Func` при виклику повертає значення `true` або `false`, яке показує, чи рівний вхідний параметр 5:

```
Func<int, bool> myFunc = x => x == 5;
bool result = myFunc(4); // returns false of course
```

Також лямбда-вирази можна використовувати, коли аргумент має тип `Expression <Func>`, наприклад в стандартних операторах запиту, які визначені у `System.Linq.Queryable`. При завданні аргументу типу `Expression <Func>` лямбда-вираз компілюється в дерево виразу. Приклад використання стандартного оператора запиту, методу `Count`, показаний нижче:

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
int oddNumbers = numbers.Count(n => n % 2 == 1);
```

Компілятор може вивести тип вхідного параметра; а також його можна визначити явно. Даний конкретний лямбда-вираз підраховує кількість цілих чисел (`n`), які при діленні на два дають залишок 1. Наступний метод створює послідовність, що містить всі елементи масиву `numbers`, які знаходяться зліва від 6, оскільки це перше число в послідовності, що не задовольняє умові:

```
var firstNumbersLessThan6 = numbers.TakeWhile(n => n < 6);
```

У цьому прикладі показано, як задати кілька вхідних параметрів шляхом розміщення їх в дужки. Цей метод повертає всі елементи в масиві чисел до того

числа, величина якого менше номера його позиції. Не слід плутати лямбда-оператор (\Rightarrow) з оператором "більше або дорівнює" ($>=$).

```
var firstSmallNumbers = numbers.TakeWhile((n, index) => n >= index);
```

1.6.5 Вивід типу в лямбда-виразах

При написанні лямбда-виразів звичайно не потрібно вказувати тип вхідних параметрів, оскільки компілятор може вивести цей тип на основі тіла лямбда-виразів, типу, що лежить в основі делегата та інших факторів. Для більшості стандартних операторів запиту перший вхідний параметр має той же тип, що і елементи у вихідній послідовності. Тому, якщо виробляється запит до `IEnumerable <Customer>`, то змінна вхідного параметра розцінюється як об'єкт типу `Customer`, а це означає, що у вас є доступ до його методів і властивостей.

```
customers.Where(c => c.City == "London");
```

Використовуються такі основні правила для лямбда-виразів:

- Лямбда-вираз повинен містити те ж число параметрів, що і тип делегата.
- Кожен вхідний параметр в лямбда-виразі повинен бути неявно перетворюваний до відповідного параметра делегата.
- Значення, що повертається лямбда-виразом (якщо таке є) має бути неявно перетворюваний в тип делегата який повертається.

Зверніть увагу: лямбда-вирази самі по собі не мають типу, оскільки система загальних типів не має вбудованої концепції "лямбда-вирази". Однак іноді буває зручно оперувати неформальним поняттям "типу" лямбда-вирази. При цьому під типом розуміється тип делегата або тип `Expression`, в який перетворюється лямбда-вираз.

1.6.6 Область дії змінної в лямбда-виразах

Лямбда-вирази можуть посилатися на зовнішні змінні, що потрапляють в область дії методу або типу, в якому визначено цей лямбда-вираз. Змінні, захоплені таким способом, зберігаються для використання в лямбда-виразах навіть у тому випадку, якщо ці змінні інакшим чином потрапляють за межі області дії і знищуються збирачем сміття. Зовнішній змінній має бути присвоєно певне значення, перш ніж вона зможе використовуватися в лямбда-виразах. Наступний приклад демонструє застосування цих правил:

```
delegate bool D();  
delegate bool D2(int i);
```

```
class Test  
{  
    D del;  
    D2 del2;
```

```

public void TestMethod(int input)
{
    int j = 0;
    // Initialize the delegates with lambda expressions.
    // Note access to 2 outer variables.
    // del will be invoked within this method.
    del = () => { j = 10; return j > input; };

    // del2 will be invoked after TestMethod goes out of scope.
    del2 = (x) => { return x == j; };

    // Demonstrate value of j:
    // Output: j = 0
    // The delegate has not been invoked yet.
    Console.WriteLine("j = {0}", j);    // Invoke the delegate.
    bool boolResult = del();

    // Output: j = 10 b = True
    Console.WriteLine("j = {0}. b = {1}", j, boolResult);
}

static void Main()
{
    Test test = new Test();
    test.TestMethod(5);

    // Prove that del2 still has a copy of
    // local variable j from TestMethod.
    bool result = test.del2(10);

    // Output: True
    Console.WriteLine(result);

    Console.ReadKey();
}
}

```

Наступні правила застосовні до області дії змінної в лямбда-виразах.

- Захоплена змінна не буде знищена збирачем сміття до тих пір, поки делегат, який на неї посилається, не вийде за межі області видимості.
- Змінна, оголошена в Лямбда-виразі, невидима у зовнішньому методі.
- Лямбда-вираз не може безпосередньо захоплювати параметри **ref** або **out** з методу, який їх включає.
- Оператор **Return** в лямбда-виразі не приводить до повернення (завершення) методу, в якому оголошено/викликано лямбда-вираз .

Лямбда-вираз не може містити оператор **goto**, оператор **break** або оператор **continue**, цільовий об'єкт якого знаходиться поза тілом або міститься в тілі анонімної функції.

Контрольні питання.

1. Необхідність використання лямбда-виразів.
2. Використання лямбда-виразів.
3. Використання лямбда-операторів.
4. Область дії змінних при використанні лямбда-виразів.

1.7 Регулярні вирази

Регулярні вирази - це частина невеликої технологічної області, неймовірно широко використовуваної в величезному діапазоні програм. Регулярні вирази можна уявити собі як міні-мова програмування, що має одне специфічне призначення: знаходити «підрядки» у великих рядкових виразах.

Це не нова технологія, спочатку вона з'явилася в середовищі UNIX і зазвичай використовується в мові програмування Perl. Розробники з Microsoft перенесли її в Windows, де донедавна ця технологія застосовувалася в основному зі сценарними мовами. Однак тепер регулярні вирази підтримуються безліччю класів .NET з простору імен System.Text.RegularExpressions. Випадки застосування регулярних виразів можна зустріти в багатьох частинах середовища .NET Framework. Зокрема, ви знайдете їх в серверних елементах управління перевіркою ASP.NET.

1.7.1 Елементи регулярних виразів

Мова регулярних виразів призначений спеціально для обробки рядків. Він включає два засоби:

- Набір керуючих кодів для ідентифікації специфічних типів символів
- Система для групування частин підрядків і проміжних результатів таких дій

За допомогою регулярних виразів можна виконувати досить складні і високорівневі дії над рядками:

- Ідентифікувати (і можливо, позначати до видалення) всі повторювані слова в рядку
- Зробити заголовними перші літери всіх слів
- Перетворити перші літери всіх слів довші трьох символів в заголовні
- Забезпечити правильну капіталізацію речень
- Виділити різні елементи в URI (наприклад, маючи <http://www.xeniumaudio.com>, виділити протокол, ім'я комп'ютера, ім'я файлу і т.д.)

Головною перевагою регулярних виразів є використання метасимволів – спеціальних символів, що задають команди, а також керуючі послідовності, які працюють подібно керуючим послідовностям C#. Це символи, упереджені знаком зворотного слеша (\) і мають спеціальне призначення.

В наступній таблиці спеціальні метасимволи регулярних виразів C# згруповані за змістом:

Таблиця 1.1 – Символи управління

Символ	Значення	Приклад	Примітка
Класи символів			
[...]	Будь-який з символів, зазначених в дужках	[a-z]	У вихідній рядку може бути будь-який символ англійського алфавіту в нижньому регістрі
[^...]	Будь-який з символів, не зазначених в дужках	[^0-9]	У вихідній рядку може бути будь-який символ окрім цифр
.	Будь символ, крім переводу рядка або іншого роздільника Unicode-рядка		
\w	Будь-який текстовий символ, який не є пропуском, знаком табуляції і т.п.		
\W	Будь-який символ, який не є текстовим символом		
\s	Будь-який пробільний символ з набору Unicode		
\S	Будь-який непробільний символ з набору Unicode.		

	Зверніть увагу, що символи \w і \S - це не одне і те ж		
\d	Будь-які ASCII-цифри. Еквівалентно [0-9]		
\D	Будь-який символ, відмінний від ASCII-цифр. Еквівалентно [^ 0-9]		
Символи повторення			
{n,m}	Відповідає попередньому шаблоном, повторення не менше n і не більше m разів	s{2,4}	"Press", "ssl", "progressss"
{n,}	Відповідає попередньому шаблоном, повторення n або більше разів	s{1,}	"ssl"
{n}	Відповідає в точності n екземплярам попереднього шаблону	s{2}	"Press", "ssl", але не "progressss"
?	Відповідає нулю або одному примірнику попереднього шаблону; попередній шаблон є необов'язковим	Еквівалентно {0,1}	
+	Відповідає одному або більше екземплярам попереднього шаблону	Еквівалентно {1,}	
*	Відповідає нулю або більше екземплярам попереднього шаблону	Еквівалентно {0,}	

Символи регулярних виразів вибору			
	Відповідає або підвиразу зліва, або підвиразу справа (аналог логічної операції АБО).		
(...)	Групування. Групує елементи в єдине ціле, яке може використовуватися з символами *, +, ?, і т.п. Також запам'ятовує символи, відповідні цій групі для використання в наступних посиланнях.		
(?:...)	Тільки групування. Групує елементи в єдине ціле, але не запам'ятовує символи, відповідні цій групі.		
Якірні символи регулярних виразів			
^	Відповідає початку рядкового виразу або початку рядка при багаторядковому пошуку.	^Hello	"Hello, world", але не "Ok, Hello world" т.як. в цьому рядку слово "Hello" не початкове
\$	Відповідає кінцю рядкового виразу або кінцю рядка при багаторядковому пошуку.	Hello\$	"World, Hello"
\b	Відповідає границі слова, тобто відповідає позиції між символом \ w і символом \ W або між символом \ w і початком або кінцем рядка.	\b(my)\b	У рядку "Hello my world" вибере слово "my"

\B	Відповідає позиції, що не є границею слів.	\B(ld)\b	Відповідність знайдеться в слові "World", але не в слові "ld"
----	--	----------	---

1.7.2 Використання регулярних виразів в C#

Безумовно, завдання пошуку і заміни підрядка в рядку можна вирішити на C# з використанням різних методів System.String і System.Text.StringBuilder. Проте в деяких випадках це вимагатиме написання великого обсягу коду C#. Якщо ви використовуєте регулярні вирази, то весь цей код скорочується буквально до кількох рядків. По суті, ви створюєте екземпляр об'єкта Regex, передаєте йому рядок для обробки, а також сам регулярний вираз (рядок, що включає інструкції мовою регулярних виразів) - і все готово.

У наступній таблиці показана частина інформації про перерахування RegexOptions, примірник якого можна передати конструктору класу Regex:

Таблиця 1.2 - Структура RegexOptions:

Член	Опис
CultureInvariant	Наказує ігнорувати національні установки рядка
ExplicitCapture	Модифікує спосіб пошуку відповідності, забезпечуючи тільки буквально відповідність
IgnoreCase	Ігнорує регістр символів у вхідному рядку
IgnorePatternWhitespace	Видаляє з рядка не захищені керуючими символами пропуски і дозволяє коментарі, що починаються зі знака фунта або хешу
Multiline	Змінює значення символів ^ і \$ так, що вони застосовуються до початку і кінця кожного рядка, а не тільки до початку і кінця всього вхідного тексту
RightToLeft	Наказує читати вхідну рядок справа наліво замість спрямування за замовчуванням - зліва направо (що зручно для деяких азійських та інших мов, які читаються в такому напрямку)

Після створення шаблону регулярного виразу з ним можна здійснити різні дії, в залежності від того, що вам необхідно. Можна просто перевірити, чи існує текст, відповідний шаблоном, у вихідній рядку. Для цього потрібно використовувати метод `IsMatch()`, який повертає логічне значення (метод 1).

Видобути всі збіги можна і простішим способом, використовуючи метод `Regex.Matches()`, який повертає об'єкт класу `MatchCollection`, який, в свою чергу, містить відомості про всі збіги, які обробник регулярних виразів знаходить у вхідному рядку. Наприклад, попередній приклад може бути переписаний для виклику методу `Matches` замість методу `Match` і методу `NextMatch` (метод 2):

```
private void matchButton_Click(object sender, EventArgs e)
{
    string input = "Some data items: " +
        "80 cnt aaaaaaa - 80 kg. " +
        "70 cnt. bbbbbbbb - 30 kg. " +
        "0.8 cnt. cccccc - 70 kg.";

    string pattern = @"b(\d+\W?kg)";
    Regex regex = new Regex(pattern);

    //----- Method 1
    Match match = regex.Match(input);
    label1.Text = "";
    while (match.Success)
    {
        label1.Text = label1.Text + match.Groups[1].Value + "\n";
        match = match.NextMatch();
    }
    //----- Method 2
    label2.Text = "";
    foreach (Match matchA in regex.Matches(input))
    {
        label2.Text = label2.Text + matchA.Groups[1].Value + "\n";
    }
}
```

Можна не просто отримувати збіги в заданому рядку, а й замінювати їх на власні значення. Для цього використовується метод `Regex.Replace()`. Як заміну методу `Replace()` можна передавати як рядок, так і шаблон заміни. У наступній таблиці показано як формуються метасимволи для заміни:

Таблиця 1.3 – Формування метасимволів заміни

Символ	Опис	Приклад шаблону	Приклад шаблону заміни	Результат (вхідний -> результуючий рядок)
\$ number	Заміщає частину рядка, відповідну групі number	\b(\w+)(\s)(\w+)\b	\$3\$2\$1	"один два" -> "два один"
\$\$	Підставляє літерал "\$"	\b(d+)\s?USD	\$\$\$1	"103 USD" -> "\$103"
\$&	Заміщає копією повної відповідності	(\\$(\d*(\.\d+)?)\{1})	**\$&	"\$1.30" -> "***\$1.30**"
\$`	Заміщає весь текст вхідного рядка до відповідності	B+	\$`	"AABBCC" -> "AAAACC"
\$'	Заміщає весь текст вхідного рядка після відповідності	B+	\$'	"AABBCC" -> "AACCCC"
\$+	Заміщає останню захоплену групу	B+(C+)	\$+	"AABBCCDD" -> "AACCCDD"

\$ _	Заміщає весь вхідний рядок	B+	\$ _	"AABBCC" -> "AAAABBCCC C"
------	-------------------------------------	----	------	---------------------------------

Контрольні питання.

1. Принципи функціонування регулярних виразів в програмуванні.
2. Об'єкт RegexOptions і його використання.

2 Візуальні програми

Середовище програмування C# є зручним для побудови віконних програм (синоніми: аплікацій, застосувань, застосунків, додатків, тощо), у випадку, якщо необхідна швидка їх реалізація. Успадкована від Borland C++ Bulder/ Borland Delphi концепція візуального форумання інтерфейсу, забезпечує швидку побудову інтерфейсу. Середовище має типовий набір графічних компонентів – кнопок, міток, перемикачів, текстових редакторів, засобів побудови графіків, та таблиць, тощо. Кожний з цих засобів це клас, який має вбудовані набори методів, які забезпечують реалізацію широкого функціоналу, як правило потрібного при побудові офісних програм. Такі програми не мають підвищених вимог до швидкодії обробки даних, відображення якісної графіки та звуку. Тому вибір мови C# для програм типу ігр, обробки відео та звуку, є неоптимальним варіантом. В тому випадку вибір мов C/C++ забезпечить приблизно в 3 рази більшу швидкість виконання розробленої програми і можливість їх легкої портації на платформи відмінні від Microsoft Windows. Для об'єктивності слід зауважити, що мова C# може бути застосована для розроблення ігрових додатків, у випадку використання додаткових зовнішніх бібліотек для роботи з графікою та звуком. У випадку розроблення бухгалтерського програмного забезпечення, програм обліку складської інформації, інформаційно-логістичних систем – вибір мови C# (або JAVA) є повністю виправданим, оскільки в цьому разі розробник одержує додаткові засоби для виявлення помилок при розробленні своєї програми, по відношенню до мови C/C++.

Принцип побудови графічної програми дуже простий: на основну форму програми перетягується необхідний графічний компонент (наприклад кнопка) і встановлюється в тому місці, де це потрібно розробникові. Далі подвійний клік по кнопці призводить до відкривання програмного коду де розробник просто дописує необхідний йому функціонал у метод, у який програма потрапить, коли користувач натисне цю кнопку.

Розглянемо деякі основні компоненти графічного інтерфейсу, за допомогою яких можна побудувати більшу частину програмного забезпечення.

Основне вікно середовища розробки виглядає так, як наведено на рис 2.1. На форму, наведений для прикладу перетягнуті кнопки, групи, радіокнопки та інші компоненти з Toolbox. В розгорнутому вигляді стають доступними наступні загальні елементи управління, (див. рис.2.2)

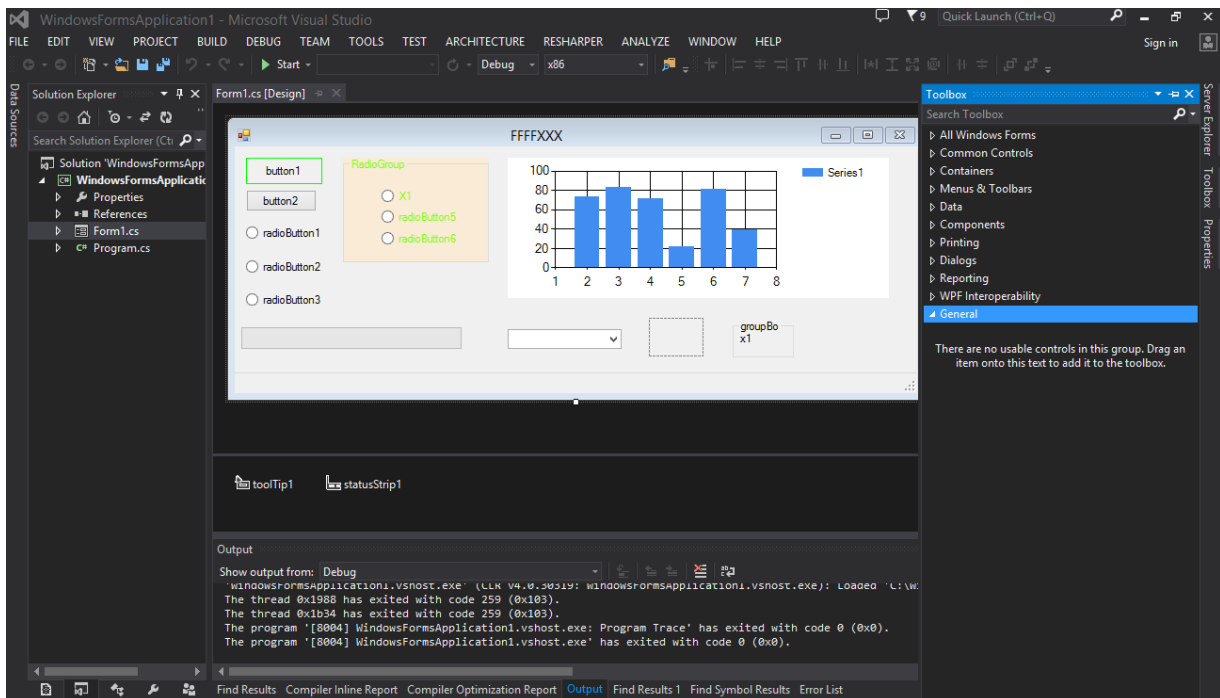


Рисунок 2.1– Основне середовище розробки графічних аплікацій

При створенні проекту «DLL» автоматично створюється у проекті один порожній клас, в консольному проекті створюється клас, що містить метод Main з порожнім кодом методу. У Windows-проекті автоматично створюються два класи - клас з ім'ям Form1 і клас з ім'ям Program. Перший з цих класів є спадкоємцем класу Form з бібліотеки FCL і успадковує всі властивості і поведінку (методи і події) батьківського класу. Клас Form підтримує організацію інтерфейсу користувача у візуальному стилі. Форма є контейнером для розміщення візуальних елементів управління - кнопок (Button), текстових полів (TextBox), списків (ListBox) і більш екзотичних елементів - таблиць (DataGridView), дерев (TreeView) і багатьох інших елементів.

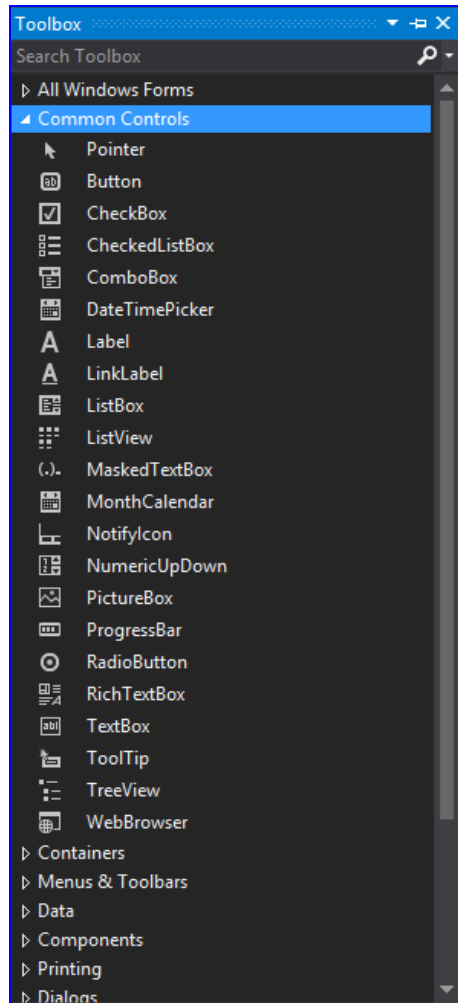


Рисунок 2.2– Панель Toolbox з основними графічними компонентами

Класи в C# синтаксично не є неподільними і можуть складатися з декількох частин, кожна з яких починається з ключового слова "partial" (частковий). Таким є і побудований автоматично клас Form1. Можливість розбиття опису одного класу на частині з'явилася ще у версії мови C# 2.0, що полегшує роботу над великим класом. Кожна частина класу зберігається в окремому файлі зі своїм ім'ям. Одна частина класу Form1 лежить у файлі з ім'ям "Form1.Designer.cs". Ця частина класу заповнюється автоматично інструментарієм, що називається «Дизайнером форми». Коли ми займаємося візуальним проектуванням форми і розміщуємо на ній різні елементи управління, змінюємо їх властивості, надаємо формі потрібний вид, задаємо обробників подій для елементів управління, то Дизайнер форми транслює наші дії в дії над об'єктами відповідних класів, створює відповідний код і вставляє його в потрібне місце класу Form1. Передбачається, що розробник проекту не втручається в роботу Дизайнера і не коригує частину класу Form1, створену Дизайнером. Тим не менш, розуміти код, створений Дизайнером, необхідно, а іноді корисно і коригувати його. Інша частина класу Form1, що зберігається у файлі "Form1.cs", призначена для розробника - саме в ній розташовуються автоматично створювані обробники подій, що відбуваються з елементами

управління, код яких створюється самим розробником. Така технологія програмування, заснована на роботі з формами, називається візуальною, подієво-керованою технологією програмування.

Клас Program, автоматично створюваний в Windows-проекті, містить точку входу - статичний метод Main. На відміну від консольного проекту, де тіло процедури Main спочатку є порожнім і повинно було заповнюватися розробником проекту, в Windows-проектах процедура Main вже готова і, як правило, розробником не змінюється.

Що ж робить автоматично створена процедура Main? Вона працює з класом Application бібліотеки FCL, викликаючи по черзі три статичні методи цього класу - EnableVisualStyles, SetCompatibleTextRenderingDefault, Run. Про призначення перших двох методів можна судити по їх іменах. Основну роботу виконує метод Run - в процесі його виклику створюється об'єкт класу Form1 і відкривається форма - візуальний образ об'єкта, з якою може працювати кінцевий користувач проекту. Якщо форма спроектована і заповнена елементами управління, то кінцевому користувачеві залишається вводити власні дані в поля форми, натискати на кнопки, взагалі бути ініціатором виникнення різних подій у світі об'єктів форми. У відповідь на виникаючі події, в програмі починають працювати обробники подій, що призводить до змін світу об'єктів. Типовою ситуацією є проведення обчислень над даними, введеними користувачем, і відображення результатів цих обчислень в полях форми, призначених для цих цілей.

Для прикладу розглянемо властивості, які належать об'єкту Button, оскільки багато цих властивостей входять і до складу інших графічних об'єктів, які наявні в C#. Деякі з них наведені на рис 2.3.

В розділі **Accessibility** наведені властивості, які можуть бути використані для запису текстових даних, які можуть бути використані для передачі інформації для клієнтів доступу і взагалі можуть бути використані для утримання таких даних для довільних потреб. До таких полів належать **AccessibleDescriptor**, **AccessibleName**, **AccessibileRole**.

В розділі **Appearance** наведені наступні властивості:

BackColor – колір фону

BackgroundImage – зображення, яке може бути накладене на зображення кнопки для покращення графічного дизайну програми

BackgroundImageLayout – поле, яке дозволяє провести вирівнювання BackgroundImage на формі кнопки по центру, розмножити цей імідж по вертикалі або горизонталі, розтягувати його пропорційно, або непропорційно.

Cursor – визначає форму курсора, у випадку якщо він потрапляє на кнопку. Таким чином можна змінити його форму при наведенні миші на кнопку.

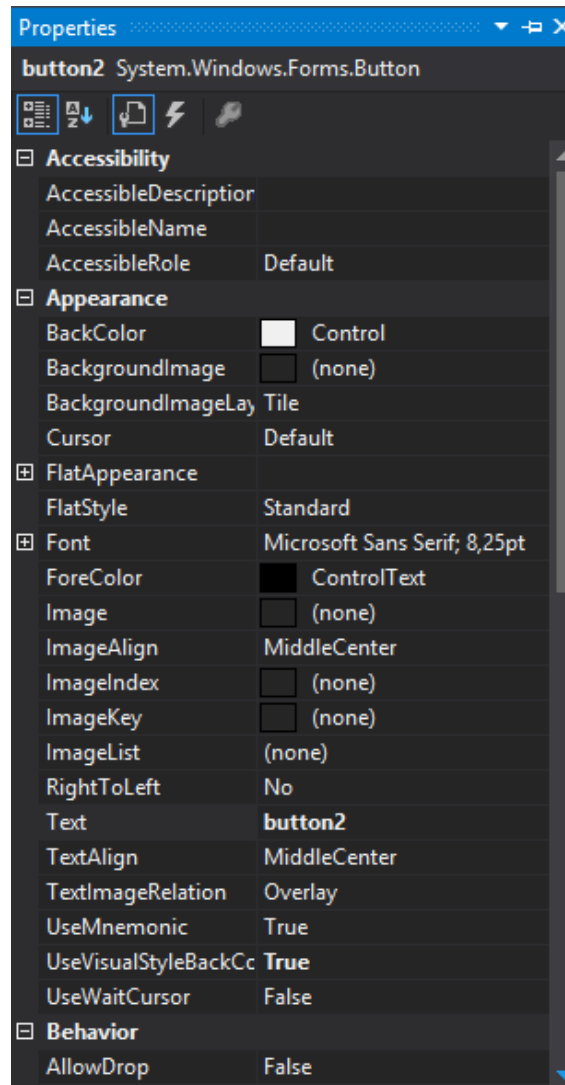


Рисунок 2.3 - Властивості

Контрольні питання:

1. Принцип функціонування візуальних компонентів в C#.
2. Принципи налаштування візуальних компонентів.

3 ASP.NET MVC

Платформа ASP.NET MVC являє собою фреймворк для створення сайтів і веб-додатків за допомогою реалізації паттерна MVC.

Концепція патерну (шаблону) MVC (model - view - controller) передбачає поділ додатка на три компоненти:

Контролер (controller) представляє клас, що забезпечує зв'язок між користувачем і системою, поданням і сховищем даних. Він отримує користувацькі дані і обробляє їх. І залежно від результатів обробки відправляє користувачу певний вивід, наприклад, у вигляді представлення- view.

Представлення (view) - це власне візуальна частина або користувацький інтерфейс програми. Як правило, html-сторінка, яку користувач бачить, зайшовши на сайт.

Модель (model) представляє клас, що описує логіку використовуваних даних.

Конкретні реалізації та визначення даного патерну можуть відрізнятися, але в силу своєї гнучкості та простоти він став дуже популярним останнім часом, особливо у сфері веб-розробки.

Свою реалізацію патерну представляє платформа ASP.NET MVC. 2013 рік ознаменувався виходом нової версії ASP.NET MVC - MVC 5, а також релізом Visual Studio 2013, яка надає інструментарій для роботи з MVC5.

Хоча в багатьох аспектах MVC 5 не надто сильно відрізнятиметься від MVC 4, багато чого з однією версією цілком застосовне до іншої, але в той же час є й істотні відмінності:

- У MVC 5 змінилася концепція аутентифікації та авторизації. Замість SimpleMembershipProvider була впроваджена система ASP.NET Identity, яка використовує компоненти OWIN і Katana.
- Для створення адаптивного і розширюваного інтерфейсу в MVC 5 використовується css-фреймворк Bootstrap
- Додані фільтри аутентифікації, а також з'явилася функціональність перевизначення фільтрів
- У MVC 5 також додані атрибути маршрутизації

Це найбільш важливі нововведення у MVC 5. Крім того, є ще ряд менш значущих, наприклад, використання за замовчуванням Entity Framework 6, деякі зміни при створенні проекту (концепція One ASP.NET), додаткові компоненти і т.д.

3.1 Початок роботи

Для створення веб-додатків на платформі ASP.NET MVC 5 необхідне середовище розробки - Visual Studio Community 2013 (або інший випуск Visual Studio 2013).

Після установки відкриємо Visual Studio 2013 і в меню File (Файл) виберемо пункт New Project ... (Створити проект). Перед нами відкриється діалогове вікно створення проекту. Оскільки в компанії Microsoft взято курс під назвою "One ASP.NET", то ми не побачимо, як в колишніх випусках Visual

Studio, різноманітність типів проектів. Замість цього нам буде доступний тільки один тип проекту:

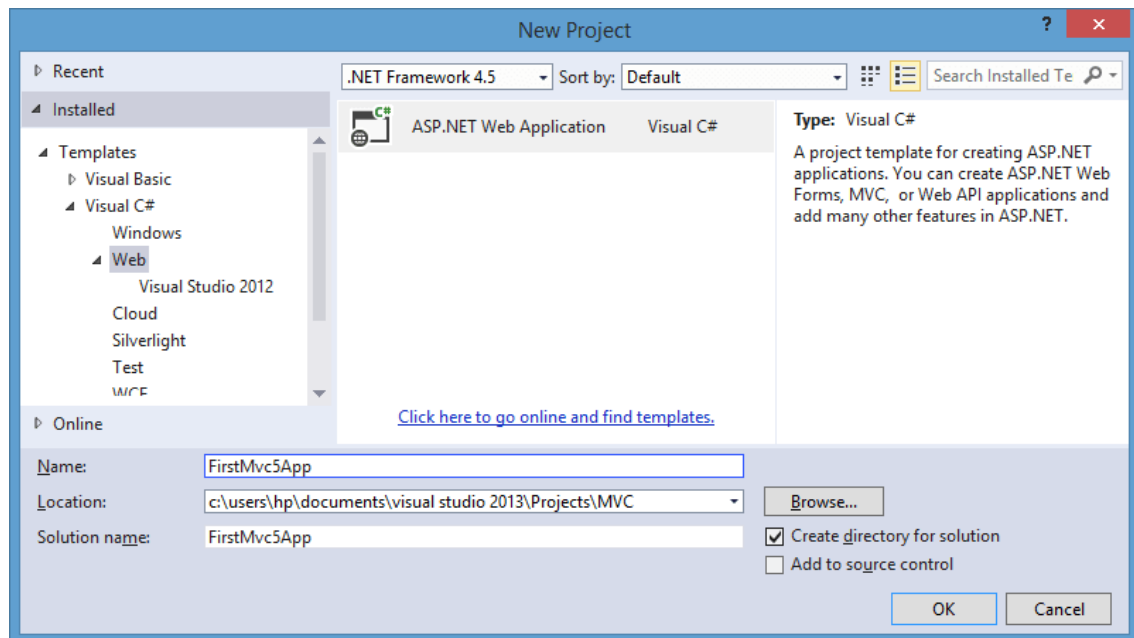


Рисунок 3.1 – Стартова сторінка

Демо яке-небудь ім'я проекту і натискаємо ОК. Після цього відобразиться вікно вибору шаблону нового додатка:

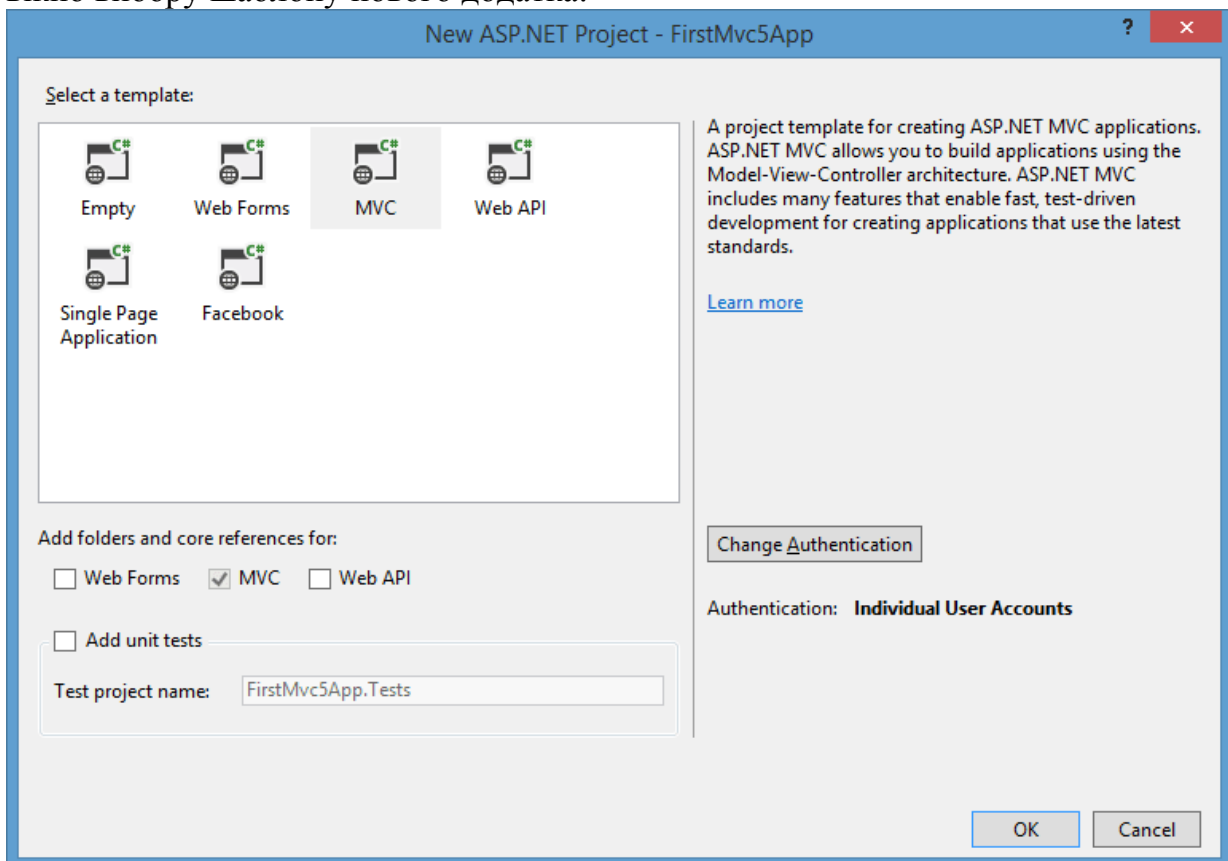


Рисунок 3.2– Вікно вибору шаблону

Типово уже обраний шаблон MVC. Крім того, дане діалогове вікно дозволяє задати опції тестування. Також нам доступний в правій частині вікна вибір механізму аутентифікації в додатку (кнопка Change Authentication). За замовчуванням встановлений тип Individual User Accounts. Не будемо його змінювати. Але якщо ми натиснемо на кнопку Change Authentication, то нам буде доступний вибір з наступних типів аутентифікації:

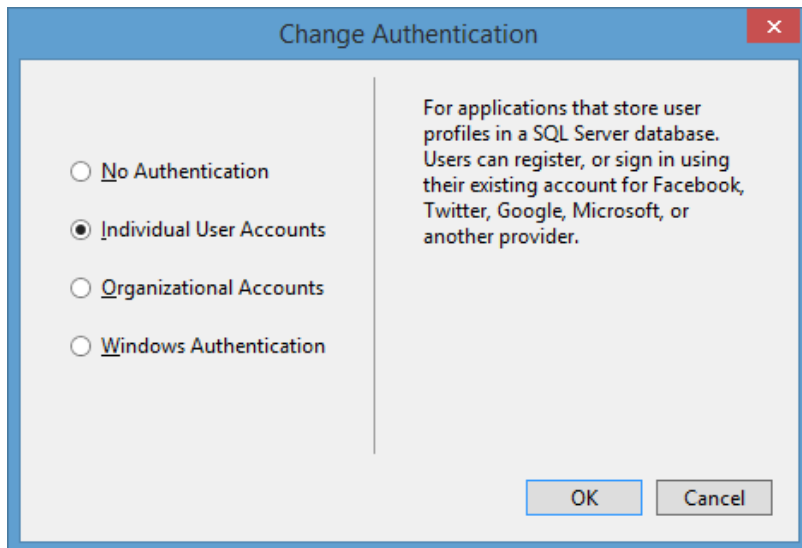


Рисунок 3.3 – Вікно типів аутентифікації

Що вони представляють?

- No Authentication: додаток не вимагає аутентифікації користувача.
- Individual User Accounts: потрібна індивідуальна аутентифікація, облікові записи користувачів зберігаються в базі даних, крім того, доступна аутентифікація за допомогою соціальних мереж.
- Organizational Accounts: в основному призначено для корпоративних додатків, які використовують Active Directory або Office 365
- Windows Authentication: для управління обліковими записами використовується аутентифікація Windows, так звані intranet-додатки

Натискаємо кнопку OK, і створюється новий проект. Він вже містить розгалужену структуру і має деякий наповнення за замовчуванням. Запустимо проект на виконання, і нам відобразиться деякий контент, який вже є за замовчуванням в додатку:

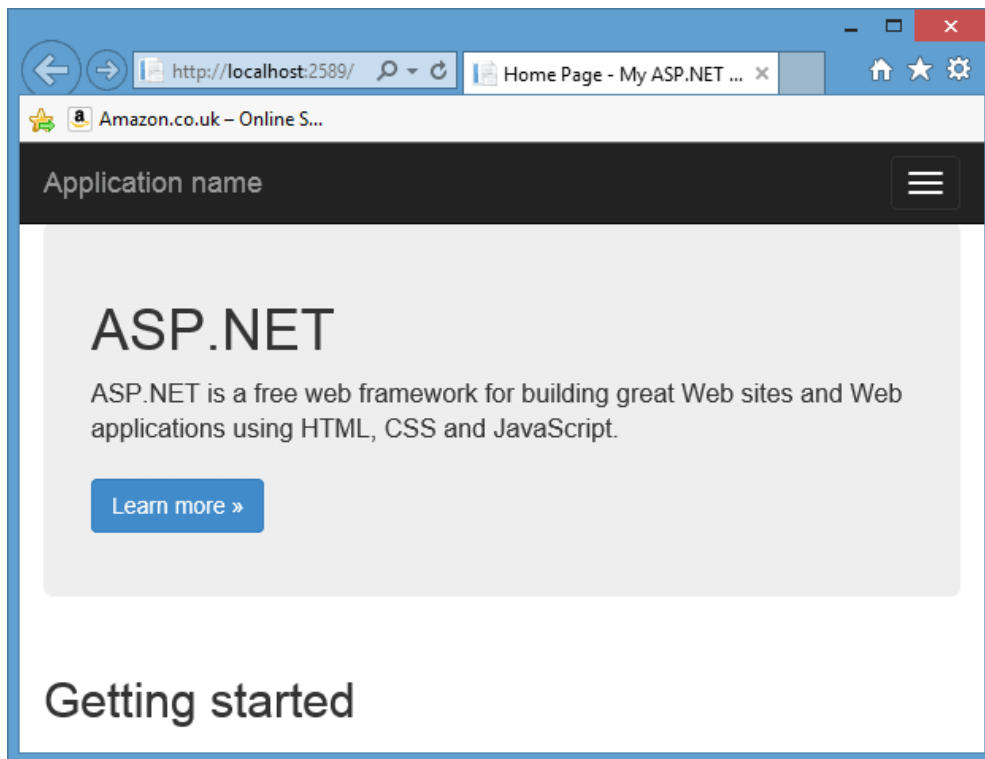


Рисунок 3.4 - Веб-сторінка яка створюється за замовчуванням.

3.1.1 Структура проекту MVC 5

Весь цей функціонал забезпечується наступною структурою проекту:

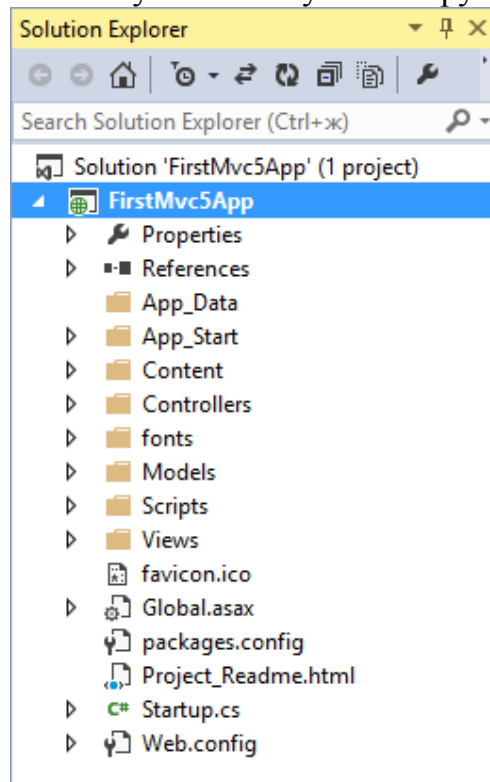


Рисунок 3.5– Структура проекту

Коротко розглянемо, для чого потрібні ці папки та файли.

- App_Data: містить файли, ресурси і бази даних, використовувані додатком
- App_Start: зберігає ряд статичних файлів, які містять логіку ініціалізації програми при запуску.
- Content: містить допоміжні файли, які не включають код на с # або javascript, і які розгортаються разом з додатком, наприклад, файли стилів css
- Controllers: містить файли класів контролерів. За замовчуванням в цю папку додаються два контролери - HomeController і AccountController
- fonts: зберігає додаткові файли шрифтів, використовуваних додатком
- Models: містить файли моделей. За замовчуванням Visual Studio додає пару моделей, що описують обліковий запис і службовців для аутентифікації користувача
- Scripts: каталог зі скриптами і бібліотеками на мові javascript
- Views: тут зберігаються представлення. Всі представлення групуються по папках, кожна з яких відповідає одному контролеру. Після обробки запиту контролер відправляє одне з цих представлень клієнту. Також тут є каталог Shared, який містить загальні для всіх представлення
- Global.asax: файл, що запускається при старті додатка і виконує початкову ініціалізацію. Як правило, тут спрацьовують методи класів, визначених у папці App_Start
- Startup.cs: оскільки в додатку MVC 5 використовуються бібліотеки, які застосовують специфікацію OWIN, то даний файл організовує зв'язок між OWIN та програмою. (OWIN представляє специфікацію, що описує взаємодію між компонентами додатку)
- Web.config: файл конфігурації програми

Конкретна структура кожного окремого додатка, природно, буде відрізнятися, але гнучкість MVC дозволяє змінювати структуру, пристосовуючи, її до своїх потреб. Описані вище моменти будуть загальними для більшості проектів.

Контрольні питання.

1. Типи ідентифікації в MVC-проектах.
2. Модель Model-View-Controller.

3.2 Створення першої інтернет-програми на ASP.NET MVC 5

Ми розглянули деякі основні поняття патерну MVC, тепер створимо першу програму. Це буде дуже простенький додаток, мета якого - дати деяке початкове розуміння роботи з ASP.NET MVC 5. Ця програма буде емулювати роботу книгарні: воно буде надавати нам вибір книг, а користувач, що зайшов на сайт, зможе оформити покупку. Отже, відкриємо Visual Studio 2013 File -> New Project .. і створимо новий проект. Назвемо новий проект, наприклад, BookStore. Потім у вікні створення нового проекту виберемо MVC. А в правій частині вікна змінимо тип аутентифікації додатку на No Authentication (оскільки поки нам система аутентифікації не потрібна):

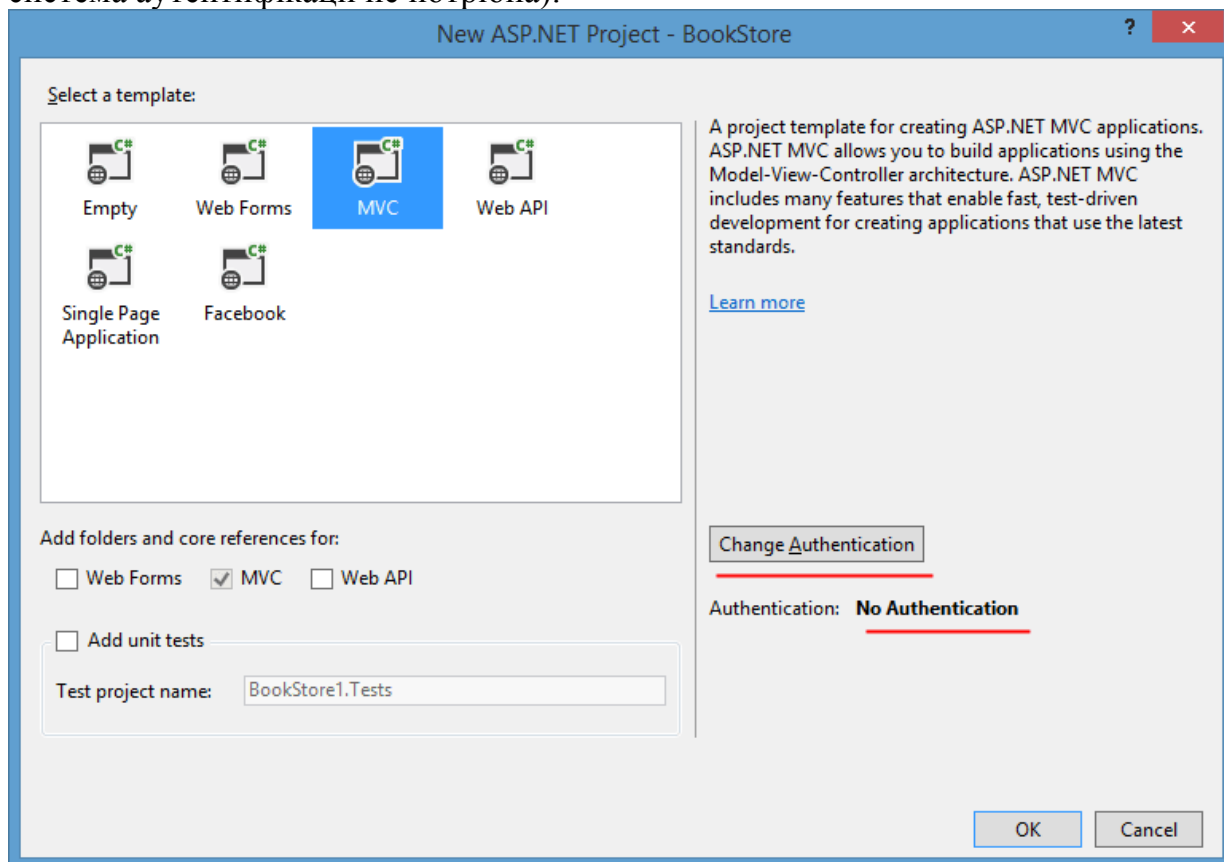


Рисунок 3.6 - Вибір типу аутентифікації

Після цього буде створений по суті проект, який практично не володіє ніякою функціональністю, хоча вже має базову структуру. Спочатку визначимо моделі даних нашого застосування. Оскільки мова йде про книгарню, то такими моделями можуть бути модель книги і модель покупки книги. У проекті вже за замовчуванням визначена папка Models. У ній будуть знаходитися наші моделі. Натиснемо на папку правою кнопкою миші і в меню виберемо Add-> Class Назвемо перший новий клас Book і додамо в нього код, що описує модель книги:

```
namespace BookStore.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; }
        // ціна
        public int Price { get; set; }
    }
}

```

Звичайно, в реальному додатку подібна модель могла б також включати зображення книги, кількість примірників, рік видання і т.д. Але для нашої задачі цілком зійде і цей набір полів. Подібним чином другий клас - модель Purchase, яка відповідатиме за окрему вчинену покупку книги:

```

using System;
namespace BookStore.Models
{
    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; }
    }
}

```

Модель являє звичайний клас на мові C#. Всі моделі тут мають набір властивостей, що описують реальні властивості об'єкта. У той же час при створенні моделей слід дотримуватися деяких умовності. Оскільки ми будемо використовувати для зберігання моделей базу даних SQL Server, то для маніпуляції над об'єктами в базі даних нам треба визначити для них первинний ключ (Primary Key), який виконує роль універсального ідентифікатора об'єкта. Тому першим властивістю в кожній моделі йде властивість Id, призначене для зберігання первинного ключа. І тут вступають в силу умовності: властивість ідентифікатора моделі повинна мати ім'я або Ім'я_моделіId, або просто Id. Так, у нас в моделі Book визначено властивість Id, тобто дана властивість є первинним ключем. А у випадку з моделлю Purchase властивість носить назву PurchaseId.

Другий спосіб полягає у визначенні ключа за допомогою атрибуту Key, встановленим над потрібним властивістю.

3.2.1 EntityFramework

Для роботи з даними в ASP.NET MVC рекомендується використовувати фреймворк Entity Framework, хоча його використання необов'язково і цілком залежить від розробника. Перевага цього фреймворку полягає в тому, що він дозволяє абстрагуватися від структури конкретної бази даних і вести всі операції з даними через модель.

Зараз наш проект не містить бібліотек EntityFramework. І щоб їх додати в проект, скористаємося пакетним менеджером NuGet. Отже, вікні Solution Explorer (Оглядач рішень) натиснемо правою кнопкою миші в структурі проекту на вузол References і в меню виберемо Manage NuGet Packages ...

У вікні управління пакетами NuGet в правому верхньому куті введіть у поле пошуку EntityFramework і натисніть Enter. Після цього в середньому стовпці будуть відображені всі знайдені пакети, які мають відношення до запиту, а найпершим буде пакет самого фреймворку EntityFramework, який нам і треба встановити:

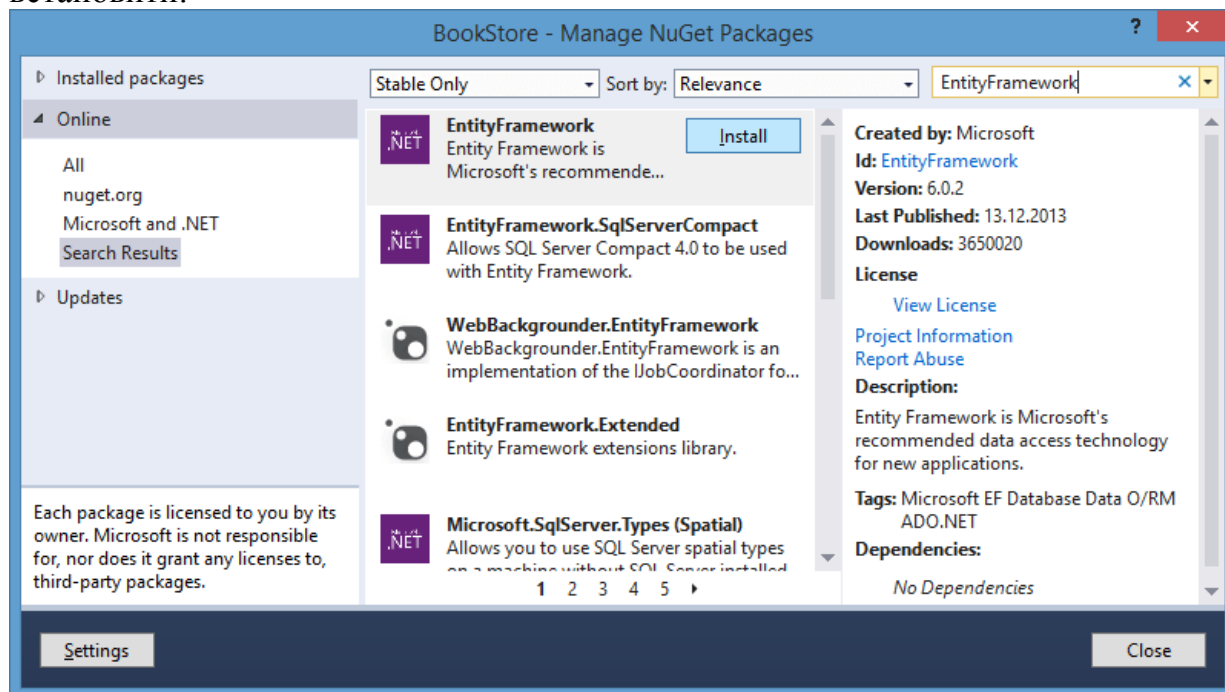


Рисунок 3.7– Установка пакету NuGet

Запустимо процес установки пакета, натиснувши на кнопку Install.

Після завершення установки створимо контекст даних. Контекст даних використовує Entity Framework для доступу до БД на основі деякої моделі. Отже, додамо в папку Models новий клас BookContext:

```
using System;  
using System.Collections.Generic;  
using System.Web;  
using System.Data.Entity;
```

```
namespace BookStore.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 зразок `public DbSet <Book> Books {get; set; }` Допомагають отримувати з БД набір даних певного типу (наприклад, набір об'єктів Book).

3.2.2 CodeFirst

Хоча ми будемо використовувати базу даних, але створювати явним чином ми її не будемо. За нас все зробить EntityFramework. Це так званий підхід Code First - у нас є моделі, і по них фреймворк буде створювати таблиці в базі даних. Для запуску роботи над модельною частиною встановимо рядок підключення. Для цього відкриємо файл web.config, знайдемо секцію configSections і відразу після неї вставимо секцію connectionStrings:

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

У цій секції ми визначаємо шлях до бази даних, яка потім буде створюватися. Вираз `|DataDirectory|` представляє заступник, який вказує, що база даних буде створюватися в проєкті в папці App_Data. Пізніше ми докладніше розберемо настройки підключення до БД.

При створенні підключення треба враховувати версію MS SQL Server, з якою доведеться працювати. Вище наведена рядок підключення для MS SQL Server 2012. При використанні версії MS SQL Server +2014 рядок підключення може трохи відрізнятись:

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

Контрольні питання.

1. Основні переваги використання фреймворку EntityFramework.
2. Функціонування підходу Code First.

3.3 Створення контролера і представлень

Так як з моделями і налаштуванням контексту даних ми закінчили, то перейдемо до іншого компоненту додатку - контролера. Для контролерів призначена папка Controllers. За замовчуванням при створенні проекту в неї додається контролер HomeController, який практично не має ніякої функціональності, і зараз його код виглядає наступним чином:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace BookStore.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            return View();
        }
        public ActionResult About()
        {
            ViewBag.Message = "Your application description page.";
            return View();
        }
        public ActionResult Contact()
        {
            ViewBag.Message = "Your contact page.";
            return View();
        }
    }
}
```

У контролері визначені за замовчуванням три методи: Index, About і Contact. Нам вони не потрібні. Змінимо код контролера на наступний:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using BookStore.Models;
```

```

namespace BookStore.Controllers
{
    public class HomeController : Controller
    {
        // створюємо контекст даних
        BookContext db = new BookContext();

        public ActionResult Index()
        {
            // одержуємо 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, ми підключаємо простір імен моделей, навіть незважаючи на те, що він знаходяться в одному проекті, але в різних просторах. Потім створюється об'єкт контексту даних, через який ми будемо взаємодіяти з бд: `BookContext db = new BookContext ()` ; Далі використовуючи властивість `db.Books`, отримуємо з бази даних набір об'єктів `Book`. Тепер треба передати цей набір в представлення. Для передачі списку об'єктів `Book` в представлення використовуємо об'єкт `ViewBag`. `ViewBag` представляє такий об'єкт, який дозволяє визначити будь-яку змінну і передати їй деяке значення, а потім у поданні витягти це значення. Так, ми визначаємо змінну `ViewBag.Books`, яка і буде зберігати набір книг. Тепер створимо саме представлення для виведення списку книг. Для представлень у проекті призначена папка `Views`. За умовчанням в цій папці вже є підкаталог для вистав контролера `Home`, в якому три представлення: `About.cshtml`, `Contact.cshtml` і `Index.cshtml`:

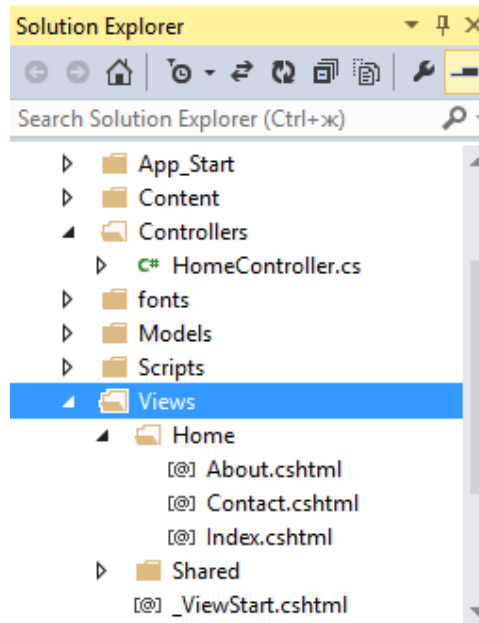


Рисунок 3.8 – Оглядач проекту

Перші два подання нам вже не знадобляться, і їх можна спокійно видалити. А представлення `Index.cshtml` відкриємо і змінимо наступним чином:

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Книжковий магазин</title>
</head>
<body>
    <div>
        <h3>Розпродаж книжок</h3>
        <table>
            <tr><td><p>Назва книги</p></td>
                <td><p>Автор</p></td>
                <td><p>Ціна</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/Buy/@b.Id">Купити</a></p></td>
                </tr>
            }
        </table>
    </div>
</body>
</html>
```

```

    }
  </table>
</div>
</body>
</html>

```

Найпершим виразом `Layout = null`; ми вказуємо, що майстер-сторінка не буде застосовуватися до цього подання. Далі ми додамо до нього майстер-сторінку і дізнаємося, навіщо вона потрібна, а поки обійдемося без неї.

Практично весь інший код представляє собою стандартний код мовою html: створення звичайної таблиці, яка виводить інформацію про продаваних книгах. Тут також використовується цікава конструкція:

`foreach (var b in ViewBag.Books)`. Ця конструкція застосовує синтаксис Razor. Докладніше про движку Razor і його сітаксісе ми поговоримо в окремому розділі, а поки вам треба знати, що після символу `@` згідно синтаксису ми можемо використовувати вирази коду мовою C# / VB.NET.

Тобто тут ми створюємо цикл. У ньому ми пробігати по всіх елементах в об'єкті `ViewBag.Books`, який був раніше створений в методі контролера. І потім отримуємо значення властивості кожного елемента за допомогою синтаксису Razor: `@b.Name` і поміщаємо його в комірку таблиці.

В останню колонку таблиці для кожного елемента додається посилання `<ahref="/Home/Buy/@b.Id"> Купити `. При натисканні на це посилання методу `Buy` контролера `HomeController` відправлятиметься запит, в якому замість `@b.Id` буде вказаний id книги. Поки у нас, правда, відсутня метод `Buy`, але скоро ми його створимо.

3.3.1 Стилiзацiя програми i майстер сторiнки

Додамо в наш додаток невелику примітивну стилізацію. Для цього визначимо файл стилів. За замовчуванням Visual Studio вже додає файл стилів `Site.css` в папку `Content`:

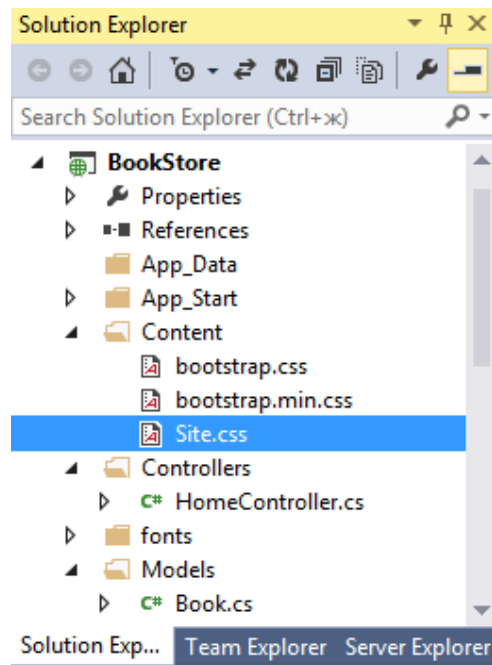


Рисунок 3.9 – Оглядач проекту

Крім файлу Site.css, у папці Content знаходиться файл css-фреймворку Bootstrap, Відкриємо файл Site.css і змінимо його вміст наступним чином:

```
body {
    font-size: 13px;
    font-family: Verdana, Arial, Helvetica, Sans-Serif;
    background-color: #f7f7fa;
    padding-left: 40px;
}

nav{
    display: block;
}

.menu {
    padding-left: 10px;
}
.menu ul {
    list-style: none;
}
.menu li {
    display: inline;
}
.menu a: hover {
    color: red;
}
table {
    vertical-align: middle;
    text-align: left;
}
```

```

}
.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. Хоча наше додатку не дуже велике, тому там буде тільки посилання на головну сторінку. Але при необхідності ви можете додати в нього додаткові пункти.

Щоб використовувати стилі, ми можемо їх просто підключити в секції `head`, як для будь-якої звичайної html-сторінки:

```

<head>
    <meta name="viewport" content="width=device-width" />
    <link type="text/css" rel="stylesheet" href="../../Content/Site.css" />
</head>

```

У нашому випадку досить вставити даний код на обидва наших VIEW. Однак це не найкращий підхід, так як стилі для обох VIEW загальні, крім того, подібних VIEW в проекті може бути не дві, а набагато більше. І якщо ми раптом змінимо посилання на файл стилів, то доведеться міняти це посилання на всіх VIEW.

І щоб вийти з цієї проблеми фреймворк ASP.NET MVC надає нам таку функціональність, як майстер-сторінки. Майстер-сторінка задає єдиний шаблон для інших VIEW які його використовують.

За умовчанням в проекті вже є майстер-сторінка, яка називається `_Layout.cshtml`. Її можна знайти в папці Views -> Shared:

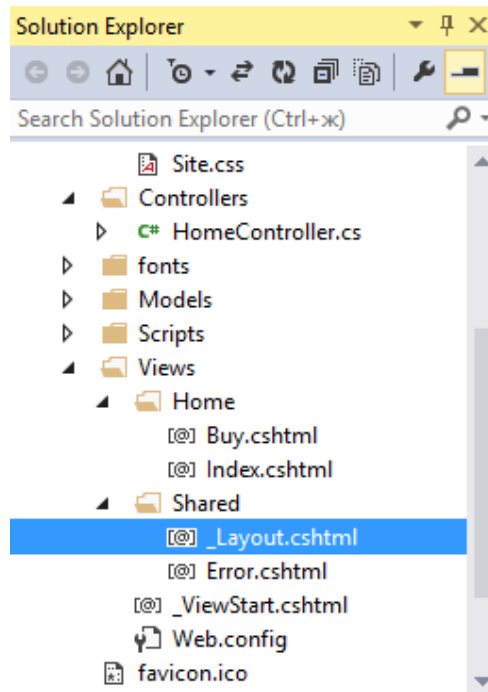


Рисунок 3.10 - Розміщення майстер сторінки в оглядачі.

Файл `_Layout.cshtml` вже має деякий вміст за замовчуванням. Змінимо його на наступний:

```
<!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>
```

Для підключення стилів тут використовувався інший спосіб - метод `Url.Content`, в який передається шлях до файлу. Після секції `head` на майстер-сторінці йде створення меню. Так як у нас всього два подання, то в якості одного єдиного пункту меню вказується посилання на головну сторінку. Для створення посилання використовується метод `Html.ActionLink`. Він генерує елемент-посилання і приймає назву посилання, метод контролера і ім'я контролера. Далі

Йде виклик методу `RenderBody()` - за допомогою цього методу в це місце буде підставлятися розмітка вже конкретних представлень. Тепер змінимо представлення так, щоб вони використовували майстер-сторінку. Оновлене представлення `Index.cshtml`:

```
@{
    Layout = "~/Views/Shared/_Layout.cshtml";
}

<div>
    <h3>Розпродаж 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 (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/Buy/@b.Id">Купити</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" action="">
        <input type="hidden" value="@ViewBag.BookId" name="BookId" />
        <table>
            <tr><td><p>Введіть своє ім'я </p></td>
                <td><input type="text" name="Person" /> </td></tr>
            <tr><td><p>Введіть адресу :</p></td><td>
                <input type="text" name="Address" /> </td></tr>
            <tr><td><input type="submit" value="Відправити" /> </td><td></td></tr>
        </table>
    </form>
</div>
```

Щоб вказати використовувану майстер-сторінку, на початку представлення прописується шлях до майстер-сторінки: `Layout = "~/Views / Shared / _Layout.cshtml"`; . Тепер нам більше не потрібні секції `head` і `body`. Ми

можемо їх видалити. Запустимо на виконання оновлений проект і переконаємося, що до нашого додатку застосована стилізація та майстер-сторінки. На цьому робота над додатком закінчена, і тепер ми можемо перейти до більш детального обговорення основних компонентів додатка MVC.

3.3.2 Контролери

Контролер є центральним компонентом в архітектурі MVC. Контролер отримує введення користувача, обробляє його і посилає назад результат обробки, наприклад, у вигляді подання. При використанні контролерів існують деякі умовності. Так, за угодами про іменування назви контролерів повинні закінчуватися на суфікс "Controller", інша ж частина до цього префікса вважається назвою контролера. Щоб звернутися контролеру з веб-браузера, нам треба в адресному рядку набрати адрес_сайта / Імя_контроллера /. Так, за запитом адрес_сайта / Home / система маршрутизації за замовчуванням викличе метод Index контролера HomeController для обробки вхідного запиту. Якщо ми хочемо відправити запит до конкретного методу контролера, то потрібно вказувати цей метод явно: адрес_сайта / Імя_контроллера / Метод_контроллера, наприклад, адрес_сайта / Home / Buy - звернення до методу Buy контролера HomeController. Контролер являє звичайний клас, який успадковується від базового класу System.Web.Mvc.Controller. У свою чергу клас Controller реалізує абстрактний базовий клас ControllerBase, а через нього і інтерфейс IController. Таким чином, формально, щоб створити свій клас контролера, достатньо створити клас, який реалізує інтерфейс IController і що має в імені суфікс Controller. Інтерфейс IController визначає один єдиний метод Execute, який відповідає за обробку контексту запиту:

```
public interface IController
{
    void Execute(RequestContext requestContext);
}
```

Тепер створимо який-небудь контролер, який реалізує даний інтерфейс. В якості проекту ми можемо взяти попередній проект. Отже, додамо в папку Controllers проекту новий клас (саме клас, а не контролер) з наступним змістом:

```
using System.Web.Mvc;
using System.Web.Routing;

namespace BookStore.Controllers
{
    public class MyController : IController
    {
        public void Execute(RequestContext requestContext)
        {
            string ip = requestContext.HttpContext.Request.UserHostAddress;
            var response = requestContext.HttpContext.Response;
            response.Write("<h2>Ваш IP-адрес: " + ip + "</h2>");
        }
    }
}
```

```
}  
}
```

При зверненні до будь-якого контролера система передає в нього контекст запиту. У цей контекст запиту включається все: куки, відправлені дані форм, рядка запиту, ідентифікаційні дані користувача і т.д. Реалізація інтерфейсу `IController` дозволяє отримати цей контекст запиту в методі `Execute` через параметр `RequestContext`. У нашому випадку ми отримуємо IP-адресу користувача через властивість `requestContext.HttpContext.Request.UserHostAddress`. Крім того, ми можемо відправити користувачеві відповідь за допомогою об'єкта `Response` і його методу `Write`. Таким чином, перейшовши по шляху адрес_сайта / Му /, користувач побачить свою ip-адресу. Хоча за допомогою реалізації інтерфейсу `IController` дуже просто створювати контролери, але в реальності частіше оперують більш високорівневими класами, як наприклад клас `Controller`, оскільки він надає більш потужні засоби для обробки запитів. І якщо при реалізації інтерфейсу `IController` ми маємо справу з одним методом `Execute`, і всі запити до цього контролеру, будуть оброблятися тільки одним методом, то при спадкуванні класу `Controller` ми можемо створювати безліч методів дій, які відповідатимуть за обробку вхідних запитів, і повертати різні результати дій. Щоб створити стандартний контролер, ми можемо також додати в папку `Controllers` простий клас і успадкуватись від класу `Controller`, наприклад:

```
using System.Web.Mvc;  
  
namespace BookStore.Controllers  
{  
    public class BookShopController : Controller  
    {  
        public ActionResult Index()  
        {  
            return View();  
        }  
    }  
}
```

Однак Visual Studio пропонує нам більш зручні засоби для створення контролерів, які передбачають їх гнучку настройку. Щоб ними скористатися, натиснемо на папку `Controllers` правою кнопкою миші і в меню виберемо `Add -> Controller` Після цього нам відобразиться вікно створення нового контролера:

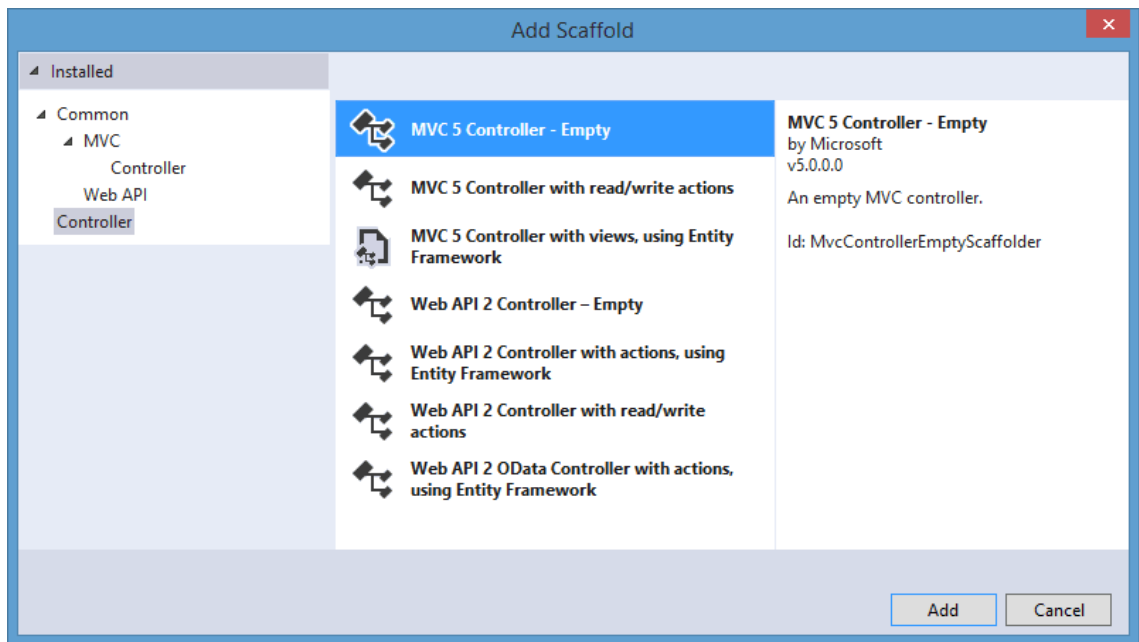


Рисунок 3.11 - Вікно створення контролера

Власне до контролерів MVC 5 тут безпосереднє відношення мають перші три пункти. Решта більше відносяться до Web API2. У переліку виберемо перший пункт - MVC5 Controller - Empty, який передбачає створення порожнього контролера. Решта два пункти дозволяють згенерувати класи з CRUD-функціональністю на основі шаблонів формування, про які ми поговоримо в розділі про моделі.

Далі нам буде запропоновано ввести ім'я, і після цього новий контролер з єдиним методом Index буде додано в проект. При такому додаванні на відміну від попередніх прикладів для даного контролера буде автоматично створено каталог в папці Views, який буде зберігати всі представлення, пов'язані з діями цього контролера.

3.3.3 Методи дій та їх параметри

Методи дій (action methods) представляють такі методи контролера, які обробляють запити за певним URL. Наприклад, розглянемо попередній проект. У ньому був визначений наступний контролер:

```
public class HomeController : Controller
{
    BookContext db = new BookContext();

    public ActionResult Index()
    {
        IEnumerable<Book> books = db.Books;
        ViewBag.Books = books;
        return View();
    }
}
```

```

[HttpGet]
public ActionResult Buy(int id)
{
    ViewBag.BookId = id;
    return View();
}
[HttpPost]
public string Buy(Purchase purchase)
{
    purchase.Date = DateTime.Now;
    db.Purchases.Add(purchase);
    db.SaveChanges();
    return "Дякуємо, " + purchase.Person + ", за покупку!";
}
}

```

Тут методи Index і Buy є методами дій або просто діями контролера. При отриманні запиту типу / Home / Index контролер передає обробку запиту дії Index. Так як запити бувають різних типів, наприклад, GET і POST, фреймворк ASP.NET MVC дозволяє визначити тип оброблюваного запиту для дії, застосувавши до нього відповідний атрибут: [HttpGet], [HttpPost], [HttpDelete] або [HttpPut]. Так, дія Buy розбита на два методи, по одному для кожного типу запиту. Однак не всі методи контролера є методами дій. Методи дій завжди мають модифікатор public. Закритих приватних методів дій не буває. Але контролер може також включати і звичайні методи, які можуть використовуватися в допоміжних цілях. Наприклад:

```

[HttpPost]
public string Buy(Purchase purchase)
{
    purchase.Date = getToday();
    db.Purchases.Add(purchase);
    db.SaveChanges();
    return "Спасибо, " + purchase.Person + ", за покупку!";
}
private DateTime getToday()
{
    return DateTime.Now;
}

```

Відповідно ми не можемо відправити з браузера запит Home / get Today /, бо метод getToday не є методом дії.

3.3.4 Передача даних в контролери і параметри

У попередньому додатку метод Buy використовував параметр purchase. Оскільки цей метод обробляє POST-запити, то ми можемо відправити йому таку форму:

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



```

<input type="hidden" value="@ViewBag.BookId" name="BookId" />
<p>Введіть своє ім'я </p>
<input type="text" name="Person" />
<p> Введіть адресу :</p>
<input type="text" name="Address" />
<input type="submit" value="Відправити" />
</form>

```

Значення атрибута name у всіх полів на цій формі відповідає назві властивості моделі, тому система автоматично зв'яже значення полів з відповідними властивостями. А в методі Buy весь цей набір властивостей перетвориться на модель Purchase. Крім POST-запитів у нас є також GET-запити, при яких всі параметри передаються в рядку запиту. Наприклад, друга версія методу Buy як параметр приймає значення типу int: public ActionResult Buy (int id). Стандартний get-запит приймає приблизно таку форму: назва_ресурса?параметр1 = значення1&параметр2=значення2. Тобто запит до даного методу міг би виглядати так: Home/ Buy?Id=2.

Назва параметрів методу має збігатися з назвою параметрів в рядку запиту. Завдяки цьому система зможе їх автоматично зв'язати. А в самому методі ми зможемо отримати цей параметр і використувувати його на свій розсуд. Крім того, система маршрутизації дозволяє створювати маршрути. Наприклад, за замовчуванням у проекті MVC визначається наступний маршрут: Контролер/Метод/id. Останній параметр є опціональним. І завдяки цьому ми можемо передати параметр id і так: Home/Buy/2 Для прикладу визначимо дію, яка буде підраховувати площа трикутника:

```

public string Square(int a, int h)
{
    double s = a*h/2;
    return "<h2>Площа трикутника з основою " + a +
        " і висотою " + h + " = " + s + "</h2>";
}

```

У цьому випадку ми можемо звернутися до дії, набравши в адресному рядку Home/Square?A = 10 & h = 3, і програма видала б нам потрібний результат. Ми також можемо задати для параметрів значення за замовчуванням:

```

public string Square(int a=10, int h=3)
{
    double s = a*h/2;
    return "<h2>Площа трикутника з основою " + a +
        " і висотою " + h + " = " + s + "</h2>";
}

```

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

3.3.5 Отримання даних з контексту запиту

Крім того, ми можемо отримати параметри, та й не тільки параметри, але й інші дані, пов'язані із запитом, з об'єктів контексту запиту. Нам доступні наступні об'єкти контексту: Request, Response, RoutedData, HttpContext і Server. Об'єкт Request містить колекцію Params, яка зберігає всі параметри, передані в запити. І ми їх можемо отримати:

```
public string Square()
{
    int a = Int32.Parse(Request.Params["a"]);
    int h = Int32.Parse(Request.Params["h"]);
    double s = a*h/2;
    return "<h2>Площа трикутника з основою " + a + " і висотою" + h + " = " + s + "</h2>";
}
```

3.3.6 Результати дій

Коли користувач звертається до ресурсу, він очікує отримати певну відповідь, наприклад, у вигляді веб-сторінки з деякими даними. На стороні сервера метод контролера, отримуючи параметри, обробляє їх і формує відповідь у вигляді результату дії.

В прикладі з обчисленням площі трикутника ми повертали html-код у вигляді рядка. Але, як правило, результатом, що повертається є об'єкт класу, похідного від ActionResult. ActionResult являє собою абстрактний клас, в якому визначено один метод ExecuteResult, що перевизначається в класах-спадкоємцях:

```
public abstract class ActionResult
{
    public abstract void ExecuteResult(ControllerContext context);
}
```

Створимо свої результати дій. Візьмемо проект, і додамо в нього нову папку Util, яка міститиме нові класи. Після додавання папки додамо в неї перший клас. Назвемо його HtmlResult. Він буде містити наступний код:

```
using System.Web.Mvc;

namespace BookStore.Util
{
    public class HtmlResult : ActionResult
    {
        private string htmlCode;
        public HtmlResult(string html)
        {
            htmlCode = html;
        }
        public override void ExecuteResult(ControllerContext context)
        {
            string fullHtmlCode = "<!DOCTYPE html><html><head>";
```

```

        fullHtmlCode += "<title>Головна сторінка</title>";
        fullHtmlCode += "<meta charset=utf-8 />";
        fullHtmlCode += "</head> <body>";
        fullHtmlCode += htmlCode;
        fullHtmlCode += "</body></html>";
        context.HttpContext.Response.Write(fullHtmlCode);
    }
}
}

```

У конструкторі класу `HtmlResult` отримуємо переданий `html`-код, а в методі `Execute` вставляємо його в загальне оточення, щоб вийшла повноцінна `html`-сторінка, і пишемо її у вихідний потік:

```
context.HttpContext.Response.Write (fullHtmlCode);
```

Щоб використовувати цей клас підключимо в контролер простір імен нового класу: `using BookStore.Util`; і додамо новий метод:

```

public ActionResult GetHtml()
{
    return new HtmlResult("<h2>Hello world!</h2>");
}

```

І звернувшись до цього методу з браузера, наприклад, `Home/GetHtml`, ми отримаємо `html`-сторінку. Хоча даний приклад досить примітивний, але в цілому він демонструє, як працюють класи результатів дій.

Створимо ще один клас результатів. Додамо в папку `Util` новий клас `ImageResult`:

```

using System.Web.Mvc;

namespace BookStore.Util
{
    public class ImageResult : ActionResult
    {
        private string path;
        public ImageResult(string path)
        {
            this.path = path;
        }
        public override void ExecuteResult(ControllerContext context)
        {
            context.HttpContext.Response.Write("<div style='width:100%;text-align:center;'>" +
                "<img style='max-width:600px;' src='" + path + "' /></div>");
        }
    }
}

```

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

```
public ActionResult GetImage()
```

```
{
    string path = "../Images/visualstudio.png";
    return new ImageResult(path);
}
```

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

3.3.7 Вбудовані класи, похідні від ActionResult

В реальності нам навряд чи буде потрібно часто створювати свої класи для обробки результату дії. Фреймворк ASP.NET MVC пропонує нам багату палітру класів результатів дій, які охоплюють більшість, якщо не всі можливі ситуації.

ContentResult: пише вказаний контент безпосередньо у відповідь у вигляді рядка, практично як попередні приклади. Так, наступний приклад:

```
public string Square(int a, int h)
{
    int s = a*h/2;
    return "<h2>Площа трикутника з основою " + a +
        " і висотою " + h + " = " + s + "</h2>";
}
```

Можна переписати з використанням ContentResult наступним чином:

```
public ContentResult Square(int a, int h)
{
    int s = a*h/2;
    return Content(""<h2>Площа трикутника з основою " + a +
        " і висотою " + h + " = " + s + "</h2>");
}
```

Навіть якщо ми залишимо в якості результату який повертається тип string, то фреймворк побачить, що тип, який повертається не є об'єктом ActionResult. І тоді автоматично створюється об'єкт ContentResult для такого рядка.

EmptyResult: по суті нічого не робить, відправляє порожню відповідь.

FileResult: є базовим класом для всіх об'єктів, які пишуть бінарну відповідь у вихідний потік.

FileContentResult: клас, похідний від FileResult, пише у відповідь масив байтів

FilePathResult: також похідний від FileResult класу, пише у відповідь файл, що знаходиться по заданому шляху.

FileStreamResult: клас, похідний від FileResult, пише бінарний потік у вихідну відповідь.

HttpStatusCodeResult: результат дії, який повертає клієнту певний статусний код HTTP.

HttpUnauthorizedResult: клас, похідний від `HttpStatusCodeResult`. Повертає клієнту відповідь у вигляді статусного коду HTTP 401, вказуючи, що користувач не пройшов авторизацію і не має прав доступу до запитаного ресурсу.

HttpNotFoundResult: похідний від `HttpStatusCodeResult`. Повертає клієнту відповідь у вигляді статусного коду HTTP 404, вказуючи, що запитаний ресурс не знайдений

JavaScriptResult: повертає у відповідь в якості вмісту код JavaScript.

JsonResult: повертає в якості відповіді об'єкт або набір об'єктів у форматі JSON.

PartialViewResult: виробляє рендеринг часткового представлення у вихідний потік.

RedirectResult: перенаправляє користувача за іншою адресою URL, повертаючи статусний код 302 для тимчасової переадресації або код 301 для постійної переадресації залежно від того, чи встановлений флаг `Permanent`.

RedirectToRouteResult: клас працює подібно `RedirectResult`, але перенаправляє користувача за певною адресою URL, вказаною через параметри маршруту

ViewResult: виробляє рендеринг представлення і відправляє результати рендеринга у вигляді html-сторінки клієнту

3.3.8 ViewResult і генерація представлень

Клас `ViewResult` є результатом дій контролера, що найбільш часто повертається. Він здійснює рендеринг представлення у веб-сторінку і повертає її у вигляді відповіді клієнту. Щоб повернути об'єкт `ViewResult` використовується метод `View`:

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        return View();
    }
}
```

Виклик методу `View` повертає об'єкт `ViewResult`. Потім `ViewResult` виробляє рендеринг певного представлення у відповідь. За замовчуванням контролер здійснює пошук представлення в проєкті за наступним шляхом:

`/Views/Імя_контролера/Імя_представлення.cshtml`

Згідно налаштувань, якщо подання не вказано явним чином, то як представлення використовуватиметься то, ім'я якого збігається з ім'ям дії контролера. Наприклад, вищеозначена дія `Index` за замовчуванням буде проводити пошук представлення `Index.cshtml` в папці `/ Views / Home /`.

Однак можна також задати ім'я представлення явним чином:

```
public class HomeController : Controller
{
    public ActionResult SomeMethod()
    {
        return View("Index");
    }
}
```

У підсумку як представлення буде вибрано /Views/Home/Index.cshtml. Ми також можемо повністю перевизначити шлях, за яким система буде шукати представлення:

```
public class HomeController : Controller
{
    public ActionResult SomeMethod()
    {
        return View("~/Views/Some/Index.cshtml");
    }
}
```

Правда, якщо такого шляху не виявиться, то додаток згенерує помилку.

3.3.9 Передача даних з контролера в представлення

Існують різні способи передачі даних з контролера в представлення. Один з них це використання об'єкта ViewData.ViewData представляє словник з пар «ключ-значення»:

```
public ActionResult SomeMethod()
{
    ViewData["Head"] = "Hello world!";
    return View("SomeView");
}
```

У цьому випадку в представленні Some View.cshtml можна отримати переданий рядок наступним чином:

```
@{ Layout = null; }
```

```
<!DOCTYPE html>
```

```
<html>
```

```
<head>
```

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

```
    <title>SomeView</title>
```

```
</head>
```

```
<body>
```

```
    <div>
```

```
        <h2>@ViewData["Head"]</h2>
```

```
    </div>
```

```
</body>
</html>
```

Ще один спосіб передачі даних представляє об'єкт `ViewBag`. Цей об'єкт дозволяє визначати різні властивості і присвоювати їм будь-яке значення. Так, ми могли б переписати попередній приклад наступним чином:

```
public ActionResult SomeMethod()
{
    ViewBag.Head = "Hello world!";
    return View("SomeView");
}
```

І таким же чином змінити представлення:

```
.....
<body>
  <div>
    <h2>@ViewBag.Head</h2>
  </div>
</body>
</html>
```

І не важливо, що спочатку об'єкт `ViewBag` не містить властивості `Head`, - вона визначається динамічно. При цьому властивості `ViewBag` можуть містити не тільки прості об'єкти типу `string` або `int`, але й складні дані. Так, у минулому прикладі ми додавали в об'єкт `ViewBag` список об'єктів моделі:

```
public ActionResult Index()
{
    IEnumerable<Book> books = db.Books;
    ViewBag.Books = books;
    return View();
}
```

Хоча `ViewData` і `ViewBag` і схожі, в той же час вони не повністю еквівалентні. Так, наприклад, не можна передавати динамічні значення з `ViewBag` в методи розширення в представленнях. Наприклад, ми не можемо написати `@Html.TextBox("name", ViewBag.Name)`, так як компілятор C# повинен знати тип кожного параметра під час компіляції, щоб вибрати потрібний метод розширення. У цьому випадку нам треба або використовувати `ViewData`: `@ Html.TextBox("name", ViewData ["Name"])`, або застосувати приведення типів: `@ Html.TextBox("name", (string) ViewBag.Name)`

І ще один спосіб пропонує об'єкт **`TempData`**. `TempData` являє собою словник, який зберігає пари «ключ-значення», як і `ViewData`, але його використання дещо відрізняється. `TempData` дозволяє зберігати передане значення протягом всього поточного запиту. Використання `TempData` аналогічно роботі з `ViewData`.

3.3.10 Переадресація і відправка кодів статусу і помилок

Існує два види переадресації: тимчасова і постійна. І залежно від виду переадресації при її виконанні сервер посилає браузерам один з двох кодів HTTP:

- статусний код 301 представляє постійну переадресацію. При даному типі переадресації передбачається, що запитуваний документ остаточно переміщений в інше місце. Після отримання даного статусного коду браузер може автоматично налаштувати запити на новий ресурс, навіть, якщо старий ресурс з часом перестане застосовувати переадресацію. Тому даний спосіб використовувати небажано.
- статусний код 302 представляє тимчасову переадресацію. При тимчасовій переадресації вважається, що запитуваний документ тимчасово переміщений на іншу сторінку.

В обох випадках для переадресації використовуватиметься об'єкт `RedirectResult`, однак метод, який повертає даний об'єкт, буде відрізнятися.

Для тимчасової переадресації застосовується метод **`Redirect`**:

```
public RedirectResult SomeMethod()
{
    return Redirect("/Home/Index");
}
```

Для постійної переадресації подібним чином використовується метод **`RedirectPermanent`**:

```
public RedirectResult SomeMethod()
{
    return RedirectPermanent("/Home/Index");
}
```

При цьому нам не обов'язково повертати з методу об'єкт `RedirectResult`. Нерідко виникає ситуація, коли в залежності від деяких умов потрібно направити користувача за однією адресою, або переадресувати на інший ресурс. Типова ситуація: авторизація користувача - якщо він авторизований, то йому відображається вибрана веб-сторінка, а якщо ні, то він перенаправляється на сторінку для логіна. наприклад:

```
public ActionResult Buy(int id)
{
    if (id > 3)
    {
        return Redirect("/Home/Index");
    }
    ViewBag.BookId = id;
    return View();
}
```

Якщо в якості параметра буде передано число більше 3, то відбудеться редирект на `Home/Index`. В інших випадках користувачеві буде повертатися представлення.

Ще один клас для створення переадресації - `RedirectToRouteResult` - дозволяє виконати більш детальну настройку перенаправлень. Він повертається двома методами: `RedirectToAction` і `RedirectToRoute`.

Метод `RedirectToRoute` дозволяє провести перенаправлення за визначеним маршрутом всередині домену:

```
public RedirectToRouteResult SomeMethod()
{
    return RedirectToRoute(new { controller="Home", action="Index" });
}
```

Метод `RedirectToAction` дозволяє перейти до певної дії певного контролера. Він також дозволяє задати передані параметри:

```
public RedirectToRouteResult SomeMethod()
{
    return RedirectToAction("Square", "Home", new { a=10,h=12 });
}
```

Методи `Redirect/RedirectToAction` представляють тимчасову переадресацію. Але вони мають свої двійники для створення постійної переадресації: `RedirectPermanent/ RedirectToActionResultPermanent`. Їх дія аналогічно, різниця лише в тому, що вони відправляють браузеру статусний код 301. Проте методи `RedirectPermanent` і `RedirectToActionResultPermanent` не рекомендується використовувати, а якщо і використовувати, то з обережністю. Так як неправильно налаштована постійна переадресація може погіршити позиції в пошукових або сприяти повного випадання сайту з пошуку.

3.3.11 Відправка помилок і статусних кодів

Іноді виникає необхідність відправити повідомлення про помилки при доступі до того чи іншого ресурсу. Зазвичай, якщо ресурс недоступний, `mvc-фреймворк` автоматично відреагує на цю ситуацію, відправивши відповідний статусний код. Але в деяких ситуаціях нам потрібно більш тонко реагувати на отриманий запит. Наприклад, у нас є контент, до якого встановлені вікові обмеження. Ми дивимося введений вік, і якщо він потрапляє під обмеження, ми можемо вислати статусний код помилки:

```
public ActionResult Check(int age)
{
    if (age < 21)
    {
        return new HttpStatusCodeResult(404);
    }
    return View();
}
```

Подібним чином ми можемо послати браузеру будь-який інший статусний код.

В якості альтернативи також можна повертати об'єкт `HttpNotFoundResult` за допомогою методу `HttpNotFound`:

```
public ActionResult Check(int age)
{
    if (age < 21)
    {
        return HttpNotFound();
    }
    return View();
}
```

І ще один клас, призначений для відправки статусних кодів - клас `HttpUnauthorizedResult`. Він сповіщає користувача, що той не має права доступу до ресурсу, відправляючи браузеру статусний код 401:

```
public ActionResult Check(int age)
{
    if (age < 21)
    {
        return new HttpUnauthorizedResult();
    }
    return View();
}
```

3.3.12 Відправка файлів в ASP.NET MVC 5

Для відправки клієнтові файлів призначений `FileResult`. Однак так як це абстрактний клас, то фактично ми матимемо справу з його спадкоємцями:

FileContentResult: відправляє клієнту масив байтів, лічений з файлу

FilePathResult: представляє просту відправку файлу безпосередньо з сервера

FileStreamResult: даний клас створює потік - об'єкт `System.IO.Stream`, за допомогою якого зчитує і відправляє файл клієнту. У всіх трьох випадках для відправки файлів застосовується метод `File`, який і повертає об'єкт `FileResult`. Тільки в залежності від обраного способу використовується відповідна перевантажена версія цього методу.

Щоб відправити файл з файлової системи (тобто використання об'єкта `FilePathResult`), нам треба вказати в методі `File` три параметри: шлях до файлу на стороні сервера, тип вмісту і ім'я файлу для приймаючої сторони (ім'я файлу необов'язково, і можна обійтися в принципі тільки двома параметрами).

```
public FileResult GetFile()
{
    // Путь к файлу
    string file_path = Server.MapPath("~/Files/PDFIcon.pdf");
    // Тип файла - content-type
    string file_type="application/pdf";
    // Ім'я файлу – необов'язково
```

```

string file_name = "PDFIcon.pdf";
return File(file_path,file_type,file_name);
}

```

Передбачається, що у нас в проєкті є папка Files, в якій лежить файл PDFIcon.pdf. Метод Server.MapPath дозволяє побудувати повний шлях до ресурсу з каталогу в проєкті. Але також можна використовувати і абсолютні шляхи, звертаючись до будь-якого файлу у файлової системи, наприклад, string file_path = @"C:\Book\PDFIcon.pdf";

І, при зверненні, наприклад, за шляхом Home/GetFile нам буде запропоновано зберегти даний файл на локальному комп'ютері.

Схожим чином працюють і класи FileContentResult, тільки замість імені файлу в методі File вказується масив байтів, в який був зчитаний файл:

```

// Відправка масиву байтів
public FileResult GetBytes()
{
    string path = Server.MapPath("~/Files/PDFIcon.pdf");
    byte[] mas = System.IO.File.ReadAllBytes(path);
    string file_type = "application/pdf";
    string file_name = "PDFIcon.pdf";
    return File(mas, file_type, file_name);
}

```

І якщо ми хочемо повернути об'єкт FileStreamResult, то в якості першого аргументу в методі File має йти об'єкт Stream для файлу, що відправляється:

```

// Відправлення потоку
public FileResult GetStream()
{
    string path = Server.MapPath("~/Files/PDFIcon.pdf");
    // Об'єкт Stream
    FileStream fs = new FileStream(path, FileMode.Open);
    string file_type = "application/pdf";
    string file_name = "PDFIcon.pdf";
    return File(fs, file_type, file_name);
}

```

3.3.13 Контекст запиту HttpContext. Cookie. Сесії

Нерідко для обробки запиту потрібно інформацію про контекст запиту: який у користувача браузер, ір-адреса, з якої сторінки або сайту користувач потрапив до нас. І ASP.NET MVC дозволяє отримати цю інформацію, використовуючи об'єкт **HttpContext**.

Хоча в контролері ми також можемо звернутися до об'єкта ControllerContext, який має властивість HttpContext і по суті надає доступ до тієї ж функціональності та інформації. Але в той же час між ними є деякі відмінності. Об'єкт HttpContext описує дані конкретного http-запиту, який

обробляється додатком. А `ControllerContext` описує дані http-запиту безпосередньо по відношенню до даного контролеру.

Вся інформація про контекст запиту доступна нам через властивості об'єкта `HttpContext`. Так, всі дані запиту містяться у властивості `Request`. **`HttpContext.Request`** представляє об'єкт класу, успадкованого від `HttpRequestBase`, і тому містить усі його властивості. Розглянемо деякі з них:

- Отримання браузера користувача: `HttpContext.Request.Browser`
- Іноді просто браузера недостатньо, тоді можна звернутися до агента користувача: `HttpContext.Request.UserAgent`
- Отримання url запиту: `HttpContext.Request.RawUrl`
- Отримання IP-адреси користувача: `HttpContext.Request.UserHostAddress`
- Отримання реферера:
`HttpContext.Request.UrlReferrer==null?""`:
`HttpContext.Request.UrlReferrer.AbsoluteUri`

Так як реферер може бути не визначений, то спочатку дивимося, чи не рівний він `null`

Наприклад:

```
public string Index()
{
    string browser = HttpContext.Request.Browser.Browser;
    string user_agent = HttpContext.Request.UserAgent;
    string url = HttpContext.Request.RawUrl;
    string ip = HttpContext.Request.UserHostAddress;
    string referrer = HttpContext.Request.UrlReferrer == null ? "" :
    HttpContext.Request.UrlReferrer.AbsoluteUri;
    return "<p>Browser: " + browser+"</p><p>User-Agent: "+user_agent+"</p><p>Url запроса: "+url+
    "</p><p>Реферер: " + referrer + "</p><p>IP-адрес: "+ip+"</p>";
}
```

3.3.14 Відправка відповіді

Якщо `HttpContext.Request` містить інформацію про запит, то властивість `HttpContext.Response` управляє відповіддю. Воно представляє об'єкт `HttpResponse`, який передає на сторону клієнта деякі значення: куки, службові заголовки, сам відповідь у вигляді коду html. Наприклад, встановимо кодування відповіді: `HttpContext.Response.Charset = "iso-8859-2"`;

Методи об'єкта `HttpResponse` дозволяють управляти відповіддю. Наприклад, метод **`AddHeader`** дозволяє додати до відповіді додатковий заголовок.

Крім того, нам необов'язково викликати метод `View` в дії контролера, щоб відправити клієнту відповідь на запит. Ми цілком можемо скористатися методом **`HttpContext.Response.Write`**:

```
public string ContextData()
```

```
{
    HttpContext.Response.Write("<h1>Hello World</h1>");

    string user_agent = HttpContext.Request.UserAgent;
    string url = HttpContext.Request.RawUrl;
    string ip = HttpContext.Request.UserHostAddress;
    string referrer = HttpContext.Request.UrlReferrer == null ? "" :
    HttpContext.Request.UrlReferrer.AbsoluteUri;
    return "<p>User-Agent: "+user_agent+"</p><p>Url запиту: "+url+
    "</p><p>Реферер: " + referrer + "</p><p>IP-адреса: "+ip+"</p>";
}
```

Або іншим чином:

```
public void ContextData()
{
    HttpContext.Response.Write("<h1>Hello World</h1>");
}
```

Метод `HttpContext.Response.Write` тут додає в потік деякий вміст даних, переданий як параметр. Але в реальності простіше використовувати методи, генеруючі об'єкти `ActionResult`, наприклад, представлення.

3.3.15 Визначення користувача.

Також об'єкт `HttpContext` містить інформацію про користувача у властивості `HttpContext.User`:

```
bool isAdmin = HttpContext.User.IsInRole ("admin"); // Визначити, чи належить
//користувач до адміністраторів
bool isAuthenticated = HttpContext.User.Identity.IsAuthenticated; // Аутентифікований //користувач
string login = HttpContext.User.Identity.Name; // Логін авторизованого //користувача
```

3.3.16 Робота з Cookie

Щоб отримати cookies, нам треба скористатися властивістю `HttpContext.Request.Cookies`:

```
string id =HttpContext.Request.Cookies["id"].Value;
```

В даному випадку, якщо у нас встановлена на стороні клієнта куки "id", то ми отримаємо її значення. Однак перш ніж отримувати значення куки, їх треба встановити. Для установки значення cookies ми можемо використовувати властивість `HttpContext.Response`. Наприклад, встановимо в cookies значення "id":

```
HttpContext.Response.Cookies["id"].Value = "ca-4353w";
```

3.3.17 Сесії

Сесії також, як і cookies, використовуються для зберігання даних, які можна отримати в будь-якому місці програми. Для роботи з ними використовується об'єкт Session. Наприклад, в одному методі контролера ми можемо встановити значення сесії:

```
public ActionResult Index()
{
    Session["name"] = "Cat";
    return View();
}
```

А отримати можна в іншому методі:

```
public string GetName()
{
    var val = Session["name"];
    return val.ToString();
}
```

Якщо ми хочемо видалити значення сесії для ключа name, ми можемо просто присвоїти значення null: `Session ["name"] = null;`

3.3.18 Асинхронні методи

Одним з ключових нововведень останніх версій фреймворку .NET стала асинхронність. Хоча фреймворк і раніше дозволяв використовувати асинхронні методи, але з появою бібліотеки Task Parallel Library робота з асинхронним кодом була гранично спрощена, а сам формат роботи змінився. Були додані нові можливості по створенню асинхронних методів з використанням нових ключових слів, таких як `async` і `await`.

При створенні нового контролера ми в настройках вже можемо вказати, як нам потрібен контролер - синхронний або асинхронний. За замовчуванням Visual Studio додає в проект стандартні контролери, методи яких, як правило, повертають об'єкт `ActionResult`. Але якщо ми при додаванні контролера в папку `Controllers` виберемо тип `MVC5 Controller with views, using Entity Framework`, то у вікні налаштування нового контролера спеціальне поле дозволить нам вказати, що новий контролер буде містити асинхронні методи:

The screenshot shows the 'Add Controller' dialog box. The 'Controller name' field contains 'StoreController'. The 'Use async controller actions' checkbox is checked. The 'Model class' and 'Data context class' fields are empty. The 'Views' section has three unchecked options: 'Generate views', 'Reference script libraries', and 'Use a layout page'. There is a text box for the layout page with an empty field and an ellipsis button. A note below the text box says '(Leave empty if it is set in a Razor _viewstart file)'. At the bottom right are 'Add' and 'Cancel' buttons.

Рисунок 3.12 - Вікно налаштувань контролера

Для чого потрібні взагалі асинхронні методи в контролерах? Асинхронні методи дозволяють оптимізувати продуктивність програми і призначені насамперед для обробки таких запитів, які займають або можуть зайняти досить тривалий час, наприклад, звернення до бази даних або звернення до зовнішнього мережевого ресурсу для отримання великої порції даних. Застосування асинхронних методів дозволяє додатку паралельно з виконанням асинхронного коду виконувати також інші запити.

Щоб зрозуміти відмінність між синхронними і асинхронними методами, розглянемо, як IIS обробляє вхідні запити. Веб-сервер підтримує пул потоків, які обслуговують запити. При зверненні користувача до веб-ресурсу IIS виділяє потік з пулу для обслуговування даного запиту. І поки даний потік не обробить призначений для нього запит, інші запити він обробляти не може.

Однак припустимо, що метод контролера в процесі обробки запиту повинен виконати запит до іншого ресурсу або до бази даних. Запит до мережевого ресурсу або БД сам по собі може зайняти деякий час. При синхронній обробці потік, що обробляє запит, тимчасово блокується, поки мережевий ресурс або БД не повернуть потрібні нам дані.

Якщо обробка запиту блокується дуже довго, то IIS починає задіяти для обслуговування інших вхідних запитів нові потоки. Однак є обмеження на загальну кількість потоків. Коли кількість потоків досягає межі, то знову вхідні запити поміщаються в чергу очікування. Однак і тут є обмеження на кількість запитів в черзі. І коли ця кількість перевищує межу, то IIS просто відхиляє всі інші запити за допомогою статусного коду 503 (Service Unavailable).

При асинхронній обробці потік не чекає, поки БД поверне йому дані, а починає обробляти запит від іншого користувача. А коли, нарешті, з мережевого ресурсу або БД прийдуть потрібні дані, потік повертається до обробки раніше оброблюваного запиту в звичайному режимі.

Перейдемо безпосередньо до програмного коду. Для створення асинхронних методів використовуються модифікатори **async** і **await**, які дозволяють виконувати тривалі операції без блокування основного потоку.

Порівняємо на прикладі виклик синхронного та асинхронного методу:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using AsyncContollers.Models;
using System.Threading.Tasks;
using System.Data.Entity;

namespace AsyncContollers.Controllers
{
    public class HomeController : Controller
    {
        BookContext db = new BookContext();

        public ActionResult Index()
        {
            IEnumerable<Book> books = db.Books;
            ViewBag.Books = books;
            return View();
        }

        // асинхронний метод
        public async Task<ActionResult> BookList()
        {
            IEnumerable<Book> books = await Task.Run(() => db.Books);
            ViewBag.Books = books;
            return View("Index");
        }
    }
}
```

Обидва методи виконують одну й ту ж операцію - вилучення даних з БД і отримують ідентичні результати. Але якщо перший синхронний метод `Index` являє звичний для нас запис, то асинхронний метод `BookList` вже виглядає інакше.

Цей метод повертає не об'єкт **ActionResult**, а об'єкт **Task<ActionResult>**. `Task` представляє асинхронну операція, виконувану тривалий час.

Крім того, щоб позначити метод як асинхронний, перед типом, повертається ставиться ключове слово **async**.

Третім ключовим моментом є використання ключового слова **await**. Воно застосовується в асинхронних методах, щоб призупинити виконання цього методу до тих пір, поки очікуване завдання не завершиться. У нашому випадку таким завданням є отримання даних з БД.

Також слід враховувати, що **await** використовується з методами, які повертають об'єкт Task. Тому отримання даних їх БД ми обгорнули в такий вираз:

```
await Task.Run (() => db.Books) ;
```

Метод Task.Run як параметр отримує деяку дію у вигляді лямбда-виразу. А цією дією і є вилучення даних з БД: db.Books.

Додавання асинхронності відбулося по всій технології .NET. Зокрема, в нову версію **Entity Framework 6** також була додана асинхронність на рівні окремих методів, які ми можемо застосовувати при операціях над даними. Наприклад, попередній приклад асинхронного методу BookList ми можемо переписати таким чином:

```
public async Task<ActionResult> BookList()
{
    ViewBag.Books = await db.Books.ToListAsync();
    return View("Index");
}
```

Метод db.Books.ToListAsync() повертає список об'єктів. Entity Framework 6 пропонує ще ряд асинхронних методів для роботи з БД, з якими ми пізніше ознайомимося. Коли слід використовувати асинхронні методи? У першу чергу їх переважно використовують при запитах до БД, до зовнішньому мережевих ресурсів, проте в кінцевому рахунку, що краще використовувати - синхронність або асинхронність, вирішує вже сам розробник виходячи з конкретного завдання.

3.4 Представлення (View)

3.4.1 Основні положення

Хоча робота додатку MVC управляється головним чином контролерами, але безпосередньо користувачеві додаток доступний у вигляді представлення, яке і формує зовнішній вигляд програми. У ASP.NET MVC5 представлення - це файли з розширенням cshtml, які містять код користувача інтерфейсу в основному на мові html. Стандартне представлення :

```
@{
    Layout = null;
}

<!DOCTYPE html>

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

```

<title>SomeView</title>
</head>
<body>
  <div>
    <h2>@ViewBag.Message</h2>
  </div>
</body>
</html>

```

Хоча представлення містить головним чином, код html, воно не є html-сторінкою. При компіляції додатку на основі необхідного представлення спочатку генерується клас на мові C#, а потім цей клас компілюється. Так, з вищенаведеного представлення буде генеруватися приблизно такий клас:

```

#pragma checksum "c:\users\hp\documents\visual studio
2013\Projects\MVC\BookStore\BookStore\Views\Home\SomeView.cshtml" "{ff1816ec-aa5e-4d10-
87f7-6f4963833460}" "1F05F4D370C9D00F8CBDFB8BD1F51D74189D0617"

```

```

namespace ASP {
    using System;
    using System.Collections.Generic;
    using System.IO;
    using System.Linq;
    using System.Net;
    using System.Web;
    using System.Web.Helpers;
    using System.Web.Security;
    using System.Web.UI;
    using System.Web.WebPages;
    using System.Web.Mvc;
    using System.Web.Mvc.Ajax;
    using System.Web.Mvc.Html;
    using System.Web.Optimization;
    using System.Web.Routing;
    using BookStore;

    public class _Page_Views_Home_SomeView_cshtml : System.Web.Mvc.WebViewPage<dynamic>
    {

        #line hidden

        public _Page_Views_Home_SomeView_cshtml() {
        }

        protected ASP.global_asax ApplicationInstance {
            get {
                return ((ASP.global_asax)(Context.ApplicationInstance));
            }
        }
    }
}

```

```

        public override void Execute() {
BeginContext("~/Views/Home/SomeView.cshtml", 0, 2, true);

WriteLiteral("\r\n");

EndContext("~/Views/Home/SomeView.cshtml", 0, 2, true);

        #line 2 "c:\users\hp\documents\visual studio
2013\Projects\MVC\BookStore\BookStore\Views\Home\SomeView.cshtml"

        Layout = null;

        #line default
        #line hidden
BeginContext("~/Views/Home/SomeView.cshtml", 27, 48, true);
WriteLiteral("\r\n\r\n<!DOCTYPE html>\r\n\r\n<html>\r\n<head>\r\n  <meta");
EndContext("~/Views/Home/SomeView.cshtml", 27, 48, true);
BeginContext("~/Views/Home/SomeView.cshtml", 75, 16, true);
WriteLiteral(" name=\"viewport\"");
EndContext("~/Views/Home/SomeView.cshtml", 75, 16, true);
BeginContext("~/Views/Home/SomeView.cshtml", 91, 29, true);
WriteLiteral(" content=\"width=device-width\"");
EndContext("~/Views/Home/SomeView.cshtml", 91, 29, true);
BeginContext("~/Views/Home/SomeView.cshtml", 120, 74, true);
WriteLiteral(" />\r\n  <title>SomeView</title>\r\n</head>\r\n<body>\r\n  <div> \r\n    <h2>");
EndContext("~/Views/Home/SomeView.cshtml", 120, 74, true);
BeginContext("~/Views/Home/SomeView.cshtml", 195, 15, false);

        #line 15 "c:\users\hp\documents\visual studio
2013\Projects\MVC\BookStore\BookStore\Views\Home\SomeView.cshtml"
        Write(ViewBag.Message);

        #line default
        #line hidden
EndContext("~/Views/Home/SomeView.cshtml", 195, 15, false);
BeginContext("~/Views/Home/SomeView.cshtml", 210, 38, true);
WriteLiteral("</h2> \r\n  </div>\r\n</body>\r\n</html>\r\n");
EndContext("~/Views/Home/SomeView.cshtml", 210, 38, true);
        }
    }
}

```

Код, звичайно, не найбільш читабельний, особливо якщо невідомо, що роблять всі класи і методи, але тут ми можемо побачити, що при компіляції створюється клас, що успадковується від класу `System.Web.Mvc.WebViewPage<T>`, де `T` - це клас моделі, яка буде використовуватися. Але так як представлення не строго типізоване, тому замість імені класу моделі йде ключове слово `dynamic`. Всі дії даного класу укладені в методі `Execute`, в якому за допомогою методу `WriteLiteral` обробляються всі наявні в представлення

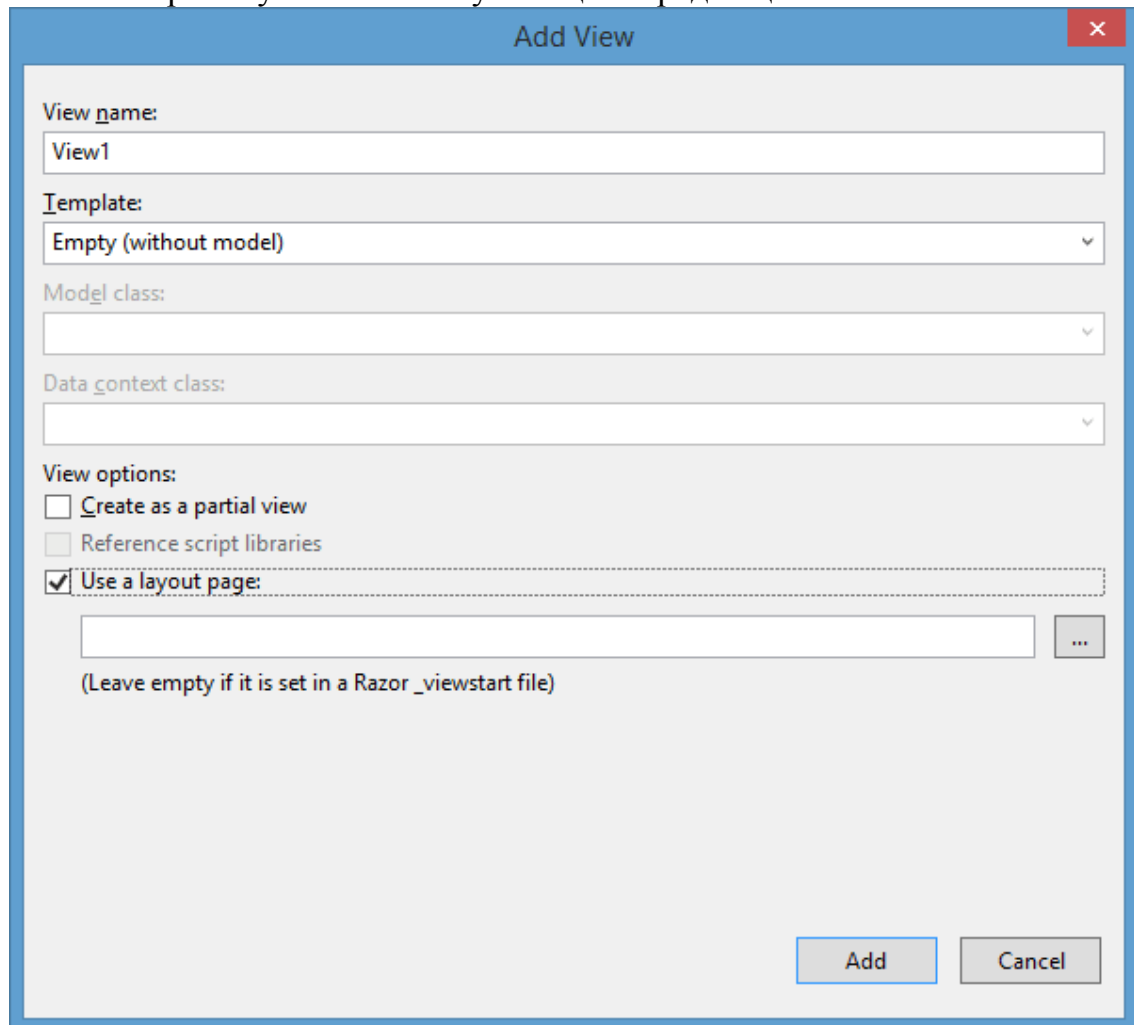
елементи розмітки html. Знайти згенеровані з представлень файли коду можна по шляху:

C:\Users\Імя_Логіна\AppData\Local\Temp\TemporaryASP.NETFiles\root.

Правда, всі папки мають зашифровані імена, тому щоб визначити потрібну папку програми, потрібно буде затратити деякий час на пошук. Крім того, самі генеровані файли коду також мають зашифровані імена. Наприклад, вищенаведений клас в нашому випадку має ім'я App_Web_nri53fza.1.cs і знаходиться в папці root\307f1c1d\36bbd4f.

3.4.2 Створення нового представлення

Для створення нового представлення виберемо у проекті папку Views і в ньому натиснемо правою кнопкою миші на підкаталог Home, і в списку виберемо пункт Add -> View ... (Представлення). У вікні додавання нового представлення пропонується налаштувати цілий ряд опцій:



The 'Add View' dialog box is shown with the following settings:

- View name:** View1
- Template:** Empty (without model)
- Model class:** (empty)
- Data context class:** (empty)
- View options:**
 - ☐ Create as a partial view
 - ☐ Reference script libraries
 - ☒ Use a layout page
- Layout page:** (empty text box with a button to browse)

(Leave empty if it is set in a Razor _viewstart file)

Рисунок 3.13 - Форма налаштувань представлень.

Розберемо всі ці налаштування:

View Name: ім'я нового подання. Після створення йому автоматично буде присвоюватися розширення cshtml.

Template: шаблон нового подання. Ми можемо вибрати з наступного списку шаблонів:

Типи шаблонів представлень:

Empty (without model): створюється порожнє представлення з початковою розміткою

Empty: також створюється порожнє представлення, але тепер нижче ми можемо вибрати модель, яка буде підключена в поданні за допомогою директивимodel

Create: генерується представлення з формою для створення нових об'єктів моделі. У цій формі для кожної властивості моделі створюється окреме поле

Delete: генерується представлення з формою для видалення об'єктів моделі. У цій формі для кожної властивості моделі створюється окреме поле

Details: генерується представлення, яке відображає значення всіх властивостей моделі

Edit: генерується представлення з формою для редагування наявних об'єктів моделі. У цій формі для кожної властивості моделі створюється окреме поле

List: створюється представлення, яке відображає всі об'єкти зі списку моделей у вигляді таблиці. Для генерації списку об'єктів в дане подання необхідно передавати з методу контролера значення типу IEnumerable<Тип_моделі>. Представлення також містить посилання на методи для виконання операцій створення/редагування/видалення.

Model class: при виборі однієї з попередніх опцій, крім опції Empty (without model), нам стає доступно це поле, в якому ми можемо вказати модель для типізації представлення. Таке представлення буде вважатися строго типізованим, тобто прив'язаним до одного класу моделі.

Data context class: також при виборі однієї з попередніх опцій, крім опції Empty (without model), стає доступним це поле, в якому можемо вибрати клас контексту даних

Create as a partial view: дозволяє створити часткове представлення

Reference Script Libraries: ця опція показує, чи буде представлення автоматично підключати стандартний набір бібліотек jQuery та інших файлів JavaScript.

Use a layout page: ця опція вказує, чи буде використовуватися майстер-сторінка або представлення буде самодостатнім. Після встановлення цієї опції стає доступним нижнє поле, в якому можна вибрати майстер-сторінку. Для бібліотеки Razor вказівка майстер-сторінки не є обов'язковою, якщо ви будете використовувати майстер-сторінку, визначену за замовчуванням у файлі _ViewStart.cshtml. Однак, якщо ви хочете перевизначити майстер-сторінку, то можете скористатися цією опцією.

3.4.3 Шляхи до файлів представлень

Всі представлення, що додаються, як правило, групуються за контролерам у відповідні папки в каталозі Views. Представлення, які ставляться до методів контролера Home, будуть перебувати в проекті в папці Views/Home. Однак при необхідності ми самі можемо створити в каталозі Views папку з довільним ім'ям, де будемо зберігати додаткові представлення, необов'язково пов'язані з певними методами контролера.

Щоб здійснити рендеринг представлення у вихідний потік, використовується метод View(). Якщо в цей метод не передається імені представлення, то за замовчуванням додаток буде працювати з тим представленням, ім'я якого збігається з ім'ям методу дії. Наприклад, наступний метод дії буде звертатися до подання Index.cshtml:

```
public ActionResult Index()
{
    IEnumerable<Book> books = db.Books;
    ViewBag.Books = books;
    return View();
}
```

Вказавши шлях до поданням явним чином, ми можемо перевизначити налаштування за замовчуванням:

```
public ActionResult Index()
{
    IEnumerable<Book> books = db.Books;
    ViewBag.Books = books;
    return View("~/Views/Some/SomeView.cshtml");
}
```

3.4.4 Синтаксис Razor

Стандартне представлення дуже схоже на звичайну веб-сторінку з кодом html. Однак воно також має вставки коду на C#, які починаються із знаку @. Цей знак використовується бібліотекою представлень Razor для переходу до коду на мові C#. Щоб зрозуміти суть роботи бібліотеки Razor і його синтаксису, спочатку подивимось, що представляють із себе бібліотеки представлень.

3.4.5 Бібліотека представлень

При виклику методу View контролер не виробляє рендеринг представлення і не генерує розмітку html. Контролер тільки готує дані і вибирає, яке представлення треба повернути в якості об'єкта ViewResult. Потім вже об'єкт ViewResult звертається до бібліотеки представлень для рендеринга представлення у вихідний результат.

Якщо раніше попередні версії ASP.NET MVC і Visual Studio за замовчуванням підтримували дві бібліотеки представлень - Web Forms і Razor, то зараз Razor в силу своєї простоти і легкості став єдиною бібліотекою

представлень за замовчуванням. Використання Razor дозволило зменшити синтаксис при виклику коду C #, зробити сам код більш "чистим".

Тут важливо розуміти, що Razor - це не нова мова, це лише спосіб рендеринга предсавлень, який має певний синтаксис для переходу від розмітки html до коду C#.

3.4.6 Основи синтаксису Razor

Використання синтаксису Razor характеризується тим, що перед виразом коду стоїть знак @, після якого здійснюється перехід до коду C#. Існують два типи переходів: до виразів коду і до блоку коду. Наприклад, перехід до виразів коду:

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

Razor автоматично розпізнає, що Name - це властивість об'єкта b. Також можна використовувати стандартні класи і методи, наприклад, виведемо поточний час:

```
<h3>@DateTime.Now.ToShortTimeString()</h3>
```

Застосування блоків коду аналогічно, тільки знак @ ставиться перед усім блоком коду, а бібліотека автоматично визначає, де цей блок коду закінчується:

```
@foreach (BookStore.Models.Book b in Model)
{
    <p>@b.Name</p>
}
```

Більше того ми можемо створювати блоки коду в представленні, створювати там змінні так само, як і у файлі коду C#:

```
{
    string head = "Hello world !";
    head = head + " Welcome to our site!";
}
<h3>@head</h3>
```

3.4.7 Строго типізовані представлення

У попередніх прикладах для передачі інформації з контролера в представлення використовувався об'єкт ViewBag:

```
@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/Buy/@b.Id">Купити</a></p></td>
    </tr>
}

```

Тут ми отримуємо доступ до елементів колекції, укладеної в ViewBag.Books, за допомогою змінної з ключовим словом var - тобто тип змінної не заданий явно і виводиться компілятором. Те ж саме ми могли б вказати тип моделі явно, застосувавши повне ім'я типу моделі:

```

@foreach (BookStore.Models.Book 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/Buy/@b.Id">Купити</a></p></td>
    </tr>
}

```

Хоча приклади з об'єктом ViewBag працюють нормально, але є й інший спосіб, іноді більш кращий, який полягає у використанні строго типізованих представлень. Подібні представлення дозволяють передавати дані не через об'єкт ViewBag, а безпосередньо в представлення через параметр методу View. Код методу контролера міг би виглядати так:

```

BookContext db = new BookContext();

public ActionResult Index()
{
    return View(db.Books);
}

```

Тепер, щоб зв'язати представлення з переданим параметром, треба додати в представлення директиву @model із зазначенням типу переданих даних. Оскільки books представляє тип IEnumerable <Book>, то подання буде виглядати так :

```

@model IEnumerable<BookStore.Models.Book>
@{
    Layout = "~/Views/Shared/_Layout.cshtml";
}
<div>
    <h3>Розпродаж 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>
    </table>

```



```

@foreach (BookStore.Models.Book b in Model)
{
|  |  |  |  |
| --- | --- | --- | --- |
| <p>@b.Name</p></td>  <p>@b.Author</p></td>  <p>@b.Price</p></td>  <p><a href="/Home/Buy/@b.Id">Купити</a></p></td> | | | |

}
</table>
</div>

```

Об'єкт Model представляє тип моделі, зазначеної в директиві @model, і буде зберігати передані з контролера дані. Але щоб не писати повністю ім'я типу моделі, ми можемо імпортувати простір імен у представленні:

```

@using BookStore.Models
@model IEnumerable<Book>

```

.....

Крім того, ми можемо автоматично створити строго типізоване представлення, вказавши в діалоговому вікні при створенні представлення відповідні параметри:

View name:
BookList

Template:
List

Model class:
Book (BookStore.Models)

Data context class:
BookContext (BookStore.Models)

View options:
☐ Create as a partial view
☐ Reference script libraries
☒ Use a layout page:
~/Views/Shared/_Layout.cshtml
(Leave empty if it is set in a Razor _viewstart file)

Add Cancel

Рисунок 3.14 - Діалогове вікно створення представлення

Для цього в полі Template треба вибрати будь-який інший шаблон, крім Empty (without model), і після цього вказати потрібний клас моделі і контексту даних. Якщо ми виберемо шаблон List, то автоматично згенероване представлення буде за своїм функціоналом ідентичне раніше розглянутому представленню з виведенням книг.

3.4.8 Майстер-сторінки

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

За замовчуванням при створенні нового проекту ASP.NET MVC5 в проект уже додається майстер-сторінка під назвою `_Layout.chnml`, яку можна знайти в каталозі `Views/Shared`.

```
<!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("Main", "Index", "Home")</li>
    </ul>
  </nav>
  @RenderBody()
</body>
</html>
```

На перший погляд це звичайне представлення за одним винятком: тут використовується метод `RenderBody()`, який є заступником і на місце якого потім будуть підставлятися інші представлення, що використовують дану майстер-сторінку. У підсумку ми зможемо легко встановити для представлення веб-додатку однаковий стиль.

Щоб у представленні використовувати майстер-сторінку, нам треба в секції `Layout` вказати шлях до потрібної майстер-сторінки. Наприклад, у представленні `Index.cshhtml` можна визначити майстер-сторінку так:

```
@{
  Layout = "~/Views/Shared/_Layout.cshhtml";
}
```

Якщо ми не використовуємо майстер-сторінку, то вказуємо `Layout = null`; Також необов'язково для всіх представлень використовувати одну й ту ж майстер-сторінку. Можна визначити кілька майстер-сторінок, наприклад, одну для форуму, одну для блогу і т.д., і в залежності від розділу сайту підключати потрібну. Додаються в проект вони також як і звичайні представлення. Майстер-сторінка може мати кілька секцій, куди представлення можуть помістити свій вміст. Наприклад, додамо до майстер-сторінки секцію `footer`:

```
<!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("Main", "Index", "Home")</li>
    </ul>
  </nav>
  @RenderBody()
  <footer>@RenderSection("Footer")</footer>
</body>
</html>

```

Тепер при запуску попереднього представлення Index ми отримаємо помилку, так як секція Footer не визначена. За замовчуванням представлення повинно передавати зміст для кожної секції майстер-сторінки. Тому додамо вниз представлення Index секцію footer. Ми можемо це зробити за допомогою виразу section:

```

@ {
Layout = "~/Views/Shared/_Layout.cshtml";
}
<!--додатковий вміст -->
@section Footer {
    Copyright. Syte Corp. 2012.
}

```

Але при такому підході, якщо у нас є багато представлень, а нам потрібно визначити нову секцію на майстер-сторінці, нам доведеться змінити всі наявні представлення, що не зручно. У цьому випадку ми можемо скористатися одним з варіантів гнучкої настройки секцій. Перший варіант полягає у використанні перевантаженої версії методу RenderSection, яка дозволяє вказати, що дану секцію не обов'язково визначати в представленні. Щоб відзначити секцію Footer в якості необов'язкової, треба передати в метод в якості другого параметра значення false:

```

<footer>@RenderSection("Footer", false)</footer>

```

Другий варіант дозволяє задати вміст секції за замовчуванням, якщо дана секція не визначена в представленні:

```

<footer>
  @if (IsSectionDefined("Footer")) {
    @RenderSection("Footer")
  }
  else
  {
    <span>Вміст елемента footer за замовчуванням.</span>
  }
</footer>

```

```
}  
</footer>
```

3.4.9 ViewStart

Якщо у нас в проєкті 2-3 представлення, ми можемо змінити вручну для кожного опис майстер-сторінки в секції Layout, якщо буде необхідно використовувати іншу майстер-сторінку. Але якщо у нас багато представлень, то це робити буде не дуже зручно.

Для більш гнучкого налаштування представлень призначена сторінка `_ViewStart.cshtml`. Код цієї сторінки виконується до коду будь-якого з представлень, розташованих у тому ж каталозі. Даний файл послідовно застосовується до кожного представлення, що знаходиться в одному каталозі.

При створенні проєкту ASP.NET MVC5 в каталог Views вже за замовчуванням додається файл `_ViewStart.cshtml`. Цей файл визначає майстер-сторінку, використовувану за замовчуванням:

```
@{  
    Layout = "~/Views/Shared/_Layout.cshtml";  
}
```

Даний код виконується для будь-якого іншого коду, визначеного у представленні, тому з інших представлень ми можемо видалити секцію Layout. Якщо ж представлення повинно використовувати іншу майстер-сторінку, то ми повинні перевизначити властивість Layout, дописуючи його визначення в початок представлення.

3.4.10 Часткові представлення

Крім звичайних представлень метод дії може також повертати часткові представлення. Їх відмінною особливістю є те, що їх можна вбудовувати в інші звичайні представлення. Часткові представлення можуть використовуватися також як і звичайні, проте найбільш зручною областю їх використання є рендеринг результатів AJAX-запиту.

За рендеринг часткових представлення відповідає об'єкт `PartialViewResult`, який повертається методом `PartialView`. Отже, визначимо в контролері нову дію `Partial`:

```
public ActionResult Partial()  
{  
    ViewBag.Message = "Це часткове представлення."  
    return PartialView();  
}
```

Тепер додамо нове представлення `Partial.cshtml`. Для цього при створенні представлення в налаштуваннях вкажемо, що воно буде частковим:

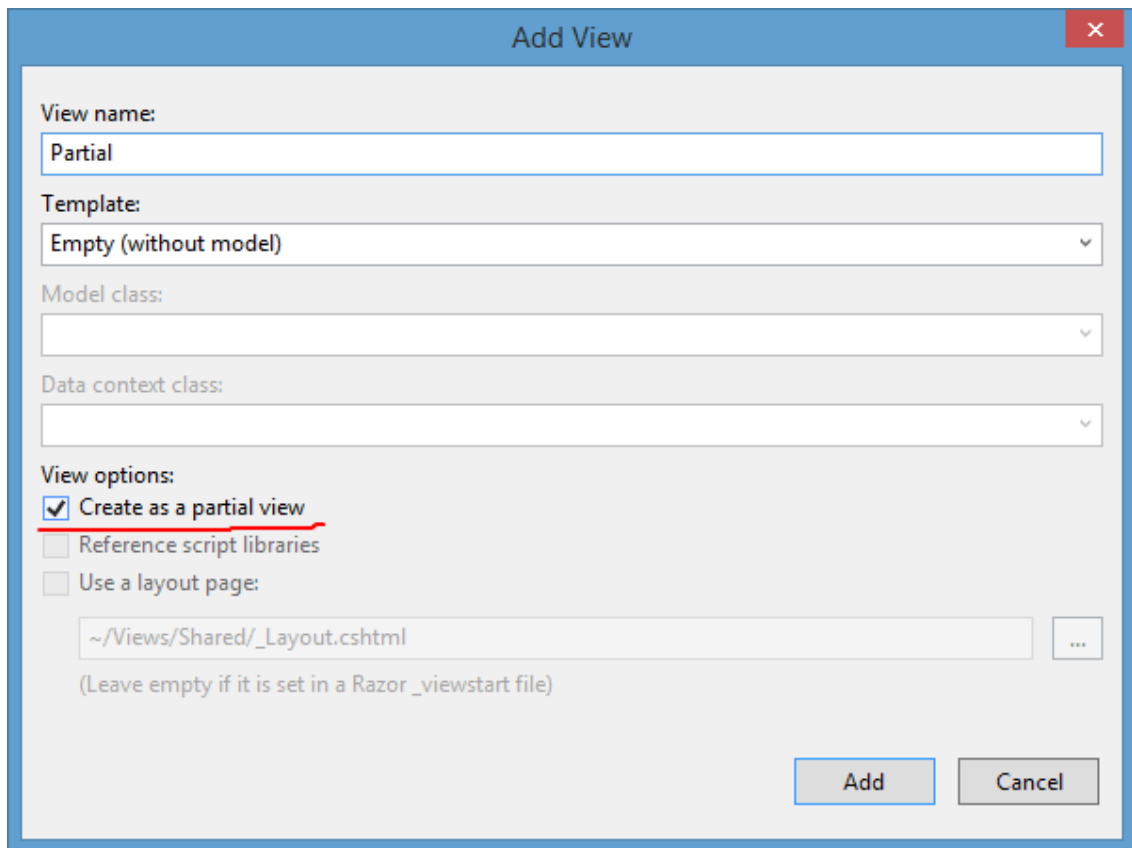


Рисунок 3.15 - Вікно налаштувань для нових представлень

Після цього в проект буде додано порожній файл часткового представлення. За своїм змістом воно схоже на звичайне представлення, тільки для нього не можна визначити майстер-сторінку. Отже, додамо в часткове представлення наступний рядок:

```
<h2> @ViewBag.Message </ h2>
```

Після цього ми можемо звернутися до дії Partial, як до звичайної дії контролера, і воно нам поверне часткове представлення.

Але сенс застосування часткових представлень полягає не в цьому, інакше вони б ніяк не відрізнялися від звичайних. Тому тепер вбудуємо його в якесь інше представлення. Для цього нам треба застосувати в будь-якому місці звичайного представлення Html.Partial:

```
@Html.Partial ("Partial")
```

У даному випадку в якості параметра ми вказуємо ім'я часткового представлення без розширення файлу. Але треба враховувати, що передати ViewBag.Message з методу Partial, як у попередньому прикладі, ми вже не зможемо. І якщо ми все таки хочемо передати його в часткове представлення, то нам треба буде передати його з методу контролера, пов'язаного з головним поданням. Тобто якщо ми

використовуємо часткове представлення у поданні Index.cshtml, то в методі Index ми можемо написати:

```
public ActionResult Index()
{
    ViewBag.Message = "Це виклик часткового представлення із звичайного";
    return View();
}
```

І, таким чином, повідомлення під ViewBag.Message буде передано як головному, так і частковим представленням. Крім Html.Partial часткове представлення можна вбудувати за допомогою іншого елемента - Html.RenderPartial. Він також приймає ім'я представлення, однак він використовується не в рядкових виразах коду Razor, а в блоці коду - тобто обрамляється фігурними дужками:

```
@ {Html.RenderPartial ("Partial");}
```

Ще одна відмінність між двома способами полягає в тому, що Html.RenderPartial безпосередньо пише дані у вихідний потік, тому може працювати трохи швидше, ніж Html.Partial. Також як і у випадку зі звичайними представленнями, ми можемо створювати строго типізовані часткові представлення, вказавши в шапці файлу директиву @model:

```
@model IEnumerable<string>
<h2>Список учнів</h2>
<ul>
@foreach (string t in Model)
{
    <li>@t</li>
}
</ul>
```

Тоді ми можемо викликати це подання так: @Html.Partial ("Partial", new string [] { "Іванюк", "Петренко", "Василюк", "Козлюк" })

3.4.11 HTML-хелпери

Як ми побачили з минулих прикладів, представлення використовують розмітку html для візуалізації вмісту. Однак фреймворк ASP.NET MVC володіє також таким інструментом як HTML-хелпери, що дозволяють генерувати html-код.

Рядкові хелпери

Рядкові хелпери схожі на звичайні визначення методів на мові C#, тільки починаються з тега @helper. Наприклад, створимо в представленні хелпер для виведення назв книг у вигляді списку:

```
@helper BookList(IEnumerable<BookStore.Models.Book> books)
{
    <ul>
        @foreach (BookStore.Models.Book b in books)
        {
            <li>@b.Name</li>
        }
    </ul>
}
```

Даний хелпер ми можемо визначити в будь-якому місці представлення. І також в будь-якому місці представлення ми можемо його використовувати, передаючи в нього об'єкт `IEnumerable <BookStore.Models.Book>`:

```
<h3>Список книг</h3>
@BookList(ViewBag.Books)
<!--або якщо використовується строго типізоване представлення -->
@BookList(Model)
```

Рядкові html-хелпери зручно використовувати, якщо необхідно створити один метод, який передбачається використовувати в поданні багаторазово. Наприклад:

```
@helper CreateList(string[] all)
{
    <ul>
        @foreach (string s in all)
        {
            <li>@s</li>
        }
    </ul>
}
@{
    string[] cities = new string[] { "Лондон", "Париж", "Київ" };
}
@{
    string[] countries = new string[] { "Великобританія", "Франція", "Україна" };
}
<h3>Міста</h3>
@CreateList(cities)
<br />
<h3>Країни</h3>
@CreateList(countries)
```

При відсутності подібного хелпера, нам би довелося по суті дублювати один і той же html-код для створення списку. Однак цей хелпер ще досить простий, а якщо нам доводиться створювати значно більш складні, але однотипні розмітки html, тоді хелпери виявляться більш корисними.

Даний підхід має один недолік - якщо хелпер дуже об'ємний, то він може сильно захаращувати розмітку представлення. І в цьому випадку його краще

винести в окремий файл коду. Модифікуємо попередній приклад. Для цього нам треба створити новий клас з методом розширення - тобто таким методом, який розширює функціональність вже існуючих класів. А ці класи вказуються як перший параметр методу. Отже, створимо в проєкті нову папку **Helpers** і додамо в неї новий клас **ListHelper**:

```
using System;
using System.Web;
using System.Web.Mvc;
using System.Linq;

namespace BookStore.Helpers
{
    public static class ListHelper
    {
        public static MvcHtmlString CreateList(this HtmlHelper html, string[] items)
        {
            TagBuilder ul = new TagBuilder("ul");
            foreach (string item in items)
            {
                TagBuilder li = new TagBuilder("li");
                li.SetInnerText(item);
                ul.InnerHtml += li.ToString();
            }
            return new MvcHtmlString(ul.ToString());
        }
    }
}
```

У новому класі хелпера визначений один статичний метод **CreateList**, що приймає в якості першого параметра об'єкт, для якого створюється метод. Оскільки цей метод розширює функціональність **html**-хелперів, які представляють клас **HtmlHelper**, то саме об'єкт цього типу і передається в даному випадку в якості першого параметра. Другий параметр методу **CreateList** - масив рядків-значень, які потім будуть виводитися в списку.

У самому методі за допомогою об'єкта **TagBuilder** конструюється стандартний елемент **html** - елемент **ul**. При обході масиву всі рядкові значення обгортаються в тег **li** та додаються до списку. І на виході повертається повноцінний елемент **ul**.

Клас **TagBuilder** має ряд членів, які можна використовувати при такому підході:

- Властивість **InnerHtml** дозволяє встановити або отримати вміст тега у вигляді рядка.
- Метод **MergeAttribute(string,string,bool)** дозволяє додати до елементу один атрибут. Для отримання всіх атрибутів можна використовувати колекцію **Attributes**.
- Метод **SetInnerText (string)** встановлює текстовий вміст всередині елементу.

- Метод **AddCssClass(sting)** додає клас css до елементу.

Після створення нового хелпера ми його можемо використовувати в представленні. Перепишемо попередній приклад наступним чином:

```
@{
    string[] cities = new string[] { "Лондон", "Париж", "Київ" };
}
@{
    string[] countries = new string[] { "Великобританія", "Франція", "Україна" };
}
@using BookStore.Helpers
<h3>Міста</h3>
@Html.CreateList(cities)
<br />
<h3>Країни</h3>
<!--або можна викликати так -->
@ListHelper.CreateList(Html, countries)
```

Додавання атрибутів

Стандартні реалізації хелперів дозволяють додати до створюваних елементів атрибути, наприклад, атрибут class, id та інші. У кастомні хелпери ми також можемо додавати атрибути. Наприклад, змінимо хелпер CreateList наступним чином:

```
using System.Collections.Generic;
using System.Linq;
using System.Reflection;
using System.Web.Mvc;

public static class ListHelper
{
    public static MvcHtmlString CreateList(this HtmlHelper html, string[] items, object obj)
    {
        var type = obj.GetType();
        var props = type.GetProperties();

        Dictionary<string, string> dic= props.ToDictionary(x => x.Name, x => x.GetValue(obj,
null).ToString());

        TagBuilder ul = new TagBuilder("ul");
        foreach (string item in items)
        {
            TagBuilder li = new TagBuilder("li");
            li.SetInnerText(item);
            ul.InnerHtml += li.ToString();
        }
        foreach (var attr in dic)
```

```

    {
        ul.MergeAttribute(attr.Key.ToString(), attr.Value.ToString());
    }
    return new MvcHtmlString(ul.ToString());
}
}

```

Тепер в якості другого параметра передається об'єкт `obj`, властивості якого повинні представляти собою передавані в хелпер атрибути. За допомогою механізму рефлексії цей об'єкт конвертується в словник `Dictionary`.

І в цьому випадку в коді представлення ми можемо передати в хелпер значення для атрибутів:

```
@Html.CreateList(new string[] { "Лондон", "Париж", "Берлін" }, new { @class = "btn", id = "citiesList" })
```

3.4.12 Робота з формами

Хоча ми можемо самі написати будь-який потрібний хелпер, але фреймворк MVC вже має великий набір вбудованих html-хелперів, які дозволяють генерувати ту чи іншу розмітку, головним чином, для роботи з формами. Тому в більшості випадків не доведеться створювати свої хелпери, і можна буде скористатися вбудованими.

Хелпер `Html.BeginForm`

Для створення форм ми цілком можемо використовувати стандартні елементи html, наприклад:

```

<form method="post" action="/Home/Buy">
  <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>

```

Це звичайна html-форма, яка після натискання на кнопку відправляє всі введені дані запитом POST на адресу `/Home/Buy`. Вбудований хелпер `BeginForm/EndForm` дозволяє створити ту ж саму форму:

```

@using(Html.BeginForm("Buy", "Home", FormMethod.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>
}

```

Метод BeginForm приймає в якості параметрів ім'я методу дії і ім'я контролера, а також тип запиту. Даний хелпер створює як відкриваючий тег <form>, так і закриваючий тег </form>. Тому при рендерінгу представлення у вихідний потік у нас вийде той же самий html-код, що і з застосуванням тега form. Тому обидва способи ідентичні.

Якщо у нас в контролері визначені дві версії одного методу - для методів POST і GET, наприклад:

```

[HttpGet]
public ActionResult Buy()
{
    return View();
}

[HttpPost]
public string Buy(Purchase purchase)
{
    .....
    return "Дякуємо за покупку";
}

```

Тобто фактично виклик сторінки з формою і відправка форми здійснюється одним і тією же дією Buy. У цьому випадку можна не вказувати в хелпері Html.BeginForm параметри:

```

@using(Html.BeginForm())
{
    .....
}

```

Введення інформації

У попередньому прикладі разом з хелперів Html.BeginForm використовувалися стандартні елементи html. Однак набір html-хелперів містить також хелпери для введення інформації користувачем. У MVC визначений широкий набір хелперів введення практично для кожного html-елемента. Що вибрати - хелпер або стандартний елементи введення html, вже вирішує сам розробник.

Незалежно від типу всі базові html-хелпери використовують як мінімум два параметри: перший параметр застосовується для установки значень для атрибутів id і name, а другий параметр - для установки значення атрибуту value.

Html.TextBox

Хелпер `Html.TextBox` генерує тег `input` зі значенням атрибута `type` рівним `text`. Хелпер `TextBox` використовують для отримання введення користувачем інформації. Так, перепишемо попередню форму із заміною полів введення на хелпер `Html.TextBox`:

```
@using(Html.BeginForm("Buy", "Home", FormMethod.Post))
{
    <input type="hidden" value="@ViewBag.BookId" name="BookId" />
    <p>Введіть своє ім'я: </p>
    @Html.TextBox("Person", "Введіть ім'я")
    <p>Введіть адресу :</p>
    @Html.TextBox("Address", " Введіть адресу")
    <p><input type="submit" value="Відправити" /></p>
}
```

Html.TextArea

Хелпер `TextArea` використовується для створення елемента `<textarea>`, який представляє багаторядкове текстове поле. Результатом виразу

```
@Html.TextArea("text", "hello <br/> world")
```

буде наступна html-розмітка:

```
<textarea cols="20" id="text" name="text" rows="2">hello <br/> world
</textarea>
```

Звернемо увагу, що хелпер декодує розміщене в нього значення, у тому числі і `html`-теги, (всі хелпери декодують значення моделей і значення атрибутів). Інші версії хелпера `TextArea` дозволяють вказати число рядків і стовпців, що визначають розмір текстового поля.

```
@Html.TextArea("text", " hello <br /> world ", 5, 50, null)
```

Цей хелпер згенерує наступну розмітку:

```
<textarea cols="50" id="text" name="text" rows="5"> hello <br /> world
</textarea>
```

Html.Hidden

У прикладі з формою ми використовували приховане поле `input type = "hidden"`, замість якого могли б використовувати хелпер `Html.Hidden`. Так, наступний виклик хелпера:

```
@Html.Hidden("BookId", "2")
```

згенерує розмітку:

```
<input id="BookId" name="BookId" type="hidden" value="2" />
```

Html.Password

Html.Password створює поле для введення пароля. Він схожий на хелпер TextBox, але замість введених символів відображає маску пароля. Наступний код

```
@Html.Password("UserPassword", "val")
```

генерує розмітку:

```
<input id="UserPassword" name="UserPassword" type="password" value="val" />
```

Html.RadioButton

Для створення перемикачів застосовується хелпер Html.RadioButton. Він генерує елемент input зі значенням type = "radio". Для створення групи перемикачів, треба присвоїти всім їм одне і те ж ім'я (властивість name):

```
@Html.RadioButton("color", "red")  
<span>червоний</span> <br />  
@Html.RadioButton("color", "blue")  
<span>синій</span> <br />  
@Html.RadioButton("color", "green", true)  
<span>зелений</span>
```

Такий код створить наступну розмітку:

```
<input id="color" name="color" type="radio" value="red" />  
<span> червоний </span> <br />  
<input id="color" name="color" type="radio" value="blue" />  
<span> синій </span> <br />  
<input checked="checked" id="color" name="color" type="radio" value="green" />  
<span> зелений </span>
```

Html.CheckBox

Html.CheckBox може застосовуватися для створення відразу двох елементів. Візьмемо, приміром, наступний код:

```
@Html.CheckBox("Enable", false)
```

Це вираз буде генерувати наступний HTML:

```
<input id="Enable" name="Enable" type="checkbox" value="true" />  
<input name="Enable" type="hidden" value="false" />
```

Тобто окрім поля флага, ще й генерується приховане поле. Браузер посилає значення флага тільки тоді, коли флаг обраний або відзначений. А приховане поле гарантує, що для елемента Enable буде встановлено значення навіть, якщо користувач не зазначив флаг.

Html.Label

Хелпер `Html.Label` створює елемент `<label/>`, а параметр, що передається в хелпер визначає значення атрибута `for` і одночасно текст на елементі. Перевантажена версія хелпера дозволяє визначити значення атрибута `for` і текст на мітці незалежно один від одного. Наприклад, оголошення хелпера `Html.Label("Name")` створює наступну розмітку:

```
<label for="Name">Name</label>
```

Елемент `label` представляє просту мітку, призначену для прикріплення інформації до елементів введення, наприклад, до текстових полів. Атрибут `for` елемента `label` повинен містити ID асоційованого елемента вводу. Якщо користувач натискає на мітку, то браузер автоматично передає фокус пов'язаному з цією міткою елементу введення.

Html.DropDownList

Хелпер `Html.DropDownList` створює випадаючий список, тобто елемент `<select />`. Для генерації такого списку потрібна колекція об'єктів `SelectListItem`, які представляють елементи списку. Об'єкт `SelectListItem` має властивості `Text` (відображуваний текст), `Value` (саме значення, яке може не збігатися з текстом) і `Selected`. Можна створити колекцію об'єктів `SelectListItem` або використовувати хелпер `SelectList`. Цей хелпер переглядає об'єкти `IEnumerable` і перетворює їх в послідовність об'єктів `SelectListItem`. Так, код `@Html.DropDownList("countires", new SelectList(new string[] { "Ukraine", "USA", "Canada", "France" }), "Countries")` генерує наступну розмітку:

```
<select id="countires" name="countires"><option value="">Countries</option>
<option>Ukraine</option>
<option>USA</option>
<option>Canada</option>
<option>France</option>
</select>
```

Тепер більш складний приклад. Виведемо в список колекцію елементів `Book`. У контролері передамо цей список через `ViewBag`:

```
BookContext db = new BookContext();
public ActionResult Index()
{
    SelectList books = new SelectList(db.Books, "Author", "Name");
    ViewBag.Books = books;
}
```

```

return View();
}

```

Тут ми створюємо об'єкт SelectList, передаючи в його конструктор набір значень для списку (db.Books), назва властивості моделі Book, яке буде використовуватися в якості значення (Author), і назва властивості моделі Book, яке буде використовуватися для відображення у списку. В даному випадку необов'язково встановлювати дві різні властивості, можна було встановити одне і для значення і відображення.

Тоді в представленні ми можемо використовувати SelectList:

```
@Html.DropDownList("Author", ViewBag.Books as SelectList)
```

І при рендерінгу представлення всі елементи SelectList додадуться до списку, що випадає.

Html.ListBox

Хелпер Html.ListBox, також як і DropDownList, створює елемент <select>, але при цьому робить можливим множинне виділення елементів (тобто для атрибута multiple встановлюється значення multiple). Для створення списку, що підтримує множинне виділення, замість SelectList можна використовувати клас MultiSelectList:

```
@Html.ListBox("countires", new MultiSelectList(new string[] { "Україна", "США", "Китай", "Індія" })))
```

Цей код згенерує наступну розмітку

```

<select Length="9" id="countries" multiple="multiple" name="countires">
<option> Україна </option>
<option>США</option>
<option>Китай</option>
<option> Індія </option>
</select>

```

З передачею одиночних значень на сервер все зрозуміло, але як передати множинні значення? Припустимо, у нас є наступна форма:

```

@using (Html.BeginForm())
{
    @Html.ListBox("countries",
        new MultiSelectList(new string[] { "Україна", "США", "Китай", "Індія" })))
    <p><input type="submit" value="Відправити" /></p>
}

```

Тоді метод контролера міг би отримувати ці значення таким чином:

```
[HttpPost]
```



```

public string Index(string[] countries)
{
    string result = "";
    foreach (string c in countries)
    {
        result += c;
        result += ";";
    }
    return "Вы выбрали: " + result;
}

```

Форма з декількома кнопками

Як правило, на формі є тільки одна кнопка для відправки. Однак у певних ситуаціях може виникнути потреба, використовувати більше однієї кнопки. Наприклад, є поле для введення значення, а дві кнопки вказують, треба це значення видалити або, навпаки, додати:

```

@using (Html.BeginForm("MyAction", "Home", FormMethod.Post))
{
    <input type="text" name="product" /><br/>
    <button name="action" value="add">додати</button>
    <button name="action" value="delete">видалити</button>
}

```

Найпростіше рішення полягає в тому, що для кожної кнопки встановлюється однакове значення атрибута name, але різний атрибут value. А метод, що приймає форму, може виглядати наступним чином:

```

[HttpPost]
public ActionResult MyAction(string product, string action)
{
    if(action=="add")
    {

    }
    else if(action=="delete")
    {

    }
    // ***** код методу
}

```

І за допомогою умовної конструкції залежно від значення параметра action, який зберігає значення атрибута value після натискання кнопки, виробляються певні дії.

3.4.13 Строго типізовані хелпери

Крім базових хелперів в ASP.NET MVC є їх двійники - строго типізовані хелпери. Цей вид хелперів приймає як параметр лямбда-вираз, в якому

вказується те властивість моделі, до якого повинен бути прив'язаний даний хелпер. Важливо враховувати, що строго типізовані хелпери можуть використовуватися тільки в строго типізованих представленнях, а тип моделі, яка передається в хелпер, повинен бути тим же самим, що зазначений для усього представлення за допомогою директиви @model.

```
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; }
}
```

І для її використання застосовувалася наступна форма:

```
<form method="post" action="">
    <input type="hidden" value="@ViewBag.BookId" name="BookId" />
    <table>
        <tr><td><p>Введіть своє ім'я </p></td>
            <td><input type="text" name="Person" /> </td></tr>
        <tr><td><p>Введіть адресу :</p></td><td>
            <input type="text" name="Address" /> </td></tr>
        <tr><td><input type="submit" value="Відправити" /> </td>
            <td></td></tr>
    </table>
</form>
```

Перепишемо цей приклад з використанням хелперів:

```
@model BookStore.Models.Purchase
@{
    Layout = "~/Views/Shared/_Layout.cshtml";
}

<div>
    <h3>Форма оформления покупки</h3>
    @using(Html.BeginForm("Buy", "Home", FormMethod.Post))
    {
        @Html.HiddenFor(m=>m.BookId)
        @Html.LabelFor(m => m.Person, "Введіть своє ім'я")
        <br />
        @Html.TextBoxFor(m=>m.Person)
        <br /><br />
        @Html.LabelFor(m => m.Address, "Введіть адресу :")
        <br />
        @Html.TextBoxFor(m=>m.Address)
        <p><input type="submit" value="Відправити " /></p>
    }
}
```

</div>

Строго типізований хелпер схожий на звичайний, тільки в кінці додається суфікс **For**: **LabelFor**. Так як строго типізовані хелпери можуть використовуватися тільки в строго типізованих представленнях, то на початку представленнях вказуємо модель, яка буде використовуватися: `@model BookStore.Models.Purchase`. Тобто, у виклику `@Html.TextBoxFor (m => m.Person)` параметр `m` являє змінну моделі `Purchase`. А лямбда-вираз `m => m.Person` вказує, що даний хелпер буде генерувати текстове поле для властивості `Person`. Таким чином, хелпер `@Html.TextBoxFor (m => m.Person)` згенерує текстове поле `<input id = "Person" name = "Person" type = "text" value = "" />`.

Для кожного базового вбудованого хелпера є свій строго типізований хелпер:

- **Html.CheckBoxFor**

Вираз `@Html.CheckBoxFor(m=>m.Enable, false)` створює розмітку:

```
<input id="Enable" name="Enable" type="checkbox" value="true" />
```

```
<input name="Enable" type="hidden" value="false" />
```

- **Html.HiddenFor**

Вираз `@Html.HiddenFor(m=>m.Name)` створює розмітку:

```
<input id = "Name" name = "Name" type = "hidden" value = "[значення_m.Name]" />
```

- **Html.LabelFor**

Хелпер `@Html.LabelFor(m=>m.Name, "Ім'я")` генерує розмітку:

```
<label for = "Name"> Ім'я </ label>
```

- **Html.PasswordFor**

Хелпер `@Html.PasswordFor(m=>m.Password)` створює розмітку:

```
<input id = "Password" name = "Password" type = "password" />
```

- **Html.RadioButtonFor**

`@Html.RadioButtonFor(m=>m.Option, "val")` генерує розмітку:

```
<input id = "Option" name = "Option" type = "radio" value = "val" />
```

- **Html.TextBoxFor**

вираз `@Html.TextBoxFor(m=>m.Name)` створює розмітку:

```
<input id = "Name" name = "Name" type = "text" />
```

- **Html.TextAreaFor**

Хелпер `@Html.TextAreaFor(m=>m.Name, 10, 9, null)` генерує код:

```
<textarea cols = "9" id = "Name" name = "Name" rows = "10"> </ textarea>
```

Контрольні питання.

1. Основні способи застосування представлень (View)
2. Застосування майстер-сторінки. Переваги і недоліки використання.
3. HTML-хелпери. Особливості використання.

3.5 Моделі

Всі сутності в додатку прийнято виділяти в окремі моделі. Залежно від поставленого завдання і складності програми можна виділити різну кількість моделей. Так, в текстах попередніх прикладів використовувалися дві моделі - клас для книги і клас для покупки книги. Моделі представляють собою прості класи і розташовуються в проекті в каталозі 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; }
    // ціна
    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; }
}
```

Модель не обов'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. Хоча тут теж є неоднозначності. Іноді бізнес-логіку опрацьовує модель а контролер зосереджується на обробці даних від вводу користувача.

Дані моделей, як правило, зберігаються в базі даних. Для роботи з базою даних дуже зручно користуватися фреймворком Entity Framework, який дозволяє абстрагуватися від написання sql-запитів, від будови бази даних і повністю зосередитися на логіці програми.

Якщо при створенні проекту MVC 5 ви виберете як типу аутентифікації "No Authentication", то після створення проекту в нього треба буде підключити EntityFramework через пакетний менеджер NuGet.

В якості альтернативи NuGet можна використовувати консоль пакетного менеджера. Для цього в меню Visual Studio виберемо View -> Other Windows -> Package Manager Console. Після цього внизу студії відкриється консоль пакетного менеджера. У ній введемо таку команду:

PM> Install-Package EntityFramework -Version 6.0.2

Після введення команди буде завантажений і встановлено пакет Entity Framework. Іноді цієї консоллю краще користуватися при установці пакетів, ніж менеджером NuGet, так як менеджер NuGet може трохи спізнюватися за випуском останніх версій пакетів. Або навпаки, нам треба встановити пакети більш ранньої версії, а NuGet може запропонувати тільки поточну версію.

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 Studio. Навпаки, ми створюємо звичайні класи, а Entity Framework сам визначає, як і де зберігати об'єкти цих класів.

Для підключення до бази даних через Entity Framework, нам потрібен посередник - контекст даних. Контекст даних являє собою клас, похідний від класу DbContext. Контекст даних містить одне або кілька властивостей типу DbSet <T>, де T представляє тип об'єкта, що зберігається в базі даних. Наприклад, створимо контекст даних для роботи з вищенаведеними моделями:

```
using System;
using System.Collections.Generic;
using System.Web;
using System.Data.Entity;

namespace BookStore.Models
{
    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.

3.5.1 Підключення до бази даних

Для зберігання даних додатку потрібна база даних. Ми можемо використовувати різні СУБД, але, як правило, в якості бази даних у зв'язці з ASP.NET MVC використовується база даних MS SQL Server, на прикладі якого ми й 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. Для зберігання баз даних проекті призначена папка App_Data. Отже, натиснемо на папку App_Data правою кнопкою миші і в контекстному меню виберемо Add-> New Item У вікні додавання нового елементу виберемо SQL Server Database і назовемо нову базу даних Bookstore.mdf:

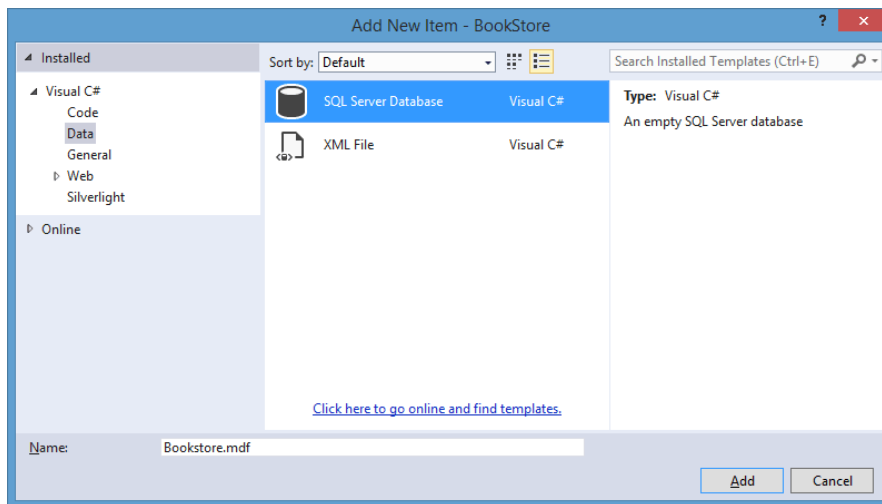


Рисунок 3.16 - Вікно створення БД

Після цього в папку папку AppData буде додана база даних, і ми можемо починати з нею працювати - додавати таблиці і дані. Але перед цим розглянемо, що у нас буде зберігатися в БД.

```
public class Book
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Author { get; set; }
    public int Price { get; set; }
}
```

І модель Purchase:

```
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; }
}
```

Нехай для роботи з ними визначений наступний контекст даних:

```
public class BookContext : DbContext
{
    public DbSet<Book> Books { get; set; }
    public DbSet<Purchase> Purchases { get; set; }
}
```

Щоб зв'язати додаток, контекст даних і БД, додамо в файл web.config рядок підключення до цієї бази даних. Для цього після секції configSections вставимо наступну секцію:

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

Зверніть увагу, що значення атрибута name (name = "Book Context") в якості значення має назву контексту даних.

Є також інший спосіб. Наприклад, якщо у нас рядок підключення має іншу назву, наприклад <add name = "DefaultConnection", і ми не хочемо його міняти, то в цьому випадку для зв'язку з контекстом даних ми можемо передати назву в конструктор базового класу:

```
public class BookContext : DbContext
{
    public BookContext(): base("DefaultConnection")
    {}
    public DbSet<Book> Books { get; set; }
    public DbSet<Purchase> Purchases { get; set; }
}
```

У Visual Studio 2013 ми можемо використовувати LocalDB.LocalDB представляє полегшену версію сервера баз даних SQL Sever Express, яка спеціально націлена на розробників. Тому в даному випадку в якості джерела даних вказуємо (LocalDB)\v11.0. Використання підстановки | DataDirectory | дозволяє опустити повний фізичний шлях до бази даних, яка зберігається в папці App_Data.

Визначення рядка підключення залежить від використовуваної СУБД і її версії. Наприклад, v11.0 припускає, що використовується MS SQL Server 2012. Для MS SQL Server 2014 при використанні LocalDB рядок підключення буде відрізнятися - він не містить визначення версії:

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

Якщо ми використовуємо не сервер localdb, а повноцінний MS SQL Server, то тоді в рядок підключення включається назва сервера:

```
<connectionStrings>
  <add name="BookContext" connectionString="Data Source=.\\SQLEXPRESS;Initial
Catalog=Bookstore;Integrated Security=True"
providerName="System.Data.SqlClient"/>
</connectionStrings>
```

Крім того, тут також замість AttachDbFilename використовується назва каталогу БД "Initial Catalog".

Для інших СУБД - MySQL, MongoDB, Postgre і т.д. рядок підключення також може відрізнятися. Але в даному випадку ми будемо використовувати LocalDB.

Тепер створимо самі таблиці. Для цього перейдемо у вікно Server Explorer і розкриємо в ньому вузол Data Connections. Далі ми побачимо вузол BookContext, який представляє нашу БД, і також розкриємо його:

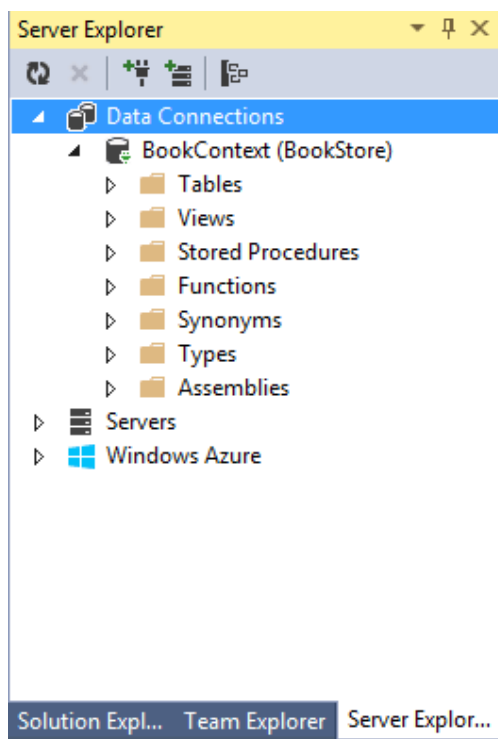


Рисунок 3.17 - Панель управління БД

Додамо першу таблицю. Для цього натискаємо правою кнопкою миші на вузол Tables і в меню виберемо пункт Add New Table. Після цього відобразиться вікно дизайнера таблиці, в якому треба визначити назви і типи стовпців нової таблиці. Щодо угод про найменуваннях таблиці при роботі з Entity Framework назви таблиць повинні відповідати імені моделі. Наприклад, якщо модель називається Book, то таблиця буде називатися Books. А Entity Framework автоматично розпізнає, що таблиця Books відповідає класу Book. Отже, створимо структуру таблиці:

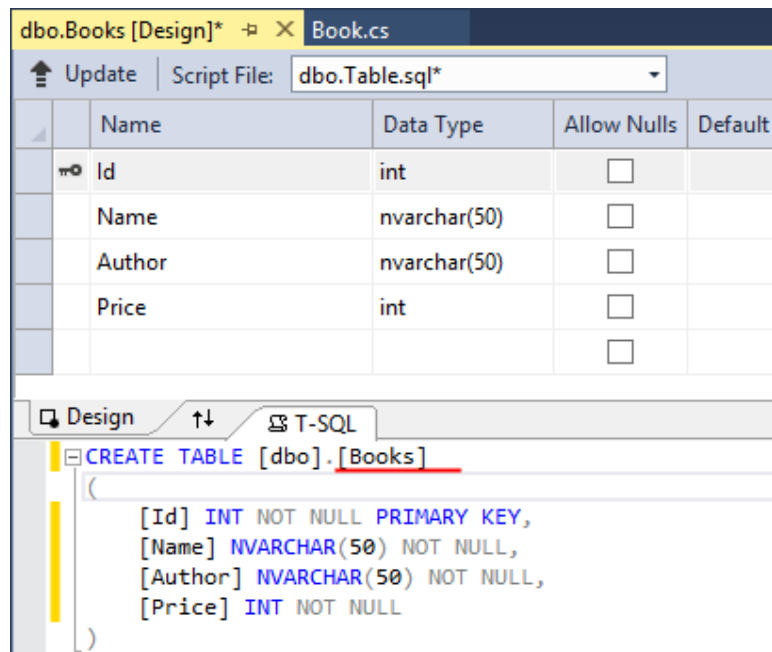


Рисунок 3.18 - Структура бази даних

Також оскільки передбачається, що стовпець Id буде інкрементувати з додаванням кожного нового об'єкта, то встановимо для нього автоінкремент у вікні Properties:

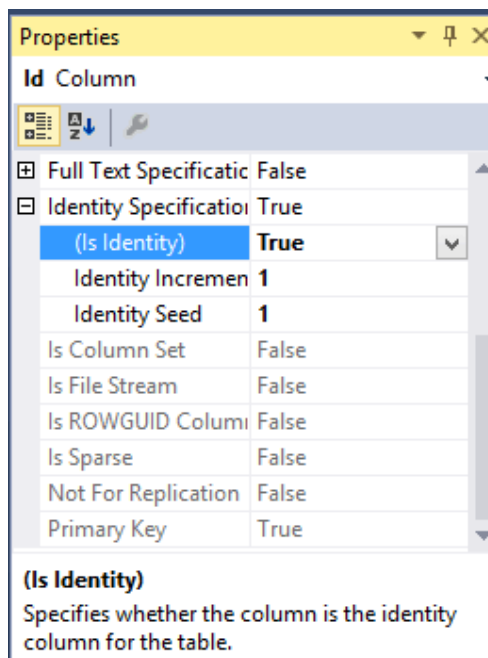


Рисунок 3.19 - Вікно властивостей

І останній крок - генерація бази даних. Для цього натиснемо на кнопку Update:

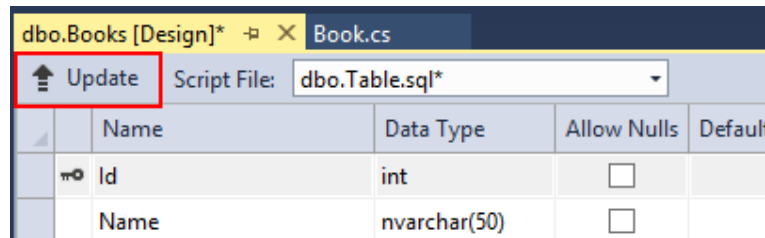


Рисунок 3.20 - Структура бази даних

У діалоговому вікні натискаємо на кнопку Update Database. Після цього в нашу базу даних додається нова таблиця. Подібним чином визначимо таблицю Purchases для моделі Purchase:

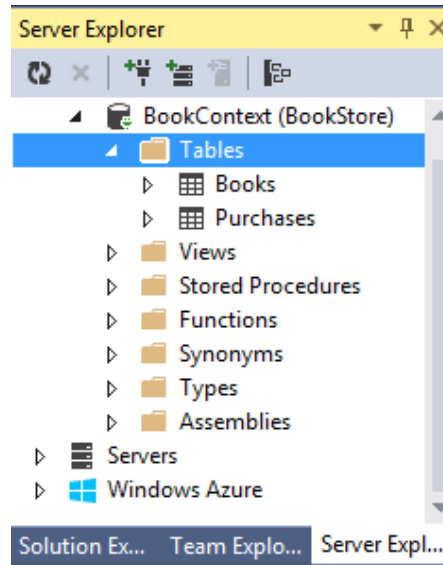


Рисунок 3.21 - Структура бази даних

Додамо в дизайнері баз даних в таблицю Books кілька записів. Для цього натиснемо у вікні Server Explorer на вузол Books і виберемо в списку пункт Show Table Data. Додамо, наприклад, наступний набір записів:

dbo.Books [Data]		Book.cs		
		Max Rows: 1000		
	Id	Name	Author	Price
	1	Война и мир	Л. Толстой	220
	2	Чайка	А. Чехов	120
	3	Отцы и дети	И. Тургенев	160
▶*	NULL	NULL	NULL	NULL

Рисунок 3.22 - Структура бази даних

Тепер ми можемо отримати ці дані в контролері Home і передати їх в представлення:

```

public class HomeController : Controller
{
    BookContext db = new BookContext();
    public ActionResult Index()
    {
        return View(db.Books);
    }
}

```

Виведення даних у поданні Index.cshtml:

```

@model IEnumerable<BookStore.Models.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>Цена</p></td><td></td>
        </tr>
        @foreach (BookStore.Models.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>

```

Закриття з'єднання

Щоб напевно бути впевненим, що підключення до бази даних закрито, слід викликати метод `Dispose` у контексті даних:

```

protected override void Dispose(bool disposing)
{
    db.Dispose();
    base.Dispose(disposing);
}

```

Це перевизначення версія методу `Dispose` контролера, яка викликається при знищенні об'єкта контролера. У неї поміщається виклик `db.Dispose()`, який знищує всі пов'язані з контекстом даних ресурси та підключення.

3.5.2 Шаблонні хелпери

Крім базових html-хелперів, розглянутих у минулому розділі і генеруючих певні елементи розмітки html, фреймворк ASP.NET MVC також має шаблонні хелпери. На відміну від розглянутих в минулому розділі html-хелперів вони не генерують певний елемент html. Шаблонні хелпери аналізують властивість моделі і генерують той елемент html, який найбільш підходить даній властивості, виходячи з його типу і метаданих.

У ASP.NET MVC є наступні шаблонні хелпери:

- **Display** Створює елемент розмітки для відображення значення вказаної властивості моделі: `Html.Display("Name")`
- **DisplayFor** Строго типізований аналог хелпера `Display`:
`Html.DisplayFor(m=>m.Name)`
- **Editor** Створює елемент розмітки для редагування вказаної властивості моделі: `Html.Editor("Name")`
- **EditorFor** Строго типізований аналог хелпера `Editor`:
`Html.EditorFor(m=>m.Name)`
- **DisplayText** Створює вираз для вказаної властивості моделі у вигляді простого рядка: `Html.DisplayText("Name")`
- **DisplayTextFor** Строго типізований аналог хелпера `DisplayText`:
`Html.DisplayTextFor (m=>m.Name)`

Це були поодинокі хелпери, які генерують розмітку тільки для однієї властивості моделі. Але крім них у фреймворку також є ще кілька шаблонів, які дозволяють створити разом всі поля для всіх властивостей моделі:

- **DisplayForModel** Створює поля для читання для всіх властивостей моделі:
`Html.DisplayForModel ()`
- **EditorForModel** Створює поля для редагування для всіх властивостей моделі: `Html.EditorForModel ()`

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

```
@{  
    Layout = "~/Views/Shared/_Layout.cshtml";  
}
```

```
@model BookStore.Models.Book
```

```
<h2>Книга № @Model.Id</h2>  
@Html.DisplayForModel()
```

Звернемося до цього ресурсу, набравши в адресному рядку браузера Home/BookView/1 і одержимо відображення інформації по книзі №1.

3.5.3 Редагування моделі

Додамо в контролер дію, яка отримуватиме по Id модель і виводитиме в представлення її властивості для редагування:

```
[HttpGet]
public ActionResult EditBook(int? id)
{
    if (id == null)
    {
        return HttpNotFound();
    }
    Book book = db.Books.Find(id);
    if (book != null)
    {
        return View(book);
    }
    return HttpNotFound();
}
```

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:

```
@{
    ViewBag.Title = "Редагувати книгу";
    Layout = "~/Views/Shared/_Layout.cshtml";
}
@model BookStore.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>
    </fieldset>
}
```

```

<p>
    @Html.LabelFor(m => m.Price, "Ціна")
    <br />
    @Html.EditorFor(m => m.Price)
</p>
<p><input type="submit" value="Відправити" /></p>
</fieldset>
}

```

Оскільки унікальний ідентифікатор `id` книги нам не треба 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:

```

[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 існує `book` вже в базі даних, і для нього треба внести в базу змінене значення, а не створювати новий запис. Після чого перенаправляються на головну сторінку.

Варто зазначити, 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 ;` 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` згенерує поля для редагування. Ми можемо змінити дані моделі, і відправити їх на сервер, де здійсниться їх збереження.

3.5.4 Додавання та видалення моделі

Додавання моделі

Продовжимо роботу з моделлю `Book` і подивимося, як можемо її додавати і видаляти з БД. Для додавання моделі спочатку визначимо деякі дії:

```

[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, яке буде виглядати наступним чином:

```
@model BookStore.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");
}
```

Видалення моделі

Тепер найважливіша частина - видалення моделі. Це важливо навіть не в плані реалізації, а в плані безпеки. Додамо проста дія, 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, а чи є такий об'єкт у 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` він згенерує `sql`-вираз `DELETE`. Але ми можемо самотійно 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 видалення має один мінус в плані безпеки. Припустимо, нам прийшов електронний лист, в який була впроваджена картинка за допомогою тега:

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

У підсумку при відкритті листа 1-й запис у таблиці може бути вилучений. Уразливість стосується не тільки листів, але може проявлятися і в інших місцях, але сенс один - GET-запит до методу `Delete` несе потенційну вразливість. Тому переробимо метод таким чином:

```
[HttpGet]
public ActionResult Delete(int id)
{
    Book b = db.Books.Find(id);
    if (b == null)
    {
        return HttpNotFound();
    }
    return View(b);
}
```



```

}
[HttpPost, ActionName("Delete")]
public ActionResult DeleteConfirmed(int id)
{
    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 маємо два. Атрибут ActionName("Delete") вказує, що метод DeleteConfirmed сприйматиметься як дію Delete. Перший метод передає модель яка має бути видалена в представлення. На представленні за допомогою натискання кнопки ми зможемо підтвердити видалення. І id який видаляється, піде другому методу за запитом POST. Таким чином, ми уникнемо вразливості GET-запиту. Ну і саме представлення:

```

@{
    Layout = "~/Views/Shared/_Layout.cshtml";
}
@model BookStore.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" />
}

```

3.5.5 Шаблони формування

Так як більшість додатків так чи інакше ґрунтуються на стандартних CRUD-операціях (Create-Read-Update-Delete), то найчастіше розробники змушені багаторазово створювати контролери та представлення для одних і тих же дій: додавання, зміни, видалення і перегляду записів з БД. І щоб полегшити розробникам життя, команда MVC додала таку корисну функціональність, як шаблони формування (scaffolding templates). Ці шаблони дозволяють за заданими моделлю і контекстом даних сформувати весь необхідний базовий код контролерів, а також всю розмітку для представлень, за допомогою яких можна управляти записами в БД.

Щоб скористатися даною функціональністю, додамо новий контролер. Натиснемо правою кнопкою миші на папку Controllers і виберемо Add->Controller.... Далі у вікні додавання нового контролера нам буде запропоновано вибрати шаблон контролера:

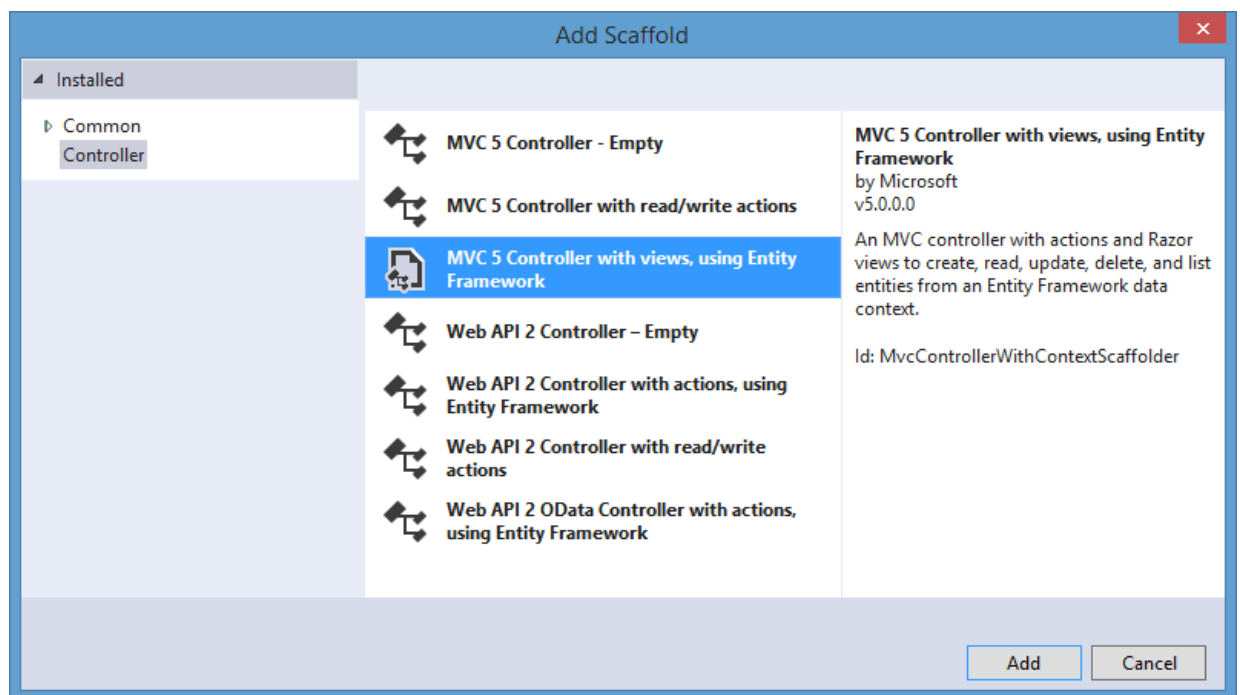


Рисунок 3.23 - Вибір контролера з шаблонами формування

Власне до MVC відносяться тільки перші три шаблони:

- **MVC5 Controller-Empty.** Цей шаблон додає в папку Controllers порожній контролер, який має один єдиний метод Index. Даний шаблон не створює представлень.
- **MVC5 Controller with read/write actions.** Даний шаблон додає в проект контролер, який містить методи Index, Details, Create, Edit і Delete. Однак ці методи не містять ніякої логіки роботи з базою даних. І нам пропонується самим створити для них код і представлення.

- **MVC5 Controller with views, using Entity Framework.** Це найбільш цікавий шаблон, який 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.

Виберемо останній пункт, тобто **MVC5 Controller with views, using Entity Framework**. Після цього відкриється вікно додавання нового контролера, в якому нам буде запропоновано встановити деякі налаштування:

- **Controller name:** ім'я контролера
- **Use async controller actions:** чи будуть автоматичні згенеровані методи контролера асинхронними. Встановимо дану опцію.
- **Model class:** клас моделі. Виберемо створену раніше модель Book (або якусь іншу наявну модель)
- **Data context class:** клас контексту даних. Виберемо контекст даних для обраної моделі.
- **Generate views:** чи треба генерувати представлення до створюваних дій контролера. При установці цієї опції стають доступними дві наступні опції. Встановимо всі ці опції.
- **Reference script libraries:** чи будуть підключати представлення бібліотеки jquery та інші необхідні файли javascript
- **Use a layout page:** чи будуть генеруватися представлення використовувати майстер-сторінку

Рисунок 3.24 - Форма додавання контролера

Встановивши всі опції, натиснемо кнопку Add, і в проект буде додано новий контролер. Він буде виглядати приблизно наступним чином:

```
using System;
using System.Collections.Generic;
```

```

using System.Data;
using System.Data.Entity;
using System.Linq;
using System.Threading.Tasks;
using System.Net;
using System.Web;
using System.Web.Mvc;
using BookStore.Models;

namespace BookStore.Controllers
{
    public class BookController : Controller
    {
        private BookContext db = new BookContext();

        public async Task<ActionResult> Index()
        {
            return View(await db.Books.ToListAsync());
        }

        public async Task<ActionResult> Details(int? id)
        {
            if (id == null)
            {
                return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
            }
            Book book = await db.Books.FindAsync(id);
            if (book == null)
            {
                return HttpNotFound();
            }
            return View(book);
        }

        public ActionResult Create()
        {
            return View();
        }

        [HttpPost]
        [ValidateAntiForgeryToken]
        public async Task<ActionResult> Create([Bind(Include="Id,Name,Author,Price")] Book book)
        {
            if (ModelState.IsValid)
            {
                db.Books.Add(book);
                await db.SaveChangesAsync();
                return RedirectToAction("Index");
            }

            return View(book);
        }
    }
}

```

```

}

public async Task<ActionResult> Edit(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    Book book = await db.Books.FindAsync(id);
    if (book == null)
    {
        return HttpNotFound();
    }
    return View(book);
}

[HttpPost]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Edit([Bind(Include="Id,Name,Author,Price")] Book book)
{
    if (ModelState.IsValid)
    {
        db.Entry(book).State = EntityState.Modified;
        await db.SaveChangesAsync();
        return RedirectToAction("Index");
    }
    return View(book);
}

public async Task<ActionResult> Delete(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    Book book = await db.Books.FindAsync(id);
    if (book == null)
    {
        return HttpNotFound();
    }
    return View(book);
}

[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<ActionResult> DeleteConfirmed(int id)
{
    Book book = await db.Books.FindAsync(id);
    db.Books.Remove(book);
    await db.SaveChangesAsync();
    return RedirectToAction("Index");
}

```

```

    }

    protected override void Dispose(bool disposing)
    {
        if (disposing)
        {
            db.Dispose();
        }
        base.Dispose(disposing);
    }
}

```

А в папці Views/Book ми знайдемо всі необхідні представлення з усім необхідним кодом, який тепер нам не треба набирати вручну. І тепер ми можемо запустити проект і перейти в адресному рядку браузера до нашого контролера.

Завдяки шаблонам формування ми можемо не думати про створення коду для стандартних операцій, що дозволяє заощадити багато часу. Правда, після генерації коду все одно доведеться вносити правки, змінювати автоматично згенерували назви на свої (наприклад, назва сторінки, автоматично генеровані посилання, тощо.), Проте від основної роботи ми вже будемо позбавлені.

3.5.6 Моделі зі складною структурою

Раніше ми використовували відносно прості моделі Book і Purchase. Але в реальних додатках більшість моделей, як правило, виявляються набагато складнішими за структурою. Наприклад, створимо дві наступні моделі, що представляють футболіста і футбольну команду:

```

public class Player
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }
    public string Position { get; set; }
    public int? TeamId { get; set; }
    public Team Team { get; set; }
}

public class Team
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Coach { get; set; }

    public ICollection<Player> Players { get; set; }
    public Team()
    {
        Players = new List<Player>();
    }
}

```

}

Крім стандартних властивостей типу string клас **Prayer** має властивість **Team**, яка визначає приналежність футболіста до певної команди. Властивість Team в даному випадку є навігаційною властивістю. Завдяки навігаційній властивості ми можемо витягувати пов'язані з об'єктом дані з БД. Але для цього треба також встановити зовнішній ключ. Зовнішній ключ складається з двох властивостей: навігаційного і звичайного. Навігаційний ми розглянули вище. А звичайний повинний приймати одне з наступних варіантів імені:

- Ім'я_навігаційної_властивості + Ім'я ключа із зв'язаної таблиці - в нашому випадку ім'я навігаційної властивості Team, а ключа з моделі Team - Id, тому в нашому випадку властивість називається TeamId.
- Ім'я_класа_зв'язаної_таблиці + Ім'я ключа із зв'язаної таблиці - в нашому випадку клас Team, а ключа з моделі Team - Id. І тут знову ж виходить TeamId.

Тепер створимо контекст даних, що використовує моделі:

```
public class SoccerContext : DbContext
{
    public DbSet<Player> Players { get; set; }
    public DbSet<Team> Teams { get; set; }
}
```

Тепер подивимося, як це розташовується в БД. Наприклад, створимо базу даних SoccerInfo.mdf. Нехай для зберігання моделей Player і Team визначені відповідно у таблицях Players і Teams.

Визначення таблиці Teams, яка буде зберігати об'єкти моделі Team, виглядає наступним чином:

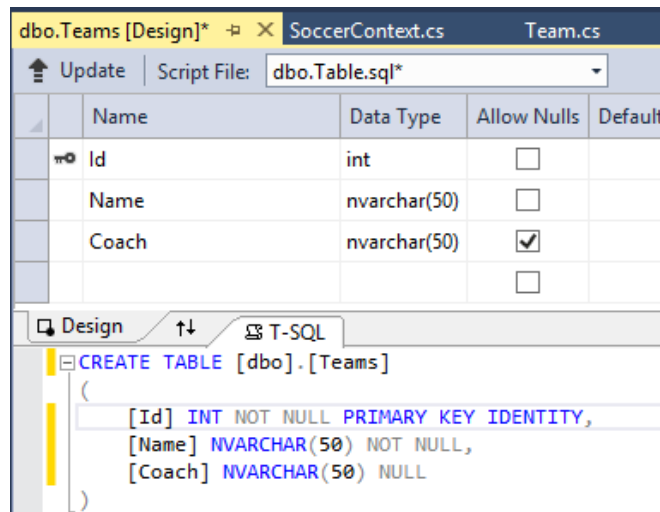


Рисунок 3.25 - Структура бази даних

Таблиця Players матиме наступну структуру:

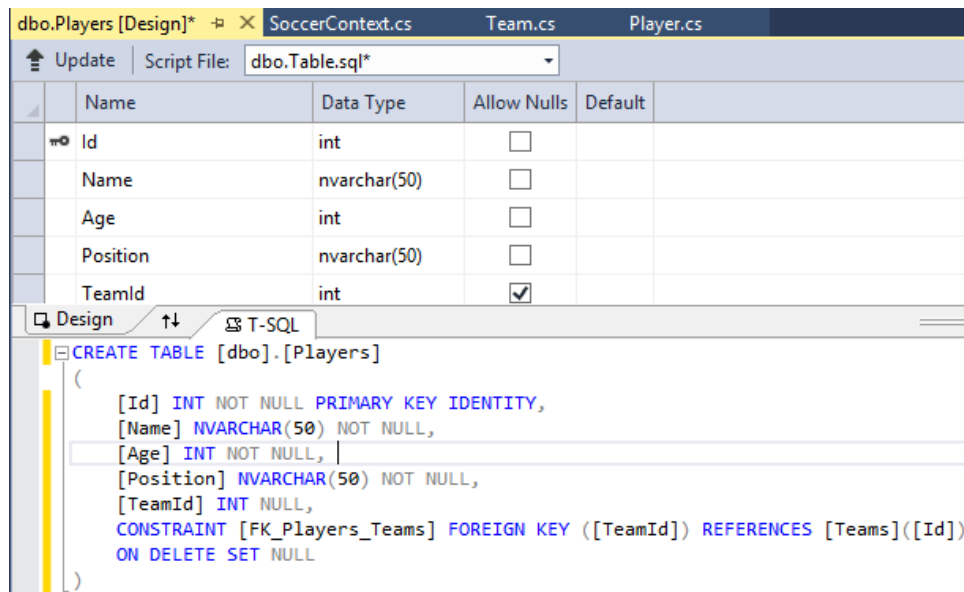


Рисунок 3.26 - Структура бази даних

На відміну від таблиці Teams тут ми також задаємо зовнішній ключ - властивість TeamId тепер буде посилатися на поле Id з таблиці Teams.

Щоб задати зовнішній ключ, ми додаємо в панелі SQL внизу під дизайном таблиці наступний рядок:

```
CONSTRAINT [FK_Players_Teams] FOREIGN KEY ([TeamId]) REFERENCES [Teams] ([Id]) ON DELETE SET NULL
```

Це звичайний вираз мови SQL, яке пов'язує стовпці двох таблиць. Остання частина цього виразу (ON DELETE SET NULL) вказує, що при видаленні об'єкта з таблиці Teams, властивості TeamId, яке посилалося на видалений об'єкт, буде присвоєно значення null.

Це треба, щоб гравці при видаленні команд не відносились більше до видалених команд. Однак ми можемо задати й іншу дію, наприклад, при видаленні команди видалити всіх її гравців. Для цього нам треба написати ON DELETE CASCADE. Тепер після визначення таблиць наповнимо їх деякими початковими даними. Додамо деякі дані в таблицю Teams:

dbo.Teams [Data] X SoccerContext.cs Team.cs			
Max Rows: 1000			
	Id	Name	Coach
	1	Реал Мадрид	Анчелотти
	2	Барселона	Мартіно
	3	Баварія	Гуардіола
	4	Боруссія	Клопп
»*	NULL	NULL	NULL

Рисунок 3.27 - Структура таблиці

І в таблицю Players (де стовпець TeamId містить деяке існуюче значення із стовпця Id таблиці Teams):

dbo.Players [Data] Max Rows: 1000					
	Id	Name	Age	Position	TeamId
	1	Месси	26	Нападающий	2
	2	Роналду	29	Нападающий	1
	3	Бейл	24	Полузащитник	1
	4	Неймар	22	Нападающий	2
	5	Иньеста	29	Полузащитник	2
	6	Рибери	30	Полузащитник	3
»*	NULL	NULL	NULL	NULL	NULL

Рисунок 3.28 - Структура таблиці

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

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using NavigationProperty.Models;
using System.Data.Entity;

namespace NavigationProperty.Controllers
{
    public class HomeController : Controller
    {
        SoccerContext db = new SoccerContext();

        // Виводимо всіх футболістів
        public ActionResult Index()
        {
            var players = db.Players.Include(p => p.Team);
            return View(players.ToList());
        }
    }
}
```

Тепер за допомогою методу Include фреймворк підвантажує для кожного гравця команду, пов'язану з даним гравцем через зовнішній ключ. Створимо представлення Index.cshtml, яке буде виводити всіх гравців:

```
@model IEnumerable<NavigationProperty.Models.Player>
@{
    ViewBag.Title = "Каталог гравців ";
}

<h2>Каталог гравців</h2>
<p>
    @Html.ActionLink("Добавить гравця", "Create")
</p>
<table>
    <tr>
```

```

        <th>Ім'я гравця</th>
        <th>Вік</th>
        <th>Позиція на полі</th>
        <th>Команда</th>
        <th></th>
    </tr>

    @foreach (var item in Model)
    {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Name)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Age)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Position)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Team.Name)
            </td>
            <td>
                @Html.ActionLink("Редагувати", "Edit", new { id = item.Id }) |
                @Html.ActionLink("Видалити", "Delete", new { id = item.Id })
            </td>
        </tr>
    }
</table>
<p>
    @Html.ActionLink("Каталог команд", "ListTeams")
</p>

```

Так як у методі контролера за допомогою методу Include до всіх моделей Player підключається свій об'єкт Team з навігаційною властивістю TeamId, то то в представленні ми можемо отримати цей зв'язаний об'єкт Team і використовувати його властивості, наприклад, отримати item.Team.Name для відображення імені команди. Подібним чином можна вивести список команд. В моделі Team є властивість Players, яка покликане зберігати пов'язаних з командою гравців. Використаємо його. Наприклад, виведемо всі дані про команду, в тому числі про її гравців. Спочатку додамо в контролер наступний метод:

```

public ActionResult TeamDetails(int? id)
{
    if (id == null)
    {
        return HttpNotFound();
    }
    Team team = db.Teams.Include(t=>t.Players).FirstOrDefault(t=>t.Id==id);
    if (team == null)

```

```

    {
        return HttpNotFound();
    }
    return View(team);
}

```

По-перше, щоб обробити ввід при відсутності переданого значення, як параметр використовуємо `int? id`. По-друге, ми підвантажуємо всіх гравців, пов'язаних з командою, у виразі `db.Teams.Include (t => t.Players) .FirstOrDefault (t => t.Id == id)`.

Представлення `TeamDetails.cshtml` для відображення даних про команду могло б виглядати так:

```

@using NavigationProperty.Models
@model Team

@{
    ViewBag.Title = "Команда " + @Model.Name;
    Layout = "~/Views/Shared/_Layout.cshtml";
}

<div>
    <h4>Команда @Model.Name</h4>
    <hr />
    <dl>
        <dt>Назва</dt>
        <dd>
            @Html.DisplayFor(model => model.Name)
        </dd>
        <dt>Тренер</dt>
        <dd>
            @Html.DisplayFor(model => model.Coach)
        </dd>
        <dt>Гравці</dt>
        <dd>
            <ul>
                @foreach (Player player in Model.Players)
                {
                    <li>@player.Name (@player.Position)</li>
                }
            </ul>
        </dd>
    </dl>
</div>

```

3.5.7 Операції для роботи зі складними моделями

У попередній темі ми створили дві моделі `Player` і `Team` і вивели елементи з таблиці `Players` на сторінку. Тепер подивимось, як проробити інші операції з моделями. Загалом-то тут буде все те ж саме, що і з звичайними моделями.

Єдина відмінність - ми повинні враховувати значення навігаційне властивість, наявне в складній моделі.

Додавання моделі

При додаванні моделі, що має зовнішній ключ, характерно все те ж саме, що і для звичайної моделі. Єдине доповнення - нам треба також передавати в представлення набір значень для зв'язку зовнішнього ключа з іншою таблицею. Отже, додамо в контролер наступну дію Create:

```
[HttpGet]
public ActionResult Create()
{
    // Формиємо список команд для передачі в представлення
    SelectList teams = new SelectList(db.Teams, "Id", "Name");
    ViewBag.Teams = teams;
    return View();
}

[HttpPost]
public ActionResult Create(Player player)
{
    //Добавляємо гравця в таблицю
    db.Players.Add(player);
    db.SaveChanges();
    // перенаправляємо на головну сторінку
    return RedirectToAction("Index");
}
```

Перший метод обробляє GET-запит і повертає представлення, передаючи в нього об'єкт SelectList - список всіх команд. Другий метод отримує введену користувачем у представленні модель (дані) і додає її в БД. А тепер створимо представлення Create.cshtml:

```
@model NavigationProperty.Models.Player

@{
    ViewBag.Title = "Додавання граця";
    Layout = "~/Views/Shared/_Layout.cshtml";
}

<h2>Додавання нового гравця</h2>

@using (Html.BeginForm())
{
    <fieldset>
        <legend>Футболіст</legend>

        <p>
            Ім'я гравця <br />
```

```

        @Html.EditorFor(model => model.Name)
    </p>

    <p>
        Вік <br />
        @Html.EditorFor(model => model.Age)
    </p>

    <p>
        Позиція на полі <br />
        @Html.EditorFor(model => model.Position)
    </p>
    <p>
        Команда <br />
        @Html.DropDownListFor(model => model.TeamId, ViewBag.Teams as SelectList)
    </p>

    <p>
        <input type="submit" value="Додати гравця" />
    </p>
</fieldset>
}
<div>
    @Html.ActionLink("До списку гравців", "Index")
</div>

```

Як і у випадку з простими моделями, ми прив'язуємо поля до певної властивості. Тут слід лише зауважити створення випадючого списку, з якого ми вибираємо команду. Обране значення в цьому списку прив'язується до властивості моделі TeamId.

Редагування моделі

Редагування працює подібним чином. Визначимо в контролері наступну дію Edit:

```

[HttpGet]
public ActionResult Edit(int? id)
{
    if (id == null)
    {
        return HttpNotFound();
    }
    // Знаходимо в БД футболіста
    Player player = db.Players.Find(id);
    if (player != null)
    {
        // Створимо список команд для передачі в представлення
        SelectList teams = new SelectList(db.Teams, "Id", "Name", player.TeamId);
        ViewBag.Teams = teams;
        return View(player);
    }
}

```

```

    }
    return RedirectToAction("Index");
}

[HttpPost]
public ActionResult Edit(Player player)
{
    db.Entry(player).State = EntityState.Modified;
    db.SaveChanges();
    return RedirectToAction("Index");
}

```

Тут також у вигляді об'єкта SelectList створюється список команд, який витягуються з БД. І після отримання запиту на редагування певної моделі Player контролер передає цю модель і список команд в представлення Edit.cshtml:

```

@model NavigationProperty.Models.Player
@{
    ViewBag.Title = "Edit";
    Layout = "~/Views/Shared/_Layout.cshtml";
}

```

```
<h2>Зміна гравця</h2>
```

```

@using (Html.BeginForm())
{
    <fieldset>
        <legend>Футболіст</legend>

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

        <p>Ім'я гравця <br />
            @Html.EditorFor(model => model.Name)
        </p>

        <p>Вік <br />
            @Html.EditorFor(model => model.Age)
        </p>

        <p>Позиція на полі <br />
            @Html.EditorFor(model => model.Position)
        </p>
        <p>Команда <br />
            @Html.DropDownListFor(model => model.TeamId, ViewBag.Teams as SelectList)
        </p>
        <p>
            <input type="submit" value="Зберегти" />
        </p>
    </fieldset>
}
<div>

```

```
@Html.ActionLink("Повернутися до списку футболістів", "Index")
</div>
```

Видалення даних проводиться так само, як і у випадку з звичайною моделлю.

3.5.8 Моделі зі зв'язками «багато-до-багатьох»

Крім моделей зі зв'язками «один-до-одного» і «один-до-багатьох», які були розглянуті в минулих темах, існують моделі за типом «багато-до-багатьох». Якщо подивитися на світ навколо себе, то ми зможемо знайти подібні моделі. Найбільш поширений і хрестоматійний приклад - навчання в університеті, де різна кількість студентів може відвідувати різну кількість дисциплін. І при цьому у нас може виникнути необхідність ведення обліку студентів з конкретної дисципліни, і водночас ведення обліку різних дисциплін для конкретного студента. Спробуємо змодельовати дану ситуацію в додатку на базі ASP.NET MVC5.

По-перше, створимо новий проект. Отже, у нас є дві моделі - студент і курс університетської дисципліни. Спочатку додамо модель Student з наступним вмістом:

```
public class Student
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Surname { get; set; }

    public virtual ICollection<Course> Courses { get; set; }
    public Student()
    {
        Courses = new List<Course>();
    }
}
```

І також додамо модель Course:

```
public class Course
{
    public int Id { get; set; }
    public string Name { get; set; }

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

    public Course()
    {
        Students = new List<Student>();
    }
}
```

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

```
public class StudentsContext : DbContext
{
    public DbSet<Student> Students { get; set; }
    public DbSet<Course> Courses { get; set; }

    public StudentsContext() : base("DefaultConnection")
    {}

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Course>().HasMany(c => c.Students)
            .WithMany(s => s.Courses)
            .Map(t => t.MapLeftKey("CourseId"))
            .MapRightKey("StudentId")
            .ToTable("CourseStudent");
    }
}
```

По-перше, щоб використовувати рядок підключення за замовчуванням, встановимо для неї контекст даних в конструкторі:

```
public StudentsContext () : base ("DefaultConnection")
```

У створюваній базі даних всі дані про студентів будуть зберігатися в таблиці **Students**, а дані про університетських курсах - у таблиці **Courses**. Але ці таблиця повинні бути пов'язані зв'язком «багато-до-багатьох». І цей зв'язок забезпечить ще одна таблиця, яка буде називатися **CourseStudent**. Для побудови цієї таблиці ми перевизначайте метод OnModelCreating, в якому за допомогою об'єкта modelBuilder створюємо нову таблицю і визначаємо її поля. Одне її поле - **CourseId** - буде посилатися на таблицю **Courses** і зберігати в собі **id** курсу. А друге поле - **StudentId** - буде посилатися на таблицю студентів і зберігати **id** студента. У результаті у нас вийде набір пар **id курсу - id студента**, завдяки цьому ми зможемо визначити зв'язок «багато-до-багатьох». Тепер проініціалізуємо базу даних початковими даними. Додамо в проект наступний клас:

```
public class CourseDbInitializer : DropCreateDatabaseAlways<StudentsContext>
{
    protected override void Seed(StudentsContext context)
    {
        Student s1 = new Student { Id = 1, Name = "Ігор", Surname = "Іванов" };
        Student s2 = new Student { Id = 2, Name = "Марія", Surname = "Василюк" };
        Student s3 = new Student { Id = 3, Name = "Олег", Surname = "Кендьо" };
        Student s4 = new Student { Id = 4, Name = "Ольга", Surname = "Семчук" };
    }
}
```



```

context.Students.Add(s1);
context.Students.Add(s2);
context.Students.Add(s3);
context.Students.Add(s4);

Course c1 = new Course { Id = 1, Name = "Операційні системи",
    Students = new List<Student>() { s1, s2, s3 } };
Course c2 = new Course { Id = 2, Name = "Алгоритми і структури даних",
    Students = new List<Student>() { s2, s4 } };
Course c3 = new Course { Id = 3, Name = "Основи HTML и CSS",
    Students = new List<Student>() { s3, s4, s1 } };

context.Courses.Add(c1);
context.Courses.Add(c2);
context.Courses.Add(c3);

base.Seed(context);
}
}

```

Зверніть увагу, як забезпечується зв'язок між курсами і студентами: ми просто додаємо набір створених студентів в колекцію Students для кожного курсу. І щоб усе це запрацювало, додамо в файл Global.asax.cs в метод Application_Start наступний рядок:

```
Database.SetInitializer(new CourseDbInitializer());
```

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

```

using System;
using System.Collections.Generic;
using System.Data;
using System.Data.Entity;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using TeamMvc4.Models;

namespace TeamMvc4.Controllers
{
    public class HomeController : Controller
    {
        private StudentsContext db = new StudentsContext();

        public ActionResult Index()
        {

```

```

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

    public ActionResult Details(int id = 0)
    {
        Student student = db.Students.Find(id);
        if (student == null)
        {
            return HttpNotFound();
        }
        return View(student);
    }

    protected override void Dispose(bool disposing)
    {
        db.Dispose();
        base.Dispose(disposing);
    }
}

```

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

```

@using TeamMvc4.Models
@model Student
@{
    ViewBag.Title = "Details";
}

<fieldset>
    <legend>Інформацію про студента</legend>

    <div class="display-label"><b>Имя</b></div>
    <div class="display-field">
        @Html.DisplayFor(model => model.Name)
    </div>

    <div class="display-label"><b>Прізвище</b></div>
    <div class="display-field">
        @Html.DisplayFor(model => model.Surname)
    </div>

    <div class="display-label"><b>Курсы</b></div>
    <ul>
        @foreach (Course c in Model.Courses)
        {
            <li>@c.Name</li>

```

```

    }
</ul>

```

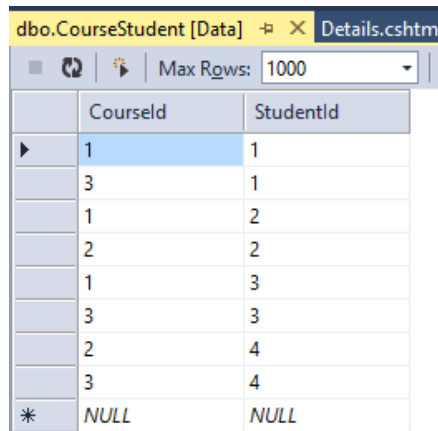
```

</fieldset>

```

Оскільки до моделі автоматично приєднуються пов'язані дані, то ми можемо їх отримати через **Model.Courses**. Так як у нас в базі даних вже є початкові дані, то можемо отримати інформацію по першому студенту, відправивши запит **Home/Details/1**.

Якщо ж ми подивимося на базу даних, то виявимо в ній, як і очікувалося, три таблиці. Єднальна таблиця **CourseStudent** матиме такий вміст:



	CourseId	StudentId
▶	1	1
	3	1
	1	2
	2	2
	1	3
	3	3
	2	4
	3	4
*	NULL	NULL

Рисунок 3.29 - Структура таблиці

Тобто всі ті ж дані, які ми вказали для початкової ініціалізації бази даних. Додамо в контролер **HomeController** наступну дію **Edit**:

```

public ActionResult Edit(int id = 0)
{
    Student student = db.Students.Find(id);
    if (student == null)
    {
        return HttpNotFound();
    }
    ViewBag.Courses = db.Courses.ToList();
    return View(student);
}

[HttpPost]
public ActionResult Edit(Student student, int[] selectedCourses)
{
    Student newStudent = db.Students.Find(student.Id);
    newStudent.Name = student.Name;
    newStudent.Surname = student.Surname;

    newStudent.Courses.Clear();
    if (selectedCourses != null)
    {

```

```

        //одержимо вибрані курси
        foreach (var c in db.Courses.Where(co => selectedCourses.Contains(co.Id)))
        {
            newStudent.Courses.Add(c);
        }
    }

    db.Entry(newStudent).State = EntityState.Modified;
    db.SaveChanges();
    return RedirectToAction("Index");
}

```

Дія Edit представляє два методу - для запиту get і для запиту post. Метод, що обробляє запит get, стандартний - він передає в представлення редаковувану модель, а також список всіх курсів через ViewBag.Courses = db.Courses.ToList(); щоб ми могли потім ці курси вивести в представленні.

Метод **post** приймає отримані дані, тільки окрім моделі Student сюди також передаються всі вибрані курси у вигляді масиву **id** курсів. У цей масив і будуть міститися всі значення всіх зазначених на формі флагів. У самому методі після проходження валідації ми встановлюємо нові значення властивостей моделі.

Потім нам треба встановити в колекції Courses у студента всі відмічені курси. Для цього проходимо по всіх курсах з бази даних, і якщо вони були відзначені на формі і відсутні у списку, то додаємо їх. Невідмічені видаляємо (якщо вони є у списку). У підсумку інформація в базі даних буде відповідне чином оновлена. І наостанок додамо саме представлення Edit.cshtml:

```

@using TeamMvc4.Models
@model Student
@{
    ViewBag.Title = "Edit";
}

```

```

@using (Html.BeginForm()) {

    <fieldset>
        <legend>Студент</legend>

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

        <div class="editor-label"><b>Ім'я</b></div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Name)
        </div>

        <div class="editor-label"><b>Прізвище</b></div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Surname)
        </div>

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

```

```

@foreach (Course c in ViewBag.Courses)
{
    <input type="checkbox" name="selectedCourses" value="@c.Id"
        @(Model.Courses.Contains(c) ? "checked=\"checked\"" : "") />@c.Name <br />
}

<p>
    <input type="submit" value="Зберегти" />
</p>
</fieldset>
}

```

Збереження моделі буде багато в чому ідентичне редагуванню.

3.5.9 Передача масивів і складних даних в контролер

З попередніх тим ми дізналися, як передавати окремі об'єкти з представлення в методи дії контролера в якості параметрів. Але в реальності може виникнути ситуація, коли буде потрібно передати в метод не один об'єкт типу `int` або дані якої-небудь моделі, а відразу декілька об'єктів. Розглянемо деякі можливі випадки.

Передача колекції

Визначимо наступну форму в представленні:

```

@using (Html.BeginForm())
{
    @Html.TextBox("names")
    @Html.TextBox("names")
    @Html.TextBox("names")
    @Html.TextBox("names")
    <input type="submit" />
}

```

Таким чином, у нас в html-розмітці буде створено чотири елемента `input`:

```

<form action="/Home/Array" method="post">
<input id="names" name="names" type="text" value="" />
<input id="names" name="names" type="text" value="" />
<input id="names" name="names" type="text" value="" />
<input id="names" name="names" type="text" value="" />
    <input type="submit" />
</form>

```

Тому при відправці форми буде формуватися колекція з **names**, що складається з чотирьох елементів, і в методі контролера ми зможемо отримати всі ці елементи:

```

[HttpPost]
public string Array(List<string> names)
{

```

```

string fin="";
for (int i = 0; i < names.Count; i++)
{
    fin += names[i] + " ";
}
return fin;
}

```

Передача колекції об'єктів моделі

Ми можемо передати в представлення масив об'єктів деякої моделі, наприклад, моделі Book:

```

[HttpGet]
public ActionResult Add()
{
    return View(db.Books.ToList());
}

```

Щоб вивести об'єкти для редагування в представлення ми можемо використовувати наступну конструкцію:

```

@model List<BookStore.Models.Book>
.....
@using (Html.BeginForm())
{
    for(int i=0; i<Model.Count; i++)
    {
        <h4>Книга № @(i+1)</h4>

        @: Name: @Html.EditorFor(m=>m[i].Name)
        @: Author: @Html.EditorFor(m => m[i].Author)
        @: Price: @Html.EditorFor(m => m[i].Price)
    }
    <input type="submit" />
}

```

Так ми генеруємо для кожного об'єкта набір полів для редагування його властивостей. І після натискання кнопки весь даний масив відправиться на сервер, де його можна отримати таким чином:

```

[HttpPost]
public string Add(List<Book> books)
{
    //.....
}

```

Передача різних об'єктів однієї моделі

У попередньому випадку ми передавали колекцію об'єктів моделі Book. Однак може виникнути ситуація, коли ми повинні розмежувати якимось передані об'єкти, а не розглядати їх як одну колекцію. Наприклад, метод контролера може виглядати так:

```
[HttpPost]
public string Add(Book book, Book myBook)
{
    //.....
}
```

Ми використовуємо два окремих об'єкти Book. Як ми можемо передати їх з представлення в контролер? Припустимо, тепер, що один об'єкт ми передаємо в якості моделі представлення, а інший створюємо в тому ж представленні:

```
@using BookStore.Models
@model Book
@{
    ViewBag.Title = "Array";
}
@{
    Book myBook = new Book() { Name = "Кобзар", Author = "Тарас Шевченко", Price = 190 };
}
<h2>Книги</h2>

@using (Html.BeginForm())
{
    @Html.EditorFor(m=>myBook)
    @Html.EditorForModel()
    <input type="submit" />
}
```

Одну модель ми передаємо з контролера в представлення:

```
[HttpGet]
public ActionResult Array()
{
    Book firstBook = db.Books.ToList<Book>().FirstOrDefault();
    return View(firstBook);
}
```

Другу модель - myBook ми створюємо вже в самому поданні. Всі поля моделі генеруються за допомогою хелпера @ Html.EditorFor (m => myBook). Зверніть увагу на ім'я моделі - myBook. Так як ми очікуємо, що даний об'єкт повинен бути переданий в метод як параметр myBook, тому він повинен мати точно таке ж ім'я, а не довільне. Таким чином, ми можемо передати на сервер два різні об'єкти однієї моделі.

Передача складних об'єктів

Припустимо, у нас є наступна модель:

```
public class Book
{
    public int Id { get; set; }
    public string Name { get; set; }
}
public class Author
{
    public int Id { get; set; }
    public string Name { get; set; }
    public ICollection<Book> Books { get; set; }
}
```

Модель автора містить посилання на колекцію книг. У контролері ми отримуємо цю модель:

```
[HttpPost]
public ActionResult GetAuthor(Author author)
{
    return View();
}
```

Тоді представлення могло б виглядати так:

```
@model ArrayPostApp.Models.Author
```

```
<h2>Додавання книг</h2>
```

```
@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()
    <div class="authorBlock">
        <label>Ім'я автора</label>
        <input type="text" name="name" />
    </div>
    <div id="booksBlock">
        <div class="bookItem">
            <h4>Книга № 1</h4>
            <div>
                <label>Назва</label>
                <div>
                    <input type="text" name="Books[0].name" />
                </div>
            </div>
        </div>

        </div>
        <p><a class="addLink">Додати новий елемент</a></p>
        <p><input type="submit" value="Додати" /></p>
    }
}
```



```

@section Scripts {
    <script>
        $(function () {
            var i = 0;
            $('addLink').click(function()
            {
                i++;
                var html2Add= "<div class='bookItem'>" +
                "<h4>Книга № "+(i+1) + "</h4>" +
                "<div><label>Назва</label><div>" +
                "<input type='text' name='Books["+i+"]name' />" +
                "</div></div></div>";
                $('#booksBlock').append(html2Add);
            })
        })
    </script>
}

```

Так як модель Author містить набір книг у властивості Books, то атрибут name відповідних текстових елементів повинен мати таку форму: name="Books[0].name". За допомогою коду javascript ми можемо динамічно додати нові елементи.

3.5.10 Міграція баз даних

Нерідко виникає ситуація, коли модель змінюється. Наприклад, ми вирішили внести в неї нові властивості. Але при цьому у нас вже є існуюча база даних, в якій є якісь дані. І щоб без втрат оновити базу даних ASP.NET MVC пропонує нам такий механізм як «міграції». Наприклад, у нас є проста модель User:

```

public class User
{
    public int Id { get; set; }
    public string Name { get; set; }
}

```

Відповідно є контекст даних, через який ми працюємо з БД:

```

public class UserContext : DbContext
{
    public UserContext() :
        base("DefaultConnection")
    {}
    public DbSet<User> Users { get; set; }
}

```

І припустимо, у нас є вся інфраструктура для роботи з цією моделлю - представлення, контролери, і також у нас є вже в базі даних декілька об'єктів

даної моделі. Але в якийсь момент ми вирішили змінити модельну базу програми. Наприклад, ми додали ще одне поле в модель User:

```
public class User
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }
}
```

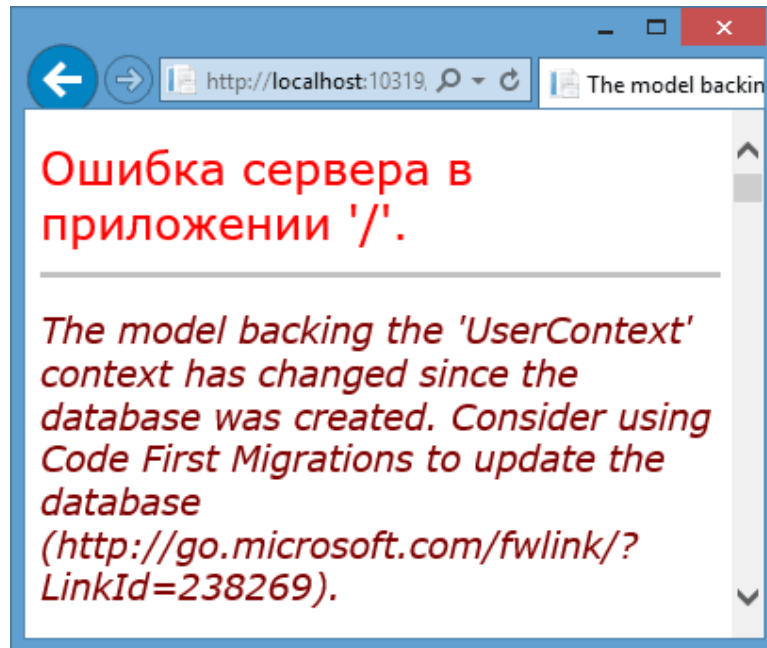
Крім того, ми вирішили додати ще одну модель, наприклад:

```
public class Company
{
    public int Id { get; set; }
    public string Name { get; set; }
}
```

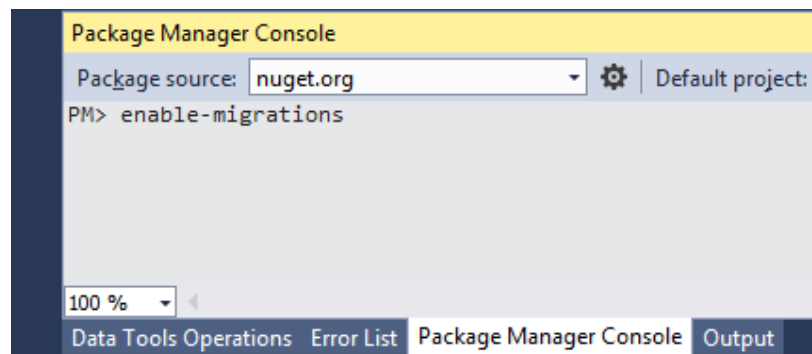
Отже, контекст даних у нас вже змінюється наступним чином:

```
public class UserContext : DbContext
{
    public UserContext() :
        base("DefaultConnection")
    { }
    public DbSet<User> Users { get; set; }
    public DbSet<Company> Companies { get; set; }
}
```

Ми можемо додати в представлення для моделі User додаткове поле для властивості Age, можемо створити контролер та представлення для нової моделі, але при спробі додати новий об'єкт в БД, ми будемо отримувати помилку:



Контекст даних змінився, і тепер нам треба провести міграцію від старої схеми бази даних до нової. І спершу знайдемо внизу Visual Studio вікно **Package Manager Console**, введемо в ньому команду: **enable-migrations** і натиснемо Enter:



Після виконання цієї команди Visual Studio в проєкті буде створена папка Migrations, в якій можна знайти файл Configuration.cs. Цей файл містить оголошення однойменного класу Configuration, який встановлює налаштування конфігурації:

```
namespace MigrationApp.Migrations
{
    using System;
    using System.Data.Entity;
    using System.Data.Entity.Migrations;
    using System.Linq;

    internal sealed class Configuration :
        DbMigrationsConfiguration<MigrationApp.Models.UserContext>
    {
        public Configuration()
```

```

{
    AutomaticMigrationsEnabled = false;
    ContextKey = "MigrationApp.Models.UserContext";
}

protected override void Seed(MigrationApp.Models.UserContext context)
{
}
}
}

```

У методі Seed можна проініціалізувати базу даних початковими даними. Тепер нам треба створити саму міграцію. Там же в консолі Package Manager Console введемо команду:

```
PM> Add-Migration "MigrateDB"
```

Після цього Visual Studio автоматично генерує клас міграції:

```

namespace MigrationApp.Migrations
{
    using System;
    using System.Data.Entity.Migrations;

    public partial class MigrateDB : DbMigration
    {
        public override void Up()
        {
            CreateTable(
                "dbo.Companies",
                c => new
                {
                    Id = c.Int(nullable: false, identity: true),
                    Name = c.String(),
                })
                .PrimaryKey(t => t.Id);

            AddColumn("dbo.Users", "Age", c => c.Int(nullable: false));
        }

        public override void Down()
        {
            DropColumn("dbo.Users", "Age");
            DropTable("dbo.Companies");
        }
    }
}

```

У методі Up за допомогою виклику методу Create Table створюється таблиця "dbo.Companies" і проводиться її настройка: створення стовпців, установка ключів. І також додається новий стовпець Age у вже наявну таблицю. Метод Down видаляє стовпець і таблицю на випадок, якщо вони існують.

Фактично ці методи рівнозначні висловом ALTER в мові SQL, яке змінює структуру бази даних і її таблиць.

І в завершенні щоб виконати міграцію, застосуємо цей клас, набравши в тій же консолі команду:

```
PM> Update-Database
```

Після цього, якщо ми подивимося на склад бази даних, то побачимо, що до неї були застосовані зміни відповідно до виконаної міграції:

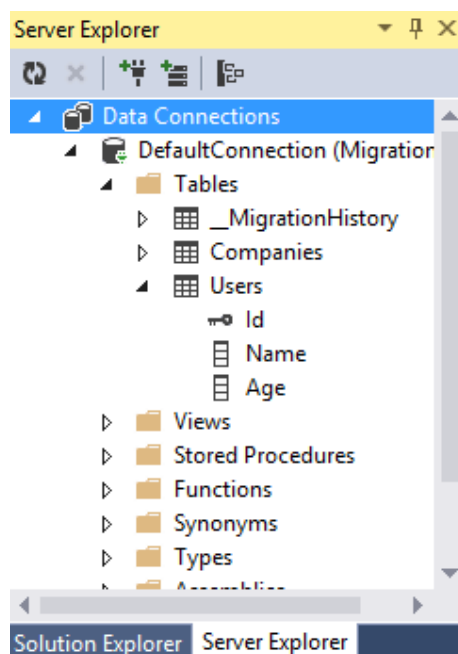


Рисунок 3.30 - Серверний оглядач

Отже, міграція виконана, і ми можемо використовувати оновлені моделі і контекст даних.

3.5.11 Створення блочних сторінок

Однією з поширених завдань при роботі з даними є проблема пагінації або розділення даних на кілька блоків - сторінок, які виводяться в представлення окремо і які супроводжуються зручною навігацією по сторінках. Для створення пагінацію ми можемо скористатися готовими плагінами. Але в даному випадку ми самі створимо механізм для посторінкового виводу.

Спочатку створимо модель, дані якої будемо виводити:

```
public class Phone
{
    public int Id { get; set; }
    public string Model { get;set;}
    public string Producer { get; set; }
}
```

Також визначаємо модель, яка буде описувати механізм пагінації], а також додаткову модель для виведення даних в представлення:

```
public class PageInfo
{
    public int PageNumber { get; set; } // номер поточної сторінки
    public int PageSize { get; set; } // кількість об'єктів на сторінці
    public int TotalItems { get; set; } // всього об'єктів
    public int TotalPages // всього сторінок
    {
        get { return (int)Math.Ceiling((decimal)TotalItems / PageSize); }
    }
}
```

```
public class IndexViewModel
{
    public IEnumerable<Phone> Phones { get; set; }
    public PageInfo PageInfo { get; set; }
}
```

Так як разом з даними моделей телефонів потрібно також використовувати в поданні модель пагінацію, то вони інкапсулюються класом IndexViewModel.

Тепер визначимо контролер і його метод для виведення даних:

```
public class HomeController: Controller
{
    List<Phone> phones;

    public HomeController ()
    {
        phones = new List<Phone> ();
        phones.Add (new Phone {Id = 1, Model = "Samsung Galaxy III", Producer = "Samsung"});
        phones.Add (new Phone {Id = 2, Model = "Samsung Ace II", Producer = "Samsung"});
        phones.Add (new Phone {Id = 3, Model = "HTC Hero", Producer = "HTC"});
        phones.Add (new Phone {Id = 4, Model = "HTC One S", Producer = "HTC"});
        phones.Add (new Phone {Id = 5, Model = "HTC One X", Producer = "HTC"});
        phones.Add (new Phone {Id = 6, Model = "LG Optimus 3D", Producer = "LG"});
        phones.Add (new Phone {Id = 7, Model = "Nokia N9", Producer = "Nokia"});
        phones.Add (new Phone {Id = 8, Model = "Samsung Galaxy Nexus", Producer = "Samsung"});
        phones.Add (new Phone {Id = 9, Model = "Sony Xperia X10", Producer = "SONY"});
        phones.Add (new Phone {Id = 10, Model = "Samsung Galaxy II", Producer = "Samsung"});
    }
    public ActionResult Index (int page = 1)
    {
        int pageSize = 3; // Кількість об'єктів на сторінку
        IEnumerable<Phone> phonesPerPages = phones.Skip ((page - 1) * pageSize) .Take (pageSize);
        PageInfo pageInfo = new PageInfo {PageNumber = page, PageSize = pageSize, TotalItems =
phones.Count};
```

```

        IndexViewModel ivm = new IndexViewModel {PageInfo = pageInfo, Phones = phonesPerPage};
        return View (ivm);
    }
}

```

Для простоти прикладу створюємо звичайний список в конструкторі контролера, проте це міг би бути і вивід з БД. Метод Index як параметр приймає номер сторінки. За замовчуванням номер сторінки буде дорівнювати одиниці. У самому методі за допомогою комбінації методів Skip і Take відбувається відбір потрібної частини даних: метод Skip() пропускає певну кількість даних, яке передається в параметрі, а метод Take() витягує певну кількість. Для створення механізму пагінації в поданні зручно використовувати хелпери. Створимо власний хелпер. Для цього додамо в проект папку Helpers і потім додамо в неї наступний клас:

```

using System.Text;
using System.Web;
using System.Web.Mvc;
//.....
public static class PagingHelpers
{
    public static MvcHtmlString PageLinks(this HtmlHelper html,
        PageInfo pageInfo, Func<int, string> pageUrl)
    {
        StringBuilder result = new StringBuilder();
        for (int i = 1; i <= pageInfo.TotalPages; i++)
        {
            TagBuilder tag = new TagBuilder("a");
            tag.MergeAttribute("href", pageUrl(i));
            tag.InnerHtml = i.ToString();
            // якщо це поточна сторінка то виділяємо її,
            // наприклад добавляючи клас
            if (i == pageInfo.PageNumber)
            {
                tag.AddCssClass("selected");
                tag.AddCssClass("btn-primary");
            }
            tag.AddCssClass("btn btn-default");
            result.Append(tag.ToString());
        }
        return MvcHtmlString.Create(result.ToString());
    }
}

```

Даний хелпер просто створити блок посилань, а також додає їм класи для візуалізації. Класи можуть бути будь-якими, але в даному випадку використовувалися стандартні класи bootstrap.

І тепер визначимо представлення, яке використовувати пагінацію:

```

@model ManualPaginApp.Models.IndexViewModel
@using ManualPaginApp.Helpers
@{
    ViewBag.Title = "Home Page";
}
<table class="table">
    <tr>
        <td>Модель</td>
        <td>Виробник</td>
    </tr>
    @foreach (var item in Model.Phones)
    {
        <tr>
            <td>@item.Model</td>
            <td>@item.Producer</td>
        </tr>
    }
</table>
<br />
<div class="btn-group">
    @Html.PageLinks(Model.PageInfo, x => Url.Action("Index",new { page = x}))
</div>

```

Оскільки хелпер пагінації знаходиться в папці `Helpers`, то необхідно спочатку представлення підключити даний простір імен `using ManualPaginApp.Helpers`. І хелпер пагінацію `Html.PageLinks` зроблять всю роботу з формування блоку посилань в відповідності із закладеною в ньому логікою.

3.5.12 Перевизначення шаблонів формування

Раніше ми розглянули функціональність шаблонів формування, які дозволяють автоматизувати процес створення контролера і представлень. Однак іноді їх функціональності буває недостатньо, і виникає необхідність перевизначити їх поведінку.

Для початку розглянемо, що представляють собою наявні шаблони. Всі шаблони для Visual Studio 2013 зберігаються в папці `C:\Program Files(x86)\Microsoft Visual Studio12.0\ Common7\ IDE\ Extensions\ Microsoft\ Web\ Mvc\ Scaffolding\ Templates`.

Ця папка містить наступні підкаталоги:

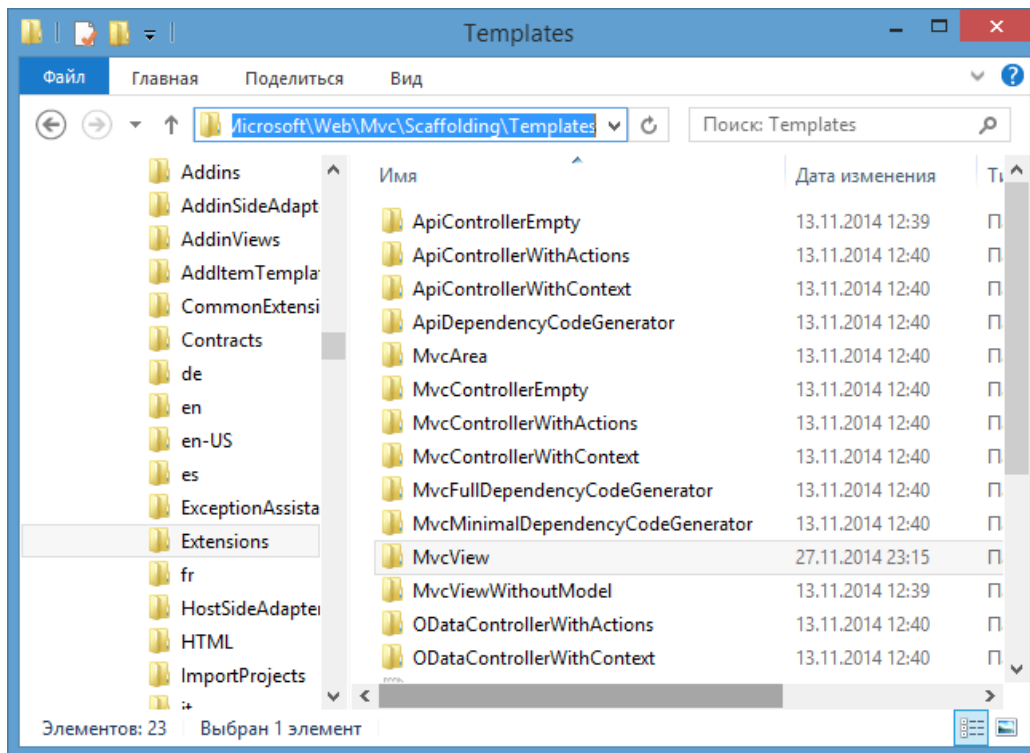


Рисунок 3.31 - Каталог шаблонів

Всі підкаталоги представляють шаблони, на основі яких генерується код контролерів або представлень. Зокрема, каталог `MvcView` містить шаблони для генерації представлень:

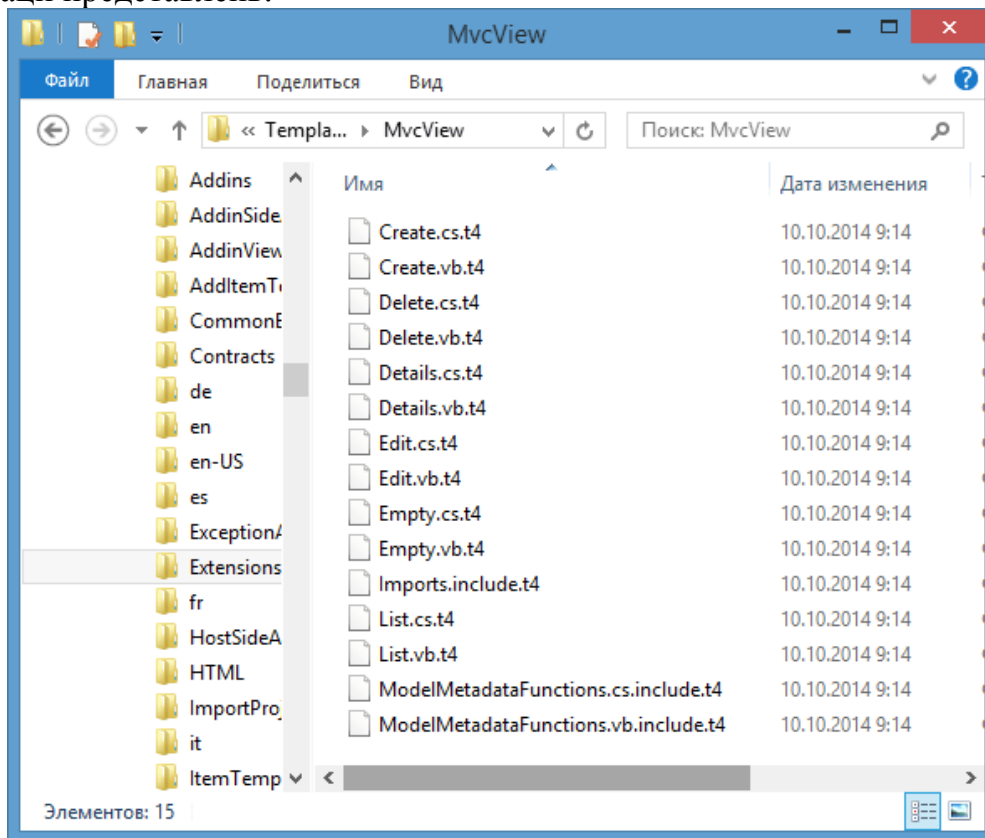


Рисунок 3.32 - Каталог шаблонів

Шаблон являє собою файл з розширенням .t4, який містить ряд директив, за якими потім генерується код. Наприклад, найменший шаблон Empty.cs.t4 має наступний код:

```
<#@ template language="C#" HostSpecific="True" #>
<#@ output extension=".cshtml" #>
<#@ include file="Imports.include.t4" #>
@model <#= ViewDataTypeName #>
<#
// The following chained if-statement outputs the file header code and markup for a partial view, a
view using a layout page, or a regular view.
if(IsPartialView) {
#>

<#
} else if(IsLayoutPageSelected) {
#>

@{
    ViewBag.Title = "<#= ViewName#>";
<#
if (!String.IsNullOrEmpty(LayoutPageFile)) {
#>
    Layout = "<#= LayoutPageFile#>";
<#
}
#>
}

<h2><#= ViewName#></h2>

<#
} else {
#>

@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title><#= ViewName #></title>
</head>
<body>
<#
    PushIndent("  ");
}
```

```

#>
<#
if(!IsPartialView && !IsLayoutPageSelected) {
#>
<div>
</div>
<#
}
#>
<#
// The following code closes the tag used in the case of a view using a layout page and the body and
html tags in the case of a regular view page
#>
<#
if(!IsPartialView && !IsLayoutPageSelected) {
    ClearIndent();
#>
</body>
</html>
<#
}
#>

```

Всі директиви і керуючі блоки шаблонів містяться між тегами <# і #>. Наприклад, директива <# @ output extension = ". Cshtml" #> вказує, що вихідний файл після обробки повинен мати розширення .cshtml. Блоки звичайного тексту розташовуються поза тегів <# і #>. Блоки тексту не обробляються і поміщаються у вихідний файл без змін. Ми можемо змінити будь-якої шаблон з існуючих, однак зміна торкнеться всіх наступних випадків використання даного шаблону. Для того, щоб не змінювати налаштувань глобально, ми можемо додати потрібні шаблони в проект в Visual Studio. Шаблони з проекту в Visual Studio перевизначають поведінку глобальних шаблонів з каталогу C:\Program Files(x86)\ Microsoft Visual Studio 12.0\ Common7\ IDE\ Extensions\ Microsoft\ Web\ Mvc\ Scaffolding\ Templates для даного проекту.

Щоб перевизначити шаблон, нам треба створити в проекті папку з ім'ям CodeTemplates. І розмістити в ній ті шаблони, які треба перевизначити. Наприклад, щоб перевизначити шаблон Delete.cs.t4:

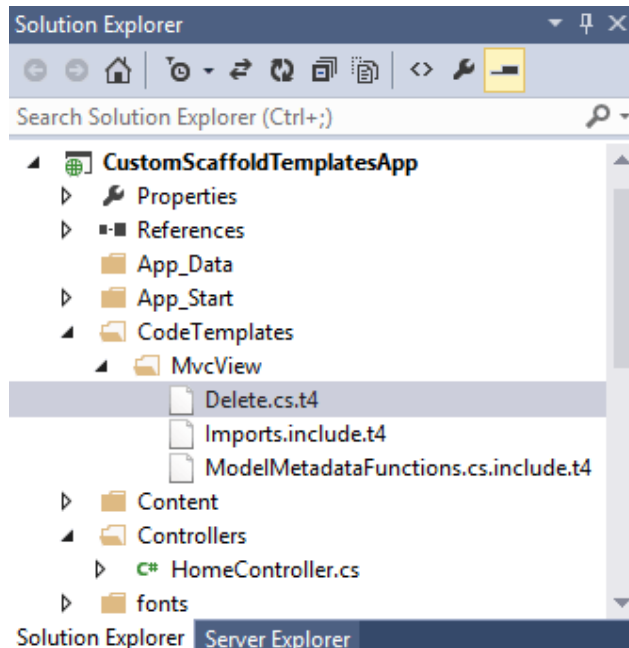


Рисунок 3.33 - Перевизначення шаблону

Оскільки шаблон Delete.cs.t4 знаходиться в папці Mvc View, то в проєкті треба створити відповідну папку в CodeTemplates і потім скопіювати туди шаблон. Крім того, додамо додатково пару файлів, від яких залежить Delete.cs.t4 і які також знаходяться в папці C:\ Program Files (x86)\ Microsoft Visual Studio 12.0\ Common7\ IDE\ Extensions\ Microsoft\ Web\ Mvc\ Scaffolding\ Templates \ MvcView. Тепер змінимо код шаблону Delete.cs.t4 на наступний:

```
<#@ template language="C#" HostSpecific="True" #>
<#@ output extension=".cshtml" #>
<#@ include file="Imports.include.t4" #>
<h2>Шаблон видалення</h2>
```

Цей шаблон по суті нічого не робить крім того, що виводить заголовок. Перша директива вказує, що буде використовуватися мова C#. Другий рядок говорить, що вихідний файл буде мати розширення .cshtml, а третій рядок підключає файл. Тепер, якщо ми спробуємо створити представлення за шаблоном Delete, то Visual Studio згенерує файл з наступним вмістом:

```
<h2> Шаблон видалення </ h2>
```

Тепер створимо більш осмислений шаблон. Для цього додаємо в проєкт в папку CodeTemplates/MvcView який-небудь файл або інший шаблон і перейменуємо його на MyTemplate:

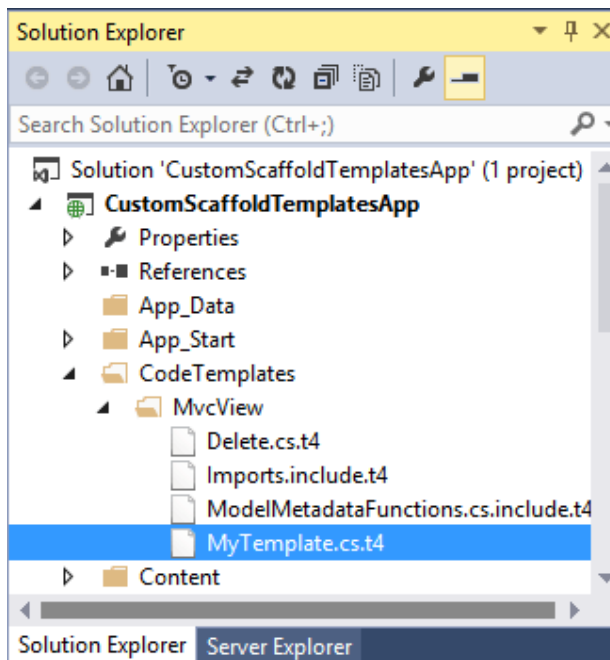


Рисунок 3.34 - Створення шаблону

Визначимо в ньому наступний код:

```
<#@ template language="C#" HostSpecific="True" #>
<#@ output extension=".cshtml" #>
<#@ include file="Imports.include.t4" #>
@model <#= ViewDataTypeName #>

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>MyTemplate</title>
</head>
<body>
<h2><#= ViewName#></h2>
<table>
<#
foreach (PropertyMetadata property in ModelMetadata.Properties) {
    if (property.Scaffold && !property.IsPrimaryKey && !property.IsForeignKey) {
#>
        <tr>
            <td>
                <b> @Html.DisplayNameFor(model => model.<#= GetValueExpression(property) #>) </b>
            </td>
            <td>
                @Html.DisplayFor(model => model.<#= GetValueExpression(property) #>)
            </td>
        </tr>
    }
}
```

Цей шаблон приймає модель з контролера і виводить всі значення її властивостей у вигляді таблиці. Тепер ми можемо використовувати цей шаблон при створенні представлень:

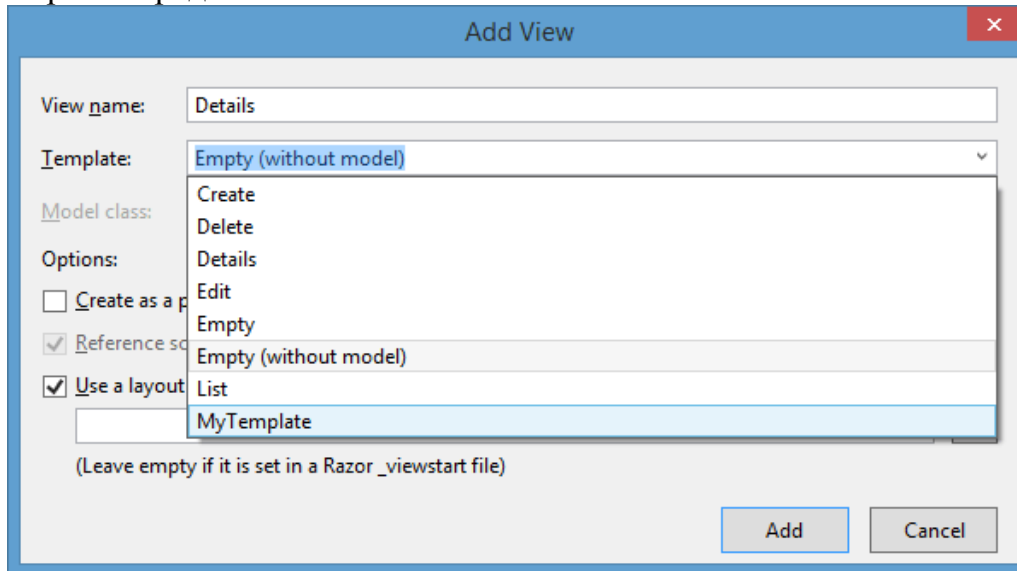


Рисунок 3.35 - Визначення шаблону

3.5.13 Перевизначення шаблонів відображення і редагування

При використанні методів `DisplayFor()`/`EditorFor()` фреймворк MVC сам визначає, яким чином і яку розмітку html створювати для відображення і редагування полів моделі. Але ми можемо перевизначити подібну поведінку, вказавши, як саме MVC повинен працювати з окремими типами даних. Припустимо, у нас є наступна модель:

```
public class Book
{
    public int Id { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
}
```

Модель містить властивості трьох типів даних: `int`, `string`, `decimal`. Перевизначимо створення розмітки для цих властивостей. Для цього додамо в проект в каталог `Views/Shared` дві нові папки: `DisplayTemplates` (для шаблонів відображення) і `EditorTemplates` (для шаблонів редагування):

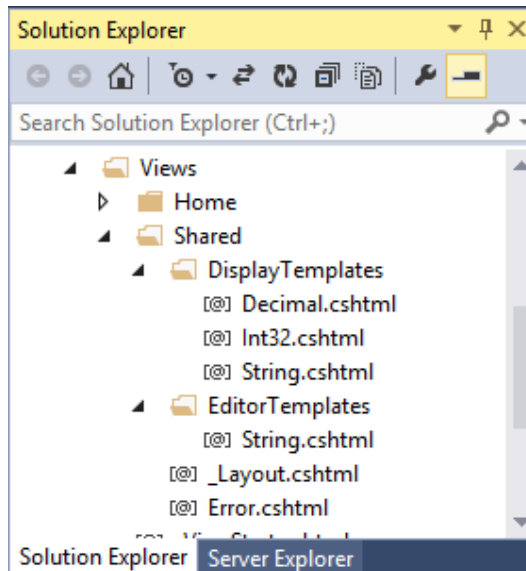


Рисунок 3.36 - Додавання додаткових папок

Оскільки клас використовує властивості трьох різних типів даних, в шаблонах відображення в папці `DisplayTemplates` створимо по файлу для кожного типу. Назви файлів носять назви класів, які представляють даний тип. Наприклад, файл `Int32.cshtml`, що використовується для візуалізації значень `int`, має наступну розмітку:

```
@model Int32
<b>@Model</b>
```

Директива `@model Int32` вказує, що в якості моделі буде використовуватися тип `int` або `Int32`. А сама інформація моделі робиться жирним шрифтом за допомогою тега ``. Файл `String.cshtml` має наступний вміст:

```
@model String
<b>"@Model"</b>
```

Тут теж саме, тільки назва вказується в лапках. Загалом можемо визначати різні теги або структуру тегів, можна, наприклад, задати класи (` "@Model" `) і т.д. Файл `Decimal.cshtml`:

```
@model decimal
@{
    IFormatProvider formatProvider =
        new System.Globalization.CultureInfo("en-EN");
    <span class="currency">@Model.ToString("C", formatProvider)</span>
}
```

Тут вже визначається логіка форматування значення в якості грошової одиниці. Тепер ми використовуємо хелпер `DisplayFor()` для виведення властивостей моделі:

```

@model CustomDisplayEditorTemplates.Models.Book
@{
    ViewBag.Title = "Display";
}
<h2>Display</h2>
<div>
    <div>
        <p>
            Id: @Html.DisplayFor(model => model.Id)
        </p>
        <p>
            Назва: @Html.DisplayFor(model => model.Name)
        </p>
        <p>
            Ціна: @Html.DisplayFor(model => model.Price)
        </p>
    </div>
</div>

```

У папку EditorTemplates додамо один шаблон форматування - String.cshtml:

```

@model string

<input type="text" name="name" style="border-color:red; height:40px; background-color:#eee;"
value="@Model" />

```

Тоді представлення (View) для редагування даних повинне мати наступний вигляд:

```

@model CustomDisplayEditorTemplates.Models.Book

@{
    ViewBag.Title = "Edit";
}

<h2>Edit</h2>

@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()
    @Html.HiddenFor(model => model.Id)
    <p>Назва: @Html.EditorFor(model => model.Name)</p>
    <p>Ціна: @Html.EditorFor(model => model.Price)</p>
    <p><input type="submit" value="Зберегти" /></p>
}

```

При створенні подібних шаблонів на основі елементів введення слід враховувати їх обмеженість: в даному випадку елемент має атрибут name = "name", тому його

значення буде зіставлятися з властивістю Name. Якщо у нас одна модель в якій є тільки одна строкова властивість, яка називається Name, то подібний підхід ще спрацює. Але у нас можуть бути декілька властивостей з типом string, які можуть називатися по-різному. У цьому випадку ми можемо визначити шаблон виводу або редагування моделі цілком. Наприклад, додамо в папку DisplayTemplates наступний файл Book.cshtml:

```
@model CustomDisplayEditorTemplates.Models.Book

<table>
  <tr><td>Id:</td><td>@Model.Id</td></tr>
  <tr><td>Назва:</td><td>@Model.Name</td></tr>
  <tr><td>Ціна:</td><td>@Model.Price грн</td></tr>
</table>
```

Custom Display EditorTemplates - в даному випадку це назва нашого проекту, в якому визначено клас Book. Значення всіх властивостей моделі виводяться через таблицю. Тоді представлення для відображення моделі буде виглядати так:

```
@model CustomDisplayEditorTemplates.Models.Book
@{
    Layout = null;
}
@Html.DisplayForModel()
```

Також додамо в каталог Editor Templates шаблон для редагування Book.cshtml:

```
@model CustomDisplayEditorTemplates.Models.Book

@using (Html.BeginForm())
{
    @Html.HiddenFor(model => model.Id)
    <table>
        <tr><td>Назва: </td><td><input type="text" name="name" value="@Model.Name"
/></td></tr>
        <tr><td>Ціна: </td><td><input type="text" name="price" value="@Model.Price" /></td></tr>
        <tr><td><input type="submit" value="Зберегти" /></td><td></td></tr>
    </table>
}
```

Використаємо цей шаблон для створення форми редагування:

```
@model CustomDisplayEditorTemplates.Models.Book
@{
    Layout = null;
}
@Html.EditorForModel()
```

3.5.14 Фільтрація даних

Розглянемо, як фільтрувати дані за певними критеріями в ASP.NET MVC. Візьмемо один з раніше розглянутих прикладів з футболістами і командами, де у нас є наступні моделі:

```
public class Player
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }
    public string Position { get; set; }

    public int? TeamId { get; set; }
    public Team Team { get; set; }
}
public class Team
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Coach { get; set; }

    public ICollection<Player> Players { get; set; }
    public Team()
    {
        Players = new List<Player>();
    }
}
```

Припустимо, при виведенні списку футболістів нам треба також передбачити можливість для їх фільтрації за деякими критеріями, наприклад, по команді і по позиції на полі. Так як ми будемо виводити в одному представленні і список гравців, і критерії для вибору, які представляють списки команд і позицій, то додамо в проект спеціальну модель:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web.Mvc;

namespace NavigationProperty.Models
{
    public class PlayersListViewModel
    {
        public IEnumerable<Player> Players { get; set; }
        public SelectList Teams { get; set; }
        public SelectList Positions { get; set; }
    }
}
```

Три властивості - для списку гравців, команд і позицій. Для виведення даних моделі визначимо представлення:

```

@model NavigationProperty.Models.PlayersListViewModel

@{
    ViewBag.Title = "Каталог гравців";
}

<h2> Каталог гравців </h2>

<p>
    @Html.ActionLink("Додати гравця", "Create")
</p>

<form method="get">
    <div class="form-inline">
        <label class="control-label">Команда: </label>
        @Html.DropDownList("team", Model.Teams as SelectList,
            htmlAttributes: new { @class = "form-control" })

        <label class="control-label">Позиція гравця: </label>
        @Html.DropDownList("position", Model.Positions as SelectList,
            htmlAttributes: new { @class = "form-control" })

        <input type="submit" value="Фільтр" class="btn btn-default" />
    </div>
</form>

<table>
    <tr>
        <th>Ім'я гравця</th>
        <th>Вік</th>
        <th>Позиція на полі </th>
        <th>Команда</th>
        <th></th>
    </tr>

    @foreach (var item in Model.Players)
    {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Name)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Age)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Position)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Team.Name)
            </td>
        </tr>
    }

```

```

        </td>
        <td>
            @Html.ActionLink("Редагувати", "Edit", new { id = item.Id }) |
            @Html.ActionLink("Видалити", "Delete", new { id = item.Id })
        </td>
    </tr>
}
</table>

```

Перед таблицею зі списком гравців, визначена форма, яка за допомогою запитів GET посилає вибрані у випадваючих списках значення на сервер. Тепер на стороні контролера визначимо метод, який буде виводити список гравців і здійснювати фільтрацію:

```

public class HomeController : Controller
{
    SoccerContext db = new SoccerContext();
    public ActionResult Index(int? team, string position)
    {
        IQueryable<Player> players = db.Players.Include(p => p.Team);
        if (team != null && team != 0)
        {
            players = players.Where(p=>p.TeamId==team);
        }
        if (!String.IsNullOrEmpty(position) && !position.Equals("Все"))
        {
            players = players.Where(p => p.Position == position);
        }

        List<Team> teams = db.Teams.ToList();
        // встановлюємо початковий елемент, який дозволить вибрати всіх
        teams.Insert(0, new Team { Name = "Все", Id = 0 });

        PlayersListViewModel plvm = new PlayersListViewModel
        {
            Players=players.ToList(),
            Teams = new SelectList(teams, "Id", "Name"),
            Positions = new SelectList(new List<string>()
            {
                "Всі",
                "Нападаючий",
                "Напівзахисник",
                "Захисник",
                "Воротар"
            })
        };
        return View(plvm);
    }
}

```

При зверненні до методу Index незалежно від того, чи були передані параметри team і position, в представлення буде виводитися список гравців. Якщо певний параметр був переданий, то до об'єкта IQueryable, що представляє вибірку, додається вираз Where:

```
players = players.Where (p => p.TeamId == team) ;
```

Для формування списку команд у представленні, отримуємо його з БД:

```
List<Team> teams = db.Teams.ToList();
```

Для можливості вибору гравців з будь-якої команди, додаємо в цей список новий пункт:

```
teams.Insert(0, new Team { Name = "Все", Id = 0 });
```

Всі отримані і створені списки використовуються для формування об'єкта Players List ViewModel, який передається в представлення.

3.5.15 Domain Model і View Model

Моделі, які використовуються в програмі, бувають різними. Раніше ми розглянули вивід і одночасно фільтрацію об'єктів Player:

```
public class Player
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }
    public string Position { get; set; }

    public int? TeamId { get; set; }
    public Team Team { get; set; }
}
```

Модель Player в даному випадку представляє модель домену, її об'єкти зберігаються в базі даних і щоб їх отримати, ми звертаємося через контекст даних до БД. Але може статися, що тільки функціональності цієї моделі виявиться недостатньо. Для передачі даних в представлення нам доведеться створити додаткову модель:

```
public class PlayersListViewModel
{
    public IEnumerable<Player> Players { get; set; }
    public SelectList Teams { get; set; }
    public SelectList Positions { get; set; }
}
```

Це модель подання або view model, яка призначена конкретно для виведення даних в деяке представлення. Важливо розуміти відмінність domain model від view model.

Інша ситуація, що часто зустрічається - редагування. Нам треба редагувати гравця, але треба змінити не всі його властивості, а тільки частину. В цьому випадку створюється додаткова модель представлення, яка містить ті властивості, які треба змінити. наприклад:

```
public class PlayerViewModel
{
    public int Id { get; set; }
    public int Age { get; set; }
    public string Position { get; set; }
}
```

І представлення для редагування використовує не модель Player, а модель PlayerViewModel. Тоді обробка введених даних могла б виглядати наступним чином:

```
[HttpPost]
public ActionResult Edit(PlayerViewModel playerModel)
{
    Player player = db.Players.Find(playerModel.Id);
    if (player == null)
        return HttpNotFound();
    player.Position = playerModel.Position;
    player.Age = playerModel.Age;
    db.SaveChanges();
    return RedirectToAction("Index");
}
```

У деяких випадках модель домену може збігатися з моделлю представлення. Наприклад, при створенні нового гравця ми можемо визначити наступне представлення:

```
@model NavigationProperty.Models.Player
```

```
@{
    ViewBag.Title = "Додавання гравця";
}
```

```
<h2> Додавання нового гравця </h2>
```

```
@using (Html.BeginForm())
{
    <fieldset>
        <legend>Футболіст</legend>

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

```

<p>
    Ім'я гравця <br />
    @Html.EditorFor(model => model.Name)
</p>

<p>
    Вік <br />
    @Html.EditorFor(model => model.Age)
</p>

<p>
    Позиція на полі <br />
    @Html.EditorFor(model => model.Position)
</p>
<p>
    Команда <br />
    @Html.DropDownListFor(model => model.TeamId, ViewBag.Teams as SelectList)
</p>

<p>
    <input type="submit" value=" Додавання гравця" />
</p>
</fieldset>
}

```

Представлення використовує модель Player, метод контролера також може отримувати цю модель і відразу зберігати її в базу даних. Тобто тут `domain model = view model`. Для маленьких додатків це цілком допустимо. Але в реальних програмах, як правило, такі ситуації зустрічаються не часто і то в основному в дуже невеликих. Для великих програм які постійно розширюються, подібна асоціація моделей може вилитися в проблеми з підтримкою і розвитком програми. Тому рекомендується для кожного окремого представлення створювати свою ViewModel.

Контрольні питання.

1. Основні способи застосування моделей (Model)
2. Підключення до баз даних.
3. Способи редагування моделей.
4. Міграція баз даних. Способи виконання.
5. Способи створення блочних сторінок.
6. Способи фільтрації даних.

3.6 Маршрутизація

3.6.1 Визначення маршрутів

У попередніх розділах ми так чи інакше стикалися з маршрутизацією. Наприклад, при зверненні до деякої дії контролера ми набирали в адресному рядку браузера :

`http://localhost: 3456 / Home / Index,`

де Home - ім'я контролера без префікса Controller, а Index - ім'я методу дії цього контролера. Якщо метод Index брав який-небудь параметр, наприклад, типу `int`:

`public ActionResult Index(int Id),`

то ми могли звернутися до цього методу і передати значення в його параметр за допомогою наступного рядка: `http://localhost:3456/Home/Index/5`.

Однак ми не говорили про те, чому ми повинні прописувати маршрут саме так, і як ми власне можемо управляти маршрутами. Розглянемо механізм визначення маршрутів. У ASP.NET MVC5 всі визначення маршрутів знаходяться у файлі `RouteConfig.cs`, який розташовується в проекті в папці `App_Start`. Якщо ми на неї подивимося, то побачимо налаштування маршруту за замовчуванням:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace Mvc5RoutesApp
{
    public class RouteConfig
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

            routes.MapRoute(
                name: "Default",
                url: "{controller}/{action}/{id}",
                defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }
            );
        }
    }
}
```

Установкою маршрутів займається статичний метод `RegisterRoutes`. Однак на цьому визначення маршрутів не закінчується, так як нам ще треба викликати цей метод в додатку при запуску. Це проводиться у файлі `Global.asax` в методі `Application_Start`:

```
protected void Application_Start()
```



```

{
    AreaRegistration.RegisterAllAreas();
    FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
    RouteConfig.RegisterRoutes(RouteTable.Routes);
    BundleConfig.RegisterBundles(BundleTable.Bundles);
}

```

Виклик методу `RouteConfig.RegisterRoutes (RouteTable.Routes);` якраз і проводить реєстрацію маршрутів у додатку. Тепер розберемо визначення маршруту за замовчуванням у файлі `RouteConfig.cs`. Перший рядок: `routes.IgnoreRoute("{resource}.axd/{*pathInfo}");` відключає обробку запитів для деяких файлів, наприклад з розширенням `*.axd` (`WebResource.axd`). Далі йде визначення маршруту за замовчуванням. Метод `routes.MapRoute` виконує зіставлення маршруту запиту. У перевантажених версіях даного методу ми можемо задати додаткові параметри зіставлення.

Розберемо параметри методу. Спочатку за допомогою властивості `name` задається ім'я маршруту - `Default`. Другий параметр - `url` задає шаблон рядка запиту або шаблон `Url`, з яким буде зіставлятися даний маршрут. Шаблон `URL` включає в себе кілька сегментів, які укладені у фігурні дужки. В даному випадку сегмент представляє частину запиту, яка знаходиться між слешів. Кожен такий сегмент шаблону містить параметр. Ці параметри називаються параметрами `URL`. В даному випадку це параметри **controller**, **action** і **id**. Взагалі параметри можуть мати різні імена, що включають будь-які алфавітно-цифрові символи.

При отриманні запиту механізм маршрутизації розбирає рядок `URL` і поміщає значення маршруту в словник - в об'єкт `RouteValueDictionary`, доступний через контекст додатку `RequestContext`. Як ключі в ньому застосовуються імена параметрів `URL`, а відповідні сегменти `URL` виступають в якості значень. Наприклад, якщо є наступний `URL` запиту:

`http://localhost/Home/Index/5`, то в цьому випадку утворюються такі пари ключів і значень в словнику `RouteValueDictionary`:

Параметр	Значення
controller	Home
action	Index
id	5

Третій параметр методу `routes.MapRoute - defaults` визначає значення за замовчуванням для маршруту. Якщо в рядку запиту вказані не всі параметри, а сам запит, наприклад, сформовано за адресою `http://localhost/`, то система маршрутизації викличе метод `Index` контролера `Home`, як зазначено в параметрі `defaults`. Також, якщо ми не вкажемо метод контролера, наприклад, `http://localhost/Home/`, також буде викликаний метод `Index` контролера `Home`. Тому якщо нам потрібно, наприклад, щоб у нас за замовчуванням клієнт звертався не до методу `Index` контролера `HomeController`, а, наприклад, до методу `Show` контролера `BookController`, то ми можемо відповідно змінити значення даного параметра:

```
defaults: new { controller = "Book", action = "Show", id = UrlParameter.Optional }
```

Останній параметр оголошений як необов'язковий `id = UrlParameter.Optional`, тому, якщо він не вказаний в рядку запиту, він не буде враховуватися і передаватися в словник параметрів `RouteValueDictionary`. Наприклад, запит `http://localhost/Home/Create/3` викличе метод `Create` контролера `HomeController`, передавши в цей метод як параметр число 3. У той же час запит `http://localhost/Home/Create/` також викличе метод `Create` контролера `HomeController`, хоча останній параметр в ньому не вказаний.

Таким чином, налаштування за замовчуванням дозволяють нам не вказувати в рядку запиту повністю назву контролера і його методу. Але у випадку, якщо такі налаштування не задані, ми повинні визначати в рядку запиту контролер і його метод. Наприклад, змінимо установку маршрутів у файлі `RouteConfig.cs` наступним чином:

```
public class RouteConfig
{
    public static void RegisterRoutes(RouteCollection routes)
    {
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
        routes.MapRoute(
            name: "Default",
            url: "{controller}/{action}"
        );
    }
}
```

Тепер звернемося до нашої програми, набравши в рядку браузера тільки адресу сайту, наприклад, `http://mysyte.com/`. У цьому випадку ми отримаємо інформацію про помилку. Помилка буде полягати в тому, що тепер нам треба повністю набирати в рядку запиту адресу ресурсу. Тому наступну адресу `http://mysyte.com/Home/Index` буде нормально працювати (якщо, звичайно, у додатку визначений контролер `Home` з методом `Index` і відповідним йому поданням). Якщо ми перейдемо за адресою `http://localhost/Home/`, то знову отримаємо помилку, так як в рядку запиту зазначений тільки один сегмент. А в визначенні маршруту у нас зазначено два сегменти - `{controller}/{action}`. Якщо для параметрів не визначені значення за замовчуванням, то рядок запиту повинна мати таке ж число сегментів, для яких не визначені значення за замовчуванням.

У той же час якщо запит буде складатися з трьох сегментів, наприклад, `http://localhost/Home/Index/1`, то ми також отримаємо помилку, тому що число сегментів в запиті більше числа, визначеного в шаблоні URL даного маршруту.

3.6.2 Робота з маршрутами

Створення нових маршрутів

Для створення маршрутів можна використовувати метод `MapRoute`. наприклад:

```
public class RouteConfig
```

```

{
    public static void RegisterRoutes(RouteCollection routes)
    {
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

        routes.MapRoute(
            name: "newRoute",
            url: "{controller}/{action}"
        );
    }
}

```

Інакшим чином, можна спочатку створити об'єкт Route, а потім додати його до колекції маршрутів RouteCollection. Для цього, визначимо два маршрути:

```

public class RouteConfig
{
    public static void RegisterRoutes(RouteCollection routes)
    {
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

        routes.MapRoute(
            name: "Default",
            url: "{controller}/{action}/{id}"
        );

        Route newRoute = new Route("{controller}/{action}", new MvcRouteHandler());
        routes.Add(newRoute);
    }
}

```

Отже, тут визначені два маршрути. Перший - маршрут Default зіставляється із запитам, що мають три сегмента. Другий - newRoute зіставляється із запитам, що мають тільки два сегменти. Так, виклик `http://localhost:5555/Home/Index/1` відповідатиме першому маршруту, тому що в ньому визначені три сегменти. А виклик `http://localhost:5555/Home/Index` - другому маршруту. Виклик `http://localhost:5555/Home` не буде відповідати ні одному маршруту, так як у нас не визначено маршрут, який приймає тільки один сегмент в шаблоні URL. Правда, ситуація з цими двома запитамі більшою мірою є штучною, так як ми можемо поєднати їх, просто визначивши параметр `id` як необов'язковий:

```

routes.MapRoute (name: "Default", url: "{controller} / {action} / {id } ", defaults: new {id =
UrlParameter.Optional});

```

При передачі значень за замовчуванням для параметрів важливо враховувати позицію параметра. Програма маршрутизації використовує значення за

замовчуванням тільки в тому випадку, якщо всі наступні параметри також мають значення за замовчуванням. Так, якщо ми задамо наступний маршрут:

```
routes.MapRoute (name: "Default", url: "{controller} / {action} / {id}", defaults: new {action = "Index"});
```

то запит, містить два сегменти, наприклад, Home/2 цієї статті не буде зіставлятися з даним маршрутом. Тому нам треба в даному випадку вказати значення за замовчуванням також і для параметра id.

3.6.3 Зіставлення запитів з файлами на диску

Ми дивилися, як зіставляти маршрути з контролерами і їхніми методами. Однак ми можемо адресувати запити також і окремих файлів сайту, наприклад, статичним html-сторінок. Робота механізму маршрутизації така, що спочатку він дивиться, чи збігається запит з відповідним файлом, що зберігаються на сервері, і якщо такого файлу не знаходить, тоді він починає зіставляти запит з певними маршрутами. Тому, додамо в папку Content проекту нову html-сторінку і назовемо її, наприклад, Welcome.html:

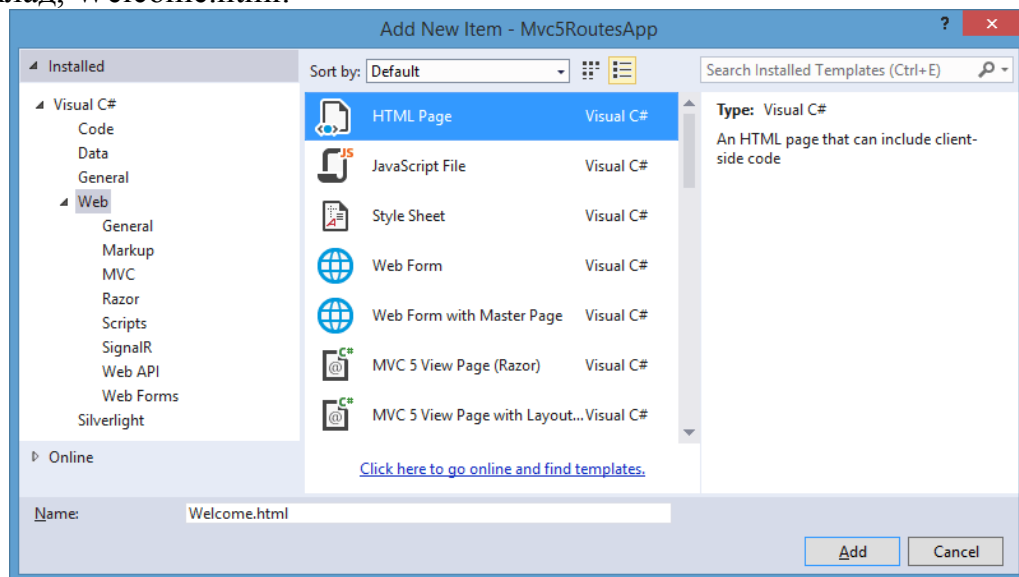


Рисунок 3.37- Створення нової html-сторінки

Для тестування створимо який-небудь контент на цій сторінці. І після її додавання в проект ми можемо до неї звертатися безпосередньо через запит Content / Welcome.html.

3.6.4 Використання префіксів в рядку запиту

Сегменти рядка запиту необов'язково повинні нести тільки значення для параметрів, визначених у маршруті. Можна також використовувати різні префікси в рядку запиту і відповідним чином налаштувати маршрут для обробки подібних запитів. Наприклад, ми хочемо, щоб запит містив префікс Ua: http://localhost:49326/Ua/Home/Index/1. Тоді оброблювальний запит маршрут може виглядати наступним чином:

```
routes.MapRoute(name: "Default", url: "Ua/{controller}/{action}/{id}", defaults: new { id =
UrlParameter.Optional }
```

Крім того, можна додавати префікси не в якості окремого сегмента, а до самого сегменту:

```
routes.MapRoute(name: "Default", url: "Ua{controller}/{action}/{id}", defaults: new { id =
UrlParameter.Optional }
```

У цьому випадку рядок запиту, відповідна даному маршруту, може виглядати так:

```
http://localhost:49326/UaHome/Index
```

3.6.5 Порядок визначення нових маршрутів

При додаванні нових маршрутів важливо враховувати їх порядок. Зазвичай більш специфічні маршрути поміщаються перед більш загальними. Наприклад, припустимо, ми визначили наступні маршрути:

```
public static void RegisterRoutes(RouteCollection routes)
{
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

    routes.MapRoute(name: "Default", url: "{controller}/{action}/{id}", defaults: new { id =
UrlParameter.Optional });

    routes.MapRoute( name: "Default2", url: "Ua{controller}/{action}/{id}", defaults: new { id =
UrlParameter.Optional });
}
```

При направленні додатком запиту типу `http://localhost:49326/UaHome/Index`, якщо у нас в додатку не визначений контролер `UaHomeController`, програма поверне помилку. Чому? Тому система маршрутизації намагається зіставити запит спочатку з першим маршрутом. Якщо запит не відповідає першому маршруту - тоді з другим і так далі за списком маршрутів, поки не знайде потрібний. В даному випадку вхідний запит відповідає і першому, і другому маршрутами. Проте перший маршрут буде шукати контролер по імені `UaHomeController` і, не знайшовши його, поверне помилку. Тому щоб подібна ситуація не відбулася, треба спочатку визначити маршрут `Default2`, який більш специфічний і не конфліктує з першим:

```
public static void RegisterRoutes(RouteCollection routes)
{
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
    routes.MapRoute( name: "Default2", url: "Ua{controller}/{action}/{id}", defaults: new { id =
UrlParameter.Optional });
```

```
routes.MapRoute(name: "Default", url: "{controller}/{action}/{id}", defaults: new { id =
UrlParameter.Optional });
}
```

Також ми можемо використовувати префікси в якості псевдонімів для контролера і його методу:

```
routes.MapRoute( name: "Default2", url: "Store/Buy", defaults: new { controller = "Home",
action="Index" } );
```

Тут для контролера HomeController використовується псевдонім Store, а для дії Index - псевдонім Buy. У підсумку даний маршрут буде зіставлятися з таким запитом, як Store/Buy. А система маршрутизації буде звертатися за таким запитом до методу Index контролера Home.

3.6.6 Отримання переданих параметрів

Щоб отримати передані значення для параметрів маршруту, можемо скористатися об'єктом RouteData. Наприклад, якщо у нас визначений стандартний маршрут:

```
routes.MapRoute(name:"Default",url:"{controller}/{action}/{id}", defaults: new {id =
UrlParameter.Optional});
```

ми можемо отримати значення параметра controller наступним чином:

```
public string Index()
{
    string controller = RouteData.Values["controller"].ToString();
    return controller;
}
```

Значення, передані в якості додаткових значень, наприклад, для параметра id, можна витягати з самого параметра методу:

```
public ActionResult Index(int id)
{
    ViewBag.OldId = id;
    //або так
    // ViewBag.OldId = RouteData.Values["id"];
    return View();
}
```

3.6.7 Передача довільної кількості параметрів в запиті

Раніше ми обмежувалися тільки трьома сегментами. Але якщо у нас метод приймає два і більше параметрів:

```
public ActionResult Index(int id=1, string name="")
{
    ViewBag.Name = name;
    return View();
}
```

Тоді ми просто можемо додати в маршрут потрібну нам кількість параметрів:
`routes.MapRoute(name:"Default",url:"{controller}/{action}/{id}/{name}",defaults:
new{id=UrlParameter.Optional, name=UrlParameter.Optional});`
Тоді даний маршрут буде обробляти запит: `Home/Index/1/name`, що еквівалентно наступному рядку запиту: `Home/Index?id=1&name=name`

Крім того, ми можемо позначити будь-яку кількість сегментів у запиті, щоб не бути жорстко прив'язаним до числа сегментів за допомогою параметра `{*catchall}`:

```
routes.MapRoute(name:"Default",url:"{controller}/{action}/{id}/{*catchall}",
defaults:new{controller="Home",action="Index", id=UrlParameter.Optional});
```

Тепер ми можемо обробляти запити з будь-якою кількістю сегментів:

Запит	Параметри запиту
<code>mysyte.com</code>	<code>controller=Home</code> <code>action=Index</code>
<code>mysyte.com/Book</code>	<code>controller=Book</code> <code>action=Index</code>
<code>mysyte.com/Book/Show</code>	<code>controller=Book</code> <code>action=Show</code>
<code>mysyte.com/Book/Show/2</code>	<code>controller=Book</code> <code>action=Show</code> <code>id=2</code>
<code>mysyte.com/Book/Show/2/Oldedition</code>	<code>controller=Book</code> <code>action=Show</code> <code>id=2</code> <code>catchall=Oldedition</code>
<code>mysyte.com/Book/Show/2/Oldedition/1960</code>	<code>controller=Book</code> <code>action=Show</code> <code>id=2</code> <code>catchall=Oldedition/1960</code>

Після отримання значення для параметра `catchall` ми самі повинні обробити його і отримати вже значення для окремих сегментів.

3.6.8 Створення обмежень для маршрутів

Іноді виникає необхідність, більш точно задати збіг рядка запиту для даного маршруту. Наприклад, нам треба задати, щоб ім'я контролера обов'язково

починалося з букви "H". Тоді нам треба визначити відповідне обмеження для імені за допомогою регулярних виразів:

```
routes.MapRoute(
    name: "Default",
    url: "{controller}/{action}/{id}",
    defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional },
    constraints: new { controller = "^H.*" }
);
```

За допомогою параметра `constraints` встановлюються обмеження маршруту. І якщо ми направимо додатком запит `Book/Index`, навіть якщо у нас є контролер `BookController` з методом `Index`, то додаток поверне помилку, так як ім'я контролера потрапляє під обмеження, а іншого маршруту, якому би відповідав запит `Book/Index`, у нас не задано.

Подібним чином ми можемо задати обмеження і для інших параметрів. Наприклад, нехай параметр `id` складається як мінімум з двох чисел:

```
routes.MapRoute(
    name: "Default",
    url: "{controller}/{action}/{id}/{*catchall}",
    defaults: new { controller = "Home", action = "Index" },
    constraints: new { controller = "^H.*", id = @"\d{2}" }
);
```

Тепер навіть запит `Home/Index/1` у нас не буде зіставлятися з маршрутом, оскільки параметр `id` складається з однієї цифри, а не з двох. Також ми можемо задати обмеження по типу методу. Наприклад, ми хочемо, щоб оброблялися тільки запити для методів з типом `GET`:

```
routes.MapRoute(
    name: "Default",
    url: "{controller}/{action}/{id}/{*catchall}",
    defaults: new { controller = "Home", action = "Index" },
    constraints: new { controller = "^H.*", id = @"\d{2}", httpMethod = new
    HttpMethodConstraint("GET") }
);
```

Тепер будуть оброблятися тільки ті запити, які представляють запити типу `GET`.

3.6.9 Створення власних обмежень

Щоб створити власне обмеження, потрібно реалізувати інтерфейс `IRouteConstraint` з єдиним методом `Match`, який має наступне визначення:

```
public interface IRouteConstraint
{
    bool Match(HttpContextBase httpContext, Route route, string parameterName,
        RouteValueDictionary values, RouteDirection routeDirection);
}
```



```
}
```

Обмеження маршруту застосовує інтерфейс `IRouteConstraint`. Це змушує бібліотеку маршрутизації викликати для обмеження маршруту метод `IRouteConstraint.Match`, щоб визначити, чи застосовується дане обмеження до цього запиту чи ні. Наприклад, створимо обмеження, яке не буде пропускати запити по деякому url. Отже, додамо в додаток наступний клас:

```
public class CustomConstraint : IRouteConstraint
{
    private string uri;
    public CustomConstraint(string uri)
    {
        this.uri = uri;
    }
    public bool Match(HttpContextBase httpContext, Route route, string parameterName,
        RouteValueDictionary values, RouteDirection routeDirection)
    {
        return !(uri == httpContext.Request.Url.AbsolutePath);
    }
}
```

Тут ми маємо на увазі, що якщо необхідний ресурс збігається зі значенням властивості `httpContext.Request.Url.AbsolutePath`, то запит не буде зіставлятися з маршрутом. Тоді визначення маршруту може виглядати наступним чином:

```
routes.MapRoute(
    name: "Default",
    url: "{controller}/{action}/{id}/{*catchall}",
    defaults: new { controller="Home", action="Index" },
    constraints: new { id = @"\d{2}", myConstraint = new
CustomConstraint("/Home/Index/12") }
);
```

Тепер запит `/Home/Index/12` не буде оброблятися, навіть якщо він задовольняє всім іншим умовам і обмеженням.

3.6.10 Ігнорування запитів

За замовчуванням в методі `RegisterRoutes` класу `RouteConfig` визначена такий рядок `routes.IgnoreRoute("{resource}.axd/{*pathInfo}");` За допомогою даного виразу ми забороняємо доступ до певного ресурсу, що знаходиться на сервері. Так, ми можемо переписати попередній приклад, де використовували обмеження для маршруту у вигляді заборони шляху `/Home/Index/12`, наступним чином:

```
public static void RegisterRoutes(RouteCollection routes)
{
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
    routes.IgnoreRoute("Home/Index/12");
}
```

```

routes.MapRoute(
    name: "Default",
    url: "{controller}/{action}/{id}/{*catchall}",
    defaults: new { controller = "Home", action = "Index", id=UrlParameter.Optional }
);
}

```

Тепер запит Home/Index/12 не буде оброблятися і зіставлятися з певним ресурсом на сервері.

3.6.11 Генерація вихідних адрес URL

Ще однією стороною механізму маршрутизації є генерація вихідних адрес у програмах, наприклад, при виведенні їх у представленнях. Найбільш простий спосіб виведення адреси є використання анкера - елемента a:

```
<a href="Home/Index/3"/>
```

Але також ми можемо використовувати спеціальні хелпери рендеринга - `Html.ActionLink` і `Html.RouteLink`

Html.ActionLink

Хелпер `ActionLink` створює гіперлінк на дію контролера. Якщо ми створюємо посилання на дію, визначене в тому ж контролері, то можемо просто вказати ім'я дії:

```
@Html.ActionLink("Натиснути тут...", "Show")
```

Що створює нам наступну розмітку:

```
<a href="/Home/Show">Натиснути тут...</a>
```

Коли треба вказати лінк на дію з іншого контролера, то в хелперів `ActionLink` в якості третього аргументу ім'я контролера. Наприклад, посилання на дію `List` контролера `Book` буде створюватися так:

```
@Html.ActionLink("Список книг", "List", "Book")
```

Крім того, якщо у нас в деякому методі `Index` контролера `Book` визначено декілька параметрів:

```

public class BookController : Controller
{
    public string Index(string author="Шевченко", int id=1)
    {
        return author + " " + id.ToString();
    }
}

```

То перевантажена версія хелпера `ActionLink` дозволяє передати параметр об'єкта (зазвичай анонімний тип) для параметра `routeValues`. Середовище виконання приймає властивості об'єкта і використовує їх для створення значень маршрутизації (імена властивостей стають іменами параметрів маршруту, а значення властивостей представляють значення параметра маршруту). Створимо посилання для вищеописаної дії контролера:

```
@Html.ActionLink("Всі книги", "Index", "Book", new { id=10}, null)
//або
@Html.ActionLink("Франко", "Index", "Book", new { author="Франко", id=5}, null)
```

Останній параметр в даному хелпері є параметром `htmlAttributes`. Ми можемо використовувати цей параметр для установки значення атрибуту елемента HTML. В даному випадку передається значення `null` (тобто ніяких атрибутів не встановлюється).

Тепер спробуємо передати атрибути, наприклад, встановити атрибути `id` і `class`:

```
@Html.ActionLink("Всі книги","Index","Book",new{author="Франко",id = 10}, new{
id="Franko",@class="link"})
```

Згенерована html-розмітка буде виглядати наступним чином:

```
<a class="link"
href="/Book/Index/10?author=%D0%A2%D0%BE%D0%BB%D1%81%D1%82%D0%BE%D0%B9"
id="Franko">Всі книги</a>
```

Зверніть увагу на знак `@` перед словом `class`: оскільки слово `"class"` є зарезервованим словом в C#, то для правильного рендеринга нам треба перед ним вказати знак `@`.

Html.RouteLink

Хелпер `RouteLink` використовує схожий шаблон, що і `ActionLink`: він приймає ім'я маршруту, але не вимагає аргументів для імені контролера та імені дії. Так, перший приклад з `ActionLink` еквівалентний наступному коду:

```
@Html.RouteLink("Всі книги", new { controller = "Book", action = "Index", author = "Франко", id =
10 }, new { id = "Franko", @class = "link" })
```

Щоб використовувати маршрут, нам треба вказати ім'я визначеного нами маршруту і потім визначити при необхідності додаткові параметри. Наприклад, візьмемо стандартний маршрут `Default`:

```
routes.MapRoute(
    name: "Default",
    url: "{controller}/{action}/{id}",
    defaults: new { controller = "Home", action = "Index", id=RouteParameter.Optional } }
```

);

Тоді створити посилання ми можемо, наприклад, так:

```
@Html.RouteLink("Всі книги", "Default", new { action = "Show" })
```

URL-хелпери

URL-хелпери схожі на хелпери `ActionLink` і `RouteLink` за тим винятком, що вони не повертають HTML, а створюють шляху URL і повертають їх у вигляді рядків. Є три типи URL-хелперів:

- `Action`
- `Content`
- `RouteUrl`

Хелпер `Action` схожий на `ActionLink` за тим винятком, що не повертає тег якоря. Наприклад, наступний код відображає адресу URL, але не саме посилання:

```
@Url.Action("Index", "Book", new { author = "Шевченко", id = 10 }, null)
```

Хелпер `RouteUrl` використовує той же шаблон, що і `Action`, але так само як і `RouteLink`, приймає ім'я маршруту і аргументи для параметрів маршруту:

```
@Url.RouteUrl(new { controller = "Book", action = "Index", author = "Шевченко", id = 10 })
```

Хелпер `Content` перетворює відносні шляхи в абсолютні. Приклад використання хелпера `Content` можна побачити в поданні `_Layout`:

```
<script src="@Url.Content("~/Scripts/jquery-1.5.1.min.js")" type="text/javascript"></script>
```

За допомогою тильди (~) хелпер `Content` генерує необхідний URL щодо розташування додатку. Без тильди URL міг би стати некоректним, якщо перенести додаток в інший віртуальний каталог.

3.6.12 Області

Незважаючи на те, що проект MVC за замовчуванням містить чітку структуру поділу на окремі функціональні частини - контролери, моделі, представлення, іноді для більш зручної роботи над програмою, особливо над великими програмами, програма ділиться на ряд областей (area).

Додамо в проект MVC область. Натиснемо правою кнопкою миші на проект і в меню виберемо `Add->Area`. У вікні додавання області дамо нової області назву, наприклад, `Store`. Після цього в структурі проекту відбудеться ряд змін: в проект буде додана нова папка `Areas`, в яку в свою чергу буде додана папка `Store` - безпосередньо для нашої нової області. У середині папки `Store` фактично опиниться міні-проект, в якому будуть папки для контролерів, моделей і представлень і клас реєстрації області.

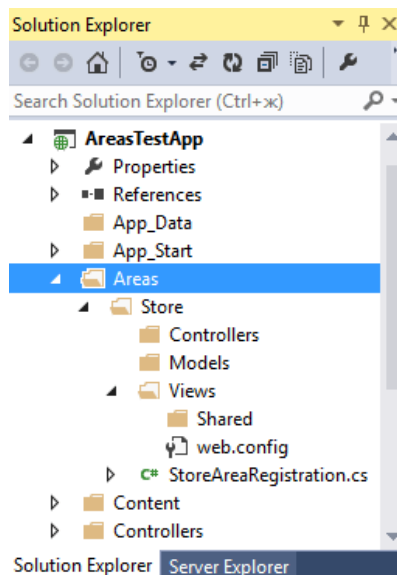


Рисунок 3.38 - Додавання області

Відкриємо файл реєстрації області Store AreaRegistration.cs:

```
using System.Web.Mvc;

namespace AreasTestApp.Areas.Store
{
    public class StoreAreaRegistration : AreaRegistration
    {
        public override string AreaName
        {
            get
            {
                return "Store";
            }
        }

        public override void RegisterArea(AreaRegistrationContext context)
        {
            context.MapRoute(
                "Store_default",
                "Store/{controller}/{action}/{id}",
                new { action = "Index", id = UrlParameter.Optional }
            );
        }
    }
}
```

Автоматично генероване визначення маршруту в цьому файлі показує, як вхідні запити будуть зіставлятися з контролерами і діями, визначеними в даній області Store. Однак щоб зіставлення запитів з областю відбувалося, треба також зареєструвати всі області у файлі Global.asax. Правда, вручну не доведеться цього робити, так як при додаванні в проект першої області Visual Studio вже автоматично це робить. Ви можете відкрити файл Global.asax і побачити зміни:

```

.....
using System.Web.Routing;

namespace AreasTestApp
{
    public class MvcApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            AreaRegistration.RegisterAllAreas();

            FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
            RouteConfig.RegisterRoutes(RouteTable.Routes);
            BundleConfig.RegisterBundles(BundleTable.Bundles);
        }
    }
}

```

Щоб протестувати нашу область, ми можемо також додати в папку Controllers, як і в основному проєкті, новий контролер і визначити в ньому дію. І також визначити для дій контролера представлення. Нехай, у нас в області Store визначений наступний контролер ShopController:

```

public class ShopController : Controller
{
    public string Index()
    {
        return "ShopController";
    }
}

```

Тоді при запуску програми ми можемо звернутися до методу Index контролера за адресою /Store/Shop/Index, вказавши спочатку ім'я області, а потім як зазвичай ім'я контролера і його методу. Але що буде, якщо ми захочемо додати в область контролер Home з методом Index, як і в основному додатку. Так як у нас визначений стандартний маршрут, який при запуску програми буде відсилати нас до методу Index контролера Home, то система маршрутизації опиниться в неоднозначному становищі: вона не буде знати, до якого саме контролера звертатися - до того, який визначений в основному додатку, або до того, який визначений в області. У підсумку ви побачите наступну картину:

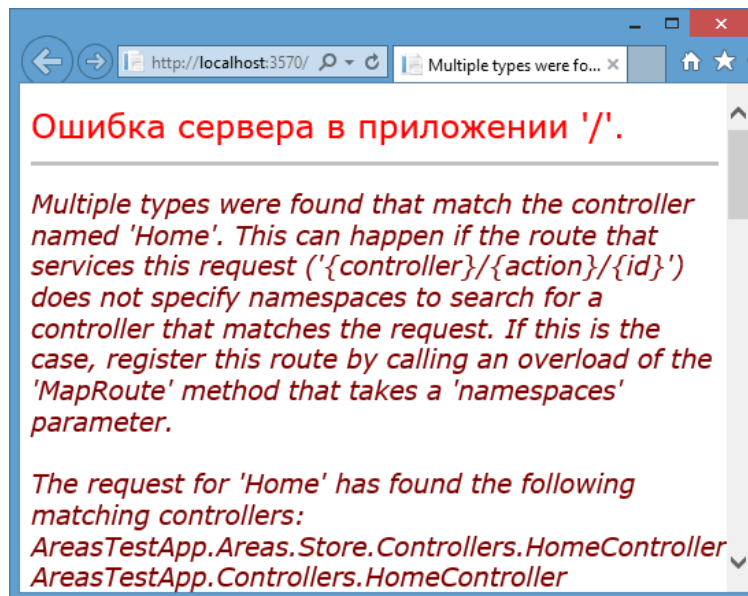


Рисунок 3.39 - Вивід помилка

Щоб уникнути подібної двоїстості, у файлі RouteConfig.cs проекту треба вказати простір імен контролера Home, який буде викликатися при запуску програми:

```
public class RouteConfig
{
    public static void RegisterRoutes(RouteCollection routes)
    {
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

        routes.MapRoute(
            name: "Default",
            url: "{controller}/{action}/{id}",
            defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional },
            namespaces: new[] { "AreasTestApp.Controllers" }
        );
    }
}
```

В даному випадку основне додаток за замовчуванням знаходиться в просторі імен AreasTestApp (так як проект називається AreasTestApp), а контролер Home - в просторі імен AreasTestApp.Controllers. Якби ми хотіли, щоб у нас під час запуску програми відпрацьовував метод Index, визначений у контролері Home в області Store, то ми могли б вказати відповідний простір імен, яке в моєму випадку - AreasTestApp.Areas.Store.Controllers

3.6.13 Генерація посилань в областях

При генерації посилань в представленнях окремих областей використовуються всі ті хелпери, наприклад, Html.ActionLink, проте тут є і деякі особливості.

Щоб згенерувати посилання на будь-яку дію контролера, які знаходяться всередині області, то ми вказуємо дію і контролер (якщо дія знаходиться в іншому контролері):

```
@Html.ActionLink("Всі книги","Index","Book",new{id=10},null)
```

У підсумку буде згенероване посилання: `Всі книги`
Якщо ж необхідна дія і контролер знаходяться в іншій області, то ми вказуємо область в параметрі хелпера:

```
@Html.ActionLink("Всі книги", "List", new { area = "Library", controller="Book" })
```

Згенероване посилання буде виглядати так: `Всі книги`
Якщо ж метод і контролер знаходяться в основному додатку, то для параметра area визначаємо порожній рядок:

```
@Html.ActionLink("Всі книги", "Index", new {area="",controller="Home" })
```

3.6.14 Створення власного обробника маршрутів

Перед тим як приступити до створення власного обробника маршрутів, розглянемо, що в цілому являє собою процес маршрутизації. Процес маршрутизації складається з наступних етапів:

1. Модуль `UrlRoutingModule` намагається зіставити поточний запит з маршрутами в таблиці `RouteTable`.
2. Якщо зіставлення завершилося вдало, то модуль маршрутизації вибирає обробник маршрутів сопоставленного маршруту - об'єкт `IRouteHandler`.
3. В об'єкта `IRouteHandler` викликається метод `GetHandler`, який повертає об'єкт `IHttpHandler`, який використовується для обробки запиту.
4. У обробника `IHttpHandler` викликається метод `ProcessRequest` для обробки запитів.

За замовчуванням обробник запитів або об'єкт `IRouteHandler` представляє екземпляр класу `MvcRouteHandler`, який повертає об'єкт `MvcHandler`, що застосовує інтерфейс `IHttpHandler`. Цей об'єкт `MvcHandler` відповідає за ініціалізацію контролера, який потім вже викликає одне з своїх дій. Однак ми можемо перевизначити таку поведінку, вказавши при визначенні маршруту свій обробник маршрутів. Отже, спочатку створимо сам обробник. Для обробника створимо деяку спеціальну папку в проекті і додамо в неї новий клас. Назвемо його `MyRouteHandler`:

```
using System.Web;
using System.Web.Routing;

namespace Routing.RouteHandlers
{
    public class MyRouteHandler : IRouteHandler
    {
        public IHttpHandler GetHttpHandler(RequestContext requestContext)
        {
            return new MyHttpHandler();
        }
    }
}
```



```

    }
}

public class MyHttpHandler : IHttpHandler
{
    public bool IsReusable
    {
        get { return false; }
    }

    public void ProcessRequest(HttpContext context)
    {
        context.Response.Write("Hello world!");
    }
}
}

```

Як було вже сказано про етапи маршрутизації, нам потрібно власне два класи: клас, який реалізує інтерфейс `IHttpHandler` і буде обробляти запит та клас, який реалізує інтерфейс `IRouteHandler`, який зіставляється з маршрутом, і викликає перший клас.

Тепер у класі `RouteConfig` пропишемо маршрут, який буде оброблятися нашим обробником, не забуваючи при цьому імпортувати простір імен, в якому оголошено наш обробник:

```

public class RouteConfig
{
    public static void RegisterRoutes(RouteCollection routes)
    {
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
        Route newRoute = new Route("{controller}/{action}", new MyRouteHandler());
        routes.Add(newRoute);
    }
}

```

Тепер ми можемо адресувати нашому додатку відповідний запит, наприклад, `Home/Index` і браузер виведе нам заданий рядок з текстом.

3.6.15 Атрибути маршрутизації.

Одним з нововведень MVC5 стала така функціональність як атрибути маршрутизації (attribute routing). Дані атрибути дозволяють зіставити певний маршрут з методом контролера. Наприклад, у нас є такий метод в контролері `HomeController`:

```

public string Test(int id, string name)
{
    return id.ToString() + ". " + name;
}

```

Йому міг би відповідати наступний маршрут, визначений у файлі `RouteConfig.cs`:

```

public class RouteConfig
{

```

```

public static void RegisterRoutes(RouteCollection routes)
{
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

    routes.MapRoute(
        name: "Default2",
        url: "{id}/{name}",
        defaults: new { controller = "Home", action = "Test" },
        constraints: new { id = @"\d+" }
    );

    routes.MapRoute(
        name: "Default",
        url: "{controller}/{action}/{id}",
        defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }
    );
}
}

```

Параметр `constraints: new { id = @"\d+" }` обмежує перший сегмент рядка запиту числами, таким чином не буде конфлікту між двома маршрутами. Тобто запит виду `http://localhost:6392/2/dnipro` буде зіставлятися з першим маршрутом, а якщо замість числа 2 йтиме рядок - то з другим.

Але атрибути маршрутизації дозволяють не визначати додатковий маршрут, а вказати зіставлення прямо в коді контролера:

```

[Route("{id:int}/{name}")]
public string Test(int id, string name)
{
    return id.ToString() + ". " + name;
}

```

Крім того, нам треба змінити код у файлі `RouteConfig.cs` наступним чином:

```

public class RouteConfig
{
    public static void RegisterRoutes(RouteCollection routes)
    {
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

        routes.MapMvcAttributeRoutes();

        routes.MapRoute(
            name: "Default",
            url: "{controller}/{action}/{id}",
            defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }
        );
    }
}

```

Тут замість визначення маршруту ми використовуємо метод `routes.MapMvcAttributeRoutes()`; який підключає в систему маршрутизації додатку функціональність атрибутів маршрутизації. Ми також можемо явно вказати ім'я контролера або методу або параметра в атрибуті: `[Route ("Home / {id: int} / {name}")]`.

Даний маршрут буде відповідати запиту `Home/4/somename`

3.6.16 Обмеження маршрутів

У визначенні сегмента `id` ми використовували обмеження, щоб явно вказати, що цей сегмент повинен представляти ціле число: `id: int`. Крім `int` ми можемо поставити ще ряд обмежень по типу:

- **alpha**: відповідає тільки алфавітним символам латинського алфавіту. Наприклад, `{id: alpha}`
- **bool**: відповідає логічеському значенню. Наприклад, `{id: bool}`
- **datetime**: відповідає значенню `DateTime`. Наприклад, `{id: datetime}`
- **decimal**: відповідає значенню `decimal`. Наприклад, `{id: decimal}`
- **double**: відповідає значенню `double`. Наприклад, `{id: double}`
- **float**: відповідає значенню `float`. Наприклад, `{id: float}`
- **length**: відповідає рядку певної довжини, або її довжина повинна бути в певному діапазоні. Наприклад, `{id: length (5)}` або `{id: length (5, 15)}`
- **long**: відповідає значенню `long`. Наприклад, `{id: long}`
- **max**: відповідає значенню `int`, яка не більше значення `max`. Наприклад, `{id: max (99)}`. Аналогічним чином діє обмеження `min`, тільки воно вказує на мінімально допустиме значення сегмента.
- **maxlength**: відповідає рядку, довжина якої не більше певного значення. Наприклад, `{id: maxlength (20)}`. Аналогічним чином працює обмеження `minlength`, вказуючи на мінімально допустиму довжину рядка
- **range**: вказує на діапазон, в межах якого повинно знаходитися значення сегмента. Наприклад, `{id: range (5, 20)}`
- **regex**: відповідає регулярному виразу. Наприклад, `{id: regex (^ \ d {3} - \ d {3} - \ d {4} $)}`

3.6.17 Значення за замовчуванням

Як і при визначенні маршруту, ми можемо задати значення для параметрів за замовчуванням:

```
[Route("{id:int}/{name=dnipro}")]
public string Test(int id, string name)
{
    return id.ToString() + ". " + name;
}
```

Так, якщо рядок запиту не буде містити останній параметр, то замість нього буде використовуватися рядок "dnipro".

3.6.18 Використання префіксів

Вище був наведений приклад атрибуту маршрутизації з назвою контролера на початку: [Route("Home/{id: int}/{name}")]. Але якщо є кілька подібних дій, звернення до яких повинно починатися з "Home", то зручно використовувати префікси:

```
[RoutePrefix("home")]
public class HomeController : Controller
{
    [Route("{id:int}/{name}")]
    public string Test(int id, string name)
    {
        return id.ToString() + ". " + name;
    }
    [Route("{id:int}")]
    public string Sead(int id)
    {
        return id.ToString();
    }
    [Route("~/lol/twit/{id:int}")]
    public string Twit(int id)
    {
        return id.ToString();
    }
}
```

Тепер запит до обох методам повинен починатися з Home: "Home/5/fds" або "Home/5". При цьому префікс не обов'язково повинен збігатися з ім'ям контролера, а може мати будь-яке значення. Останній маршрут усуває дію префікса за допомогою знака тильди (~) на початку маршруту. Для того, щоб до цього методу звернутися, необхідно буде використовувати запит <http://localhost:6392/lol/twit/2>.

3.6.19 Маршрутизація та вкладені ресурси

Нерідко в різних соцмережах або інших сервісах можна зустріти посилання такого типу: <https://twitter.com/SomeName/following> або, наприклад,

<https://plus.google.com/u/0/12344566556/posts>.

Ці два посилання об'єднує схожа схема маршрутизації - спочатку йде спільний шлях, потім ідентифікатор користувача (SomeName або 12344566556), і далі якась властивість, що відноситься до користувача - following (передплати в твіттері) або posts (всі пости в Google+). Якщо ми подивимося з погляду організації даних, то вийде, що в деякий клас користувача зберігає об'єкти постів чи підписок. Таким чином, ці об'єкти виявляються вкладеними в об'єкт

користувача. Подивимося, як ми зможемо створити систему маршрутів, щоб звертатися до цих об'єктів як до вкладених і в asp.net mvc.

Припустимо, у нас є такі класи, які пов'язані за зовнішнім ключем і об'єкти яких зберігаються в базі даних:

```
public class Book
{
    public int Id { get; set; }
    public string Name { get; set; }

    public int AuthorId { get; set; }
    public Author Author { get; set; }
}
public class Author
{
    public int Id { get; set; }
    public string Name { get; set; }
}
```

Кожен об'єкт Book зберігає посилання на пов'язаного учасника з таблиці авторів. Контекст даних міг би виглядати так:

```
public class BookContext : DbContext
{
    public DbSet<Book> Books { get; set; }
    public DbSet<Author> Authors { get; set; }
}
```

Тепер наше завдання полягає в тому, щоб передати користувачеві інформацію про будь-яку книгу і про її автора. Для цього створимо в контролері HomeController дві дії:

```
public ActionResult GetBook(int? id)
{
    if (id == null)
        return HttpNotFound();
    Book book = db.Books.Include(b => b.Author).FirstOrDefault(b=>b.Id == id);
    if (book == null)
        return HttpNotFound();
    return View(book);
}
public ActionResult GetAuthor(int? id)
{
    if (id == null)
        return HttpNotFound();
    Book book = db.Books.Include(b => b.Author).FirstOrDefault(b=>b.Id == id);
    if (book == null)
        return HttpNotFound();
    return View(book.Author);
}
```

Кожна дія отримує потрібну книгу по id і передає в строго типізоване представлення об'єкт книги або об'єкт автора книги. Тепер додамо потрібні маршрути. Змінимо код у файлі RouteConfig.cs наступним чином:

```
public class RouteConfig
{
    public static void RegisterRoutes(RouteCollection routes)
    {
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

        routes.MapRoute(
            name: "author",
            url: "book/{id}/author",
            defaults: new { controller = "Home", action = "GetAuthor" }
        );

        routes.MapRoute(
            name: "book",
            url: "book/{id}",
            defaults: new { controller = "Home", action = "GetBook" }
        );

        routes.MapRoute(
            name: "Default",
            url: "{controller}/{action}/{id}",
            defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }
        );
    }
}
```

Отже, тут ми додаємо два маршрути. Перший маршрут буде обробляти запит типу book/1/author і зіставляти його з методом GetAuthor (фактично представлятиме запит home/getauthor/1). А другий маршрут буде обробляти запит типу home/book/1 і зіставляти його з методом GetBook.

Тепер ми можемо запустити програми та отримати інформацію про першу книгу, набравши в браузері book/1/, а якщо ми наберемо book/1/author, то отримаємо інформацію про автора цієї книги. Також ми могли використовувати атрибути маршрутизації, щоб створити подібні маршрути:

```
[Route("book/{id}")]
public ActionResult GetBook(int? id)
{
    // *****
}

[Route("book/{id}/author")]
public ActionResult GetAuthor(int? id)
{
    // *****
}
```

А код у файлі RouteConfig.cs виглядав би наступним чином:

```
public class RouteConfig
{
    public static void RegisterRoutes(RouteCollection routes)
    {
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

        routes.MapMvcAttributeRoutes();

        routes.MapRoute(
            name: "Default",
            url: "{controller}/{action}/{id}",
            defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }
        );
    }
}
```

Контрольні питання.

1. Особливості визначення маршрутів.
2. Яким способом визначаються нові маршрути?
3. Способи передачі довільної кількості даних.
4. Способи обмеження маршрутів.

3.7 Метадані та валідація моделі

При роботі з моделлю в MVC великого значення відіграють метадані. Вони дозволяють вказати на додаткову інформацію про об'єкт, наприклад, про те, як відображати його властивості в представленні, або на те, як здійснювати валідацію вводу.

3.7.1 Анотації даних для відображення властивостей

Анотації даних являють собою атрибути, які ви можете знайти в просторі імен System.ComponentModel.DataAnnotations (хоча декілька атрибутів визначено в інших просторах).

Атрибут Display

Отже, припустимо у нас є деяка модель Book:

```
public class Book
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Author { get; set; }
    public int Year { get; set; }
}
```

І ми намагаємося отримати з бази даних один об'єкт цієї моделі:

```
public class HomeController : Controller
{
    BookContext db = new BookContext();
    public ActionResult Index()
    {
        var firstBook = db.Books.FirstOrDefault();
        return View(firstBook);
    }
}
```

Використовуючи відповідний хелпер, ми виводимо цю модель представлення:

```
@Html.DisplayForModel()
@model DataAnnotations.Models.Book
@{
    ViewBag.Title = "Книга";
}
@Html.DisplayForModel()
```

Як видно, заголовки у властивостей точно такі ж, як і назви властивостей, що не дуже зручно. Адже ми могли назвати властивості як завгодно. До того ж ми не хочемо, щоб назви були англійською мовою.

Звичайно, в цьому випадку ми могли б використовувати `@Html.LabelFor` для налаштування назви для кожної окремої властивості, але в цьому випадку у нас би збільшився в кілька разів обсяг коду, і ми не змогли б одним рядком вивести всю інформацію про модель. Тому даний підхід не оптимальний. Можна виправити модель, вказавши у неї параметри відображення:

```
using System.ComponentModel.DataAnnotations;
```

```
.....
```

```
public class Book
{
    public int Id { get; set; }
    [Display(Name = "Назва")]
    public string Name { get; set; }
    [Display(Name = "Автор")]
    public string Author { get; set; }
    [Display(Name = "Рік")]
    public int Year { get; set; }
}
```

Властивість `Name` атрибуту `Display` містить рядок, яка буде відображатися замість імені властивості. Більше не потрібно нічого змінювати.

Атрибут **Hidden Input**

У попередньому прикладі у нас залишилася одна проблема - це поле `Id`. Іноді, звичайно, може знадобитися висновок поля `Id`. Але, наприклад, якби ми

виводили модель в режимі редагування за допомогою хелпера `@Html.EditorForModel()`, то дане поле було б доступне для редагування. Щоб приховати це поле ми можемо застосувати атрибут `HiddenInput`:

```
using System.ComponentModel.DataAnnotations;
using System.Web.Mvc;
```

```
.....
public class Book
{
    [HiddenInput (DisplayValue=false)]
    public int Id { get; set; }
    [Display(Name = "Назва")]
    public string Name { get; set; }
    [Display(Name = "Автор")]
    public string Author { get; set; }
    [Display(Name = "Рік")]
    public int Year { get; set; }
}
```

Властивість `DisplayValue = false` вказує, що треба приховати дане поле.

При використанні хелперів редагування (`Html.EditorFor/Html.EditorForModel`) для даної властивості буде згенеровано приховане поле: `<input type = "hidden" id = "Id" name = "Id" value = "1" />`

Атрибут **ScaffoldColumn**

При редагуванні моделі атрибут `Hidden Input` повністю не закриває поля, так як ми можемо подивитися вихідний код сторінки і знайти відповідні поля. Щоб повністю приховати властивість від хелперів, використовується атрибут `ScaffoldColumn`:

```
[ScaffoldColumn(false)]
public int Id { get; set; }
```

Тепер хелпери редагування не побачать дану властивість і не створять для неї навіть приховане поле на сторінці.

Атрибут **DataType**

Атрибут `DataType` дозволяє надавати середовищі виконання інформацію про використання властивості. Наприклад, припустимо, у нас є властивість `Password`:

```
[DataType(DataType.Password)]
public string Password { get; set; }
```

Для властивості з атрибутом `DataType.Password` HTML-хелпери створюють елемент вводу, у якого атрибут `type` має значення `"password"`. Тоді в браузері при введенні даних не будуть видні символи, які вводяться, а замість них будуть

виводитися точки. Перерахування `DataType` може приймати кілька різних значень:

Значення	Опис
<code>Currency</code>	Відображає текст у вигляді валюти
<code>DateTime</code>	Відображає дату і час
<code>Date</code>	Відображає тільки дату, без часу
<code>Time</code>	Відображає тільки час
<code>Text</code>	Відображає однорядковий текст
<code>MultilineText</code>	Відображає багаторядковий текст (елемент <code>textarea</code>)
<code>Password</code>	Відображає символи з використанням маски
<code>Url</code>	Відображає рядок URL
<code>EmailAddress</code>	Відображає електронну адресу

Атрибут `UIHint`

Даний атрибут вказує, якою буде використовуватися шаблон відображення при створенні розмітки `html` для даної властивості. Шаблон керує, як властивість буде рендеритись на сторінці.

Є наступні вбудовані шаблони:

- `Boolean` - Хелпери редагування створюють флаг (`checkbox`) для булевих значень. Для значень типу `bool?` (`nullable`) створюється елемент `select` з параметрами `True`, `False` і `Not Set`. Хелпери відображення генерують ті ж елементи `html`, що і хелпери редагування, тільки з атрибутом `disabled`;
- `Collection` - Використовується відповідний шаблон для рендеринга кожного елемента колекції. Причому елементи можуть бути різних типів;
- `Decimal` - Хелпери редагування створюють однорядкове текстове поле - елемент `input`;
- `EmailAddress` - Хелпери редагування створюють однорядкове текстове поле. Хелпери відображення генерують елемент посилання, де атрибут `href` має значення `mailto: url`;
- `HiddenInput` - Створюється приховане поле - елемент `hidden input`;
- `Html` - Хелпери редагування створюють однорядкове текстове поле. Хелпери відображення просто показують текст;
- `MultilineText` - Хелпери редагування створюють багаторядкове текстове поле (елемент `textarea`);
- `Object` - Хелпери вивчають властивості об'єкта і вибирають найбільш підходящі для нього шаблони;
- `Password` - Хелпери редагування створюють текстове поле для введення символів з використанням маски. Хелпери відображення показують пароль як `*`, без використання маски;

- String - Хелпери редагування створюють однорядкове текстове поле;
- Url - Хелпери редагування створюють текстове поле. Хелпери відображення створюють елемент посилання для даного Url;

Наприклад, використовуємо останній шаблон Url:

```
public class Book
{
    [HiddenInput(DisplayValue = false)]
    public int Id { get; set; }

    [Display(Name = "Назва")]
    [UIHint("Url")]
    public string Name { get; set; }

    [Display(Name = "Автор")]
    public string Author { get; set; }

    [Display(Name = "Рік")]
    public int Year { get; set; }
}
```

3.7.2 Основи валідації

Отже, щоб подивитися, що з себе представляє валідація, створимо проект за шаблоном Basic Application.

Додамо в наш додаток нову модель Book, яку частково ми розробляли раніше:

```
public class Book
{
    [HiddenInput(DisplayValue = false)]
    public int Id { get; set; }

    [Required]
    [Display(Name = "Назва")]
    public string Name { get; set; }

    [Required]
    [Display(Name = "Автор")]
    public string Author { get; set; }

    [Required]
    [Display(Name = "Рік")]
    public int Year { get; set; }
}
```

Зверніть увагу, що в порівнянні з попередньою версією даної моделі, ми додали атрибут Required. Тепер створимо контролер, який буде управляти об'єктами даної моделі. Зробимо контролер (нехай контролер називається

допустимо BookController) типізований: як шаблон виберемо MVC 5 Controller with views, using Entity Framework, а в якості класу моделі вкажемо нашу модель Book, а в якості контексту даних = BookContext:

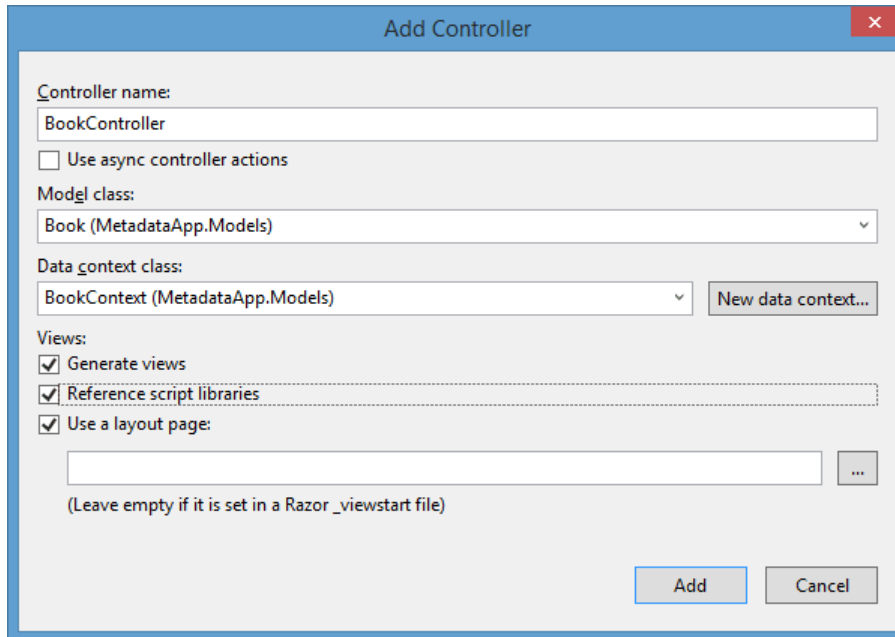


Рисунок 3.40 - Додавання контролера

У результаті у нас буде за замовчуванням створений набір представлень з управління об'єктами моделі. Запустимо додаток і звернемося до дії Create нашого контролера (запит Book/Create), яка повинна додавати новий запис у БД. Не заповнюючи поля, ми можемо відразу натиснути на кнопку відправки форми, при цьому на екрані з'являться червоні повідомлення про помилки, оскільки жодне поле не було заповнене.

У даному випадку ми задіяний механізм валідації, який використовується в mvc за замовчуванням. Які ж основні моменти валідації в даному випадку:

1. Використання атрибутів валідації при оголошенні моделі. Ми вказали для кожної властивості атрибут Required, завдяки чому фреймворк знає, що дана властивість обов'язково повинно містити деяке значення.
2. Використання хелперів валідації. При кожному властивості ми використовуємо хелпер валідації Html.ValidationMessageFor, Завдяки чому і відображається повідомлення про помилку:

```
<div class="editor-label">
    @Html.LabelFor(model => model.Name)
</div>
<div class="editor-field">
    @Html.EditorFor(model => model.Name)
    @Html.ValidationMessageFor(model => model.Name)
</div>
```

3. Щоб налаштувати стиль відображення на стороні клієнта, у файлі Site.css за замовчуванням вказано стиль для відповідних класів:

```

/* styles for validation helpers */
.field-validation-error {
    color: #b94a48;
}
.field-validation-valid {
    display: none;
}
input.input-validation-error {
    border: 1px solid #b94a48;
}
input[type="checkbox"].input-validation-error {
    border: 0 none;
}
.validation-summary-errors {
    color: #b94a48;
}
.validation-summary-valid {
    display: none;
}

```

4. Валідація на стороні сервера. Проте якщо у вас відключений javascript в браузері, то ви не побачите повідомлень з помилками, а форма благополучно відправиться на сервер. Але у фреймворку передбачена також валідація на стороні сервера:

```

public ActionResult Create([Bind(Include="Id,Name,Author,Year")] Book book)
{
    if (ModelState.IsValid)
    {
        db.Books.Add(book);
        db.SaveChanges();
        return RedirectToAction("Index");
    }

    return View(book);
}

```

За допомогою властивості `ModelState.IsValid` ми дізнаємося, проходить модель валідацію чи ні, і залежно від результату здійснюємо ті чи інші дії. Такий у загальному механізм валідації програми за замовчуванням.

3.7.3 Атрибути валідації

Атрибут `Required`

Застосування цього атрибуту до властивості моделі означає, що дана властивість має бути обов'язково встановлена. Щоб при валідації ми не отримували незрозумілих повідомлень про помилку, цей атрибут дозволяє настроїти текст повідомлення:

```

public class Book
{
    [HiddenInput(DisplayValue = false)]
    public int Id { get; set; }

    [Required(ErrorMessage = "Поле повинне бути встановлене")]
    [Display(Name = "Назва")]
    public string Name { get; set; }

    [Required]
    [Display(Name = "Автор")]
    public string Author { get; set; }

    [Required]
    [Display(Name = "Рік")]
    public int Year { get; set; }
}

```

А якщо ми явно не встановимо текст повідомлення, то при виведенні помилки буде відображатися стандартний текст повідомлення.

Атрибут **RegularExpression**

Використання даного атрибута припускає, що вводиться значення повинно відповідати зазначеному в цьому атрибуті регулярному виразу. Найбільш поширений приклад - це перевірка адреси електронної пошти на коректність. Наприклад, клас моделі містить властивість Email:

```

[RegularExpression(@"[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}", ErrorMessage =
"Некоректна адреса")]
public string Email { get; set; }

```

Якщо введені значення не буде відповідати регулярному виразу, то буде відображено повідомлення про помилку.

Атрибут **StringLength**

Щоб користувач не міг ввести дуже довгий текст, використовується атрибут **StringLength**. Особливо це актуально, якщо в базі даних встановлено обмеження на розмір файлів рядків. Першим параметром йде максимальна допустима довжина рядка. Іменовані параметри, зокрема **MinimumLength** і **ErrorMessage**, дозволяють задати додаткові опції відображення.

```

public class Book
{
    [HiddenInput(DisplayValue = false)]
    public int Id { get; set; }

    [Required(ErrorMessage = "Поле повинне бути заповнене")]

```

```

[StringLength(50, MinimumLength = 3, ErrorMessage = "Довжина рядка повинна бути від 3 до
50 символів")]
[Display(Name = "Назва")]
public string Name { get; set; }

[Required]
[Display(Name = "Автор")]
public string Author { get; set; }

[Required]
[Display(Name = "Рік")]
public int Year { get; set; }
}

```

Атрибут Range

Атрибут Range визначає мінімальні та максимальні обмеження для числових даних:

```

[Display(Name = "Рік")]
[Range(1700,2000,ErrorMessage="Недопустимий рік")]
public int Year { get; set; }

```

Атрибут Range може працювати як з цілочисельними значеннями, так і з числами з плаваючою крапкою. А ще одна перевантажена версія його конструктора приймає параметр Type і два рядки (які дозволяють створити діапазон віку).

```

[Range(typeof(decimal), "0.00", "49.99")]
public decimal Price { get; set; }

```

Атрибут Compare

Атрибут **Compare** гарантує, що дві властивості об'єкта моделі мають одне і те ж значення. Якщо, наприклад, треба, щоб користувач ввів пароль двічі:

```

[DataType(DataType.Password)]
public string Password { get; set; }

[Compare("Password",ErrorMessage="Паролі не співпадають")]
[DataType(DataType.Password)]
public string PasswordConfirm { get; set; }

```

Якщо користувач введе вдруге інший пароль, відмінний від першого, то він побачить помилку. Таким чином, використовуючи атрибути валідації моделі, ми можемо керувати валідацією і укладати нескладну логіку перевірки значень властивостей вже в атрибути властивостей, не вдаючись до коду.

Атрибут Remote

Атрибут Remote на відміну від інших класів атрибутів знаходиться в просторі імен System.Web.Mvc. Він дозволяє виконувати валідацію на стороні клієнта із зворотними викликами на сервер.

Наприклад, назва книги, представлені властивістю Name, не повинно бути одно певному значенню. При валідації на стороні клієнта важко гарантувати дотримання ряду перевірок, особливо якщо ми задіємо бази даних. А за допомогою атрибуту Remote ми можемо послати значення властивості Name на сервер, а там вже перевірити.

```
public class Book
{
    [Required]
    [Remote("CheckName", "Home", ErrorMessage = "Name is not valid.")]
    public string Name { get; set; }
}
```

В атрибуті можна встановити ім'я дії і ім'я контролера, які повинні викликатися кодом на стороні клієнта, а також ще ряд іменованих параметрів, зокрема, повідомлення про помилку за допомогою параметра ErrorMessage. Клієнтський код посилає введені користувачем значення для властивості Name автоматично, а перевантажений конструктор атрибуту дозволяє вказати додаткові поля, значення яких треба посилати на сервер.

```
[HttpGet]
public JsonResult CheckName(string name)
{
    var result = !(name=="Назва");
    return Json(result, JsonRequestBehavior.AllowGet);
}
```

Ця дія контролера приймає як параметр ім'я властивості, що підлягає валідації, і повертає true або false у формі об'єкта у форматі JSON. При цьому якщо повертається false, то ми побачимо повідомлення про помилку, задане параметром ErrorMessage.

3.7.4 Валідація моделі в контролері

Крім валідації на стороні клієнта, ми можемо здійснювати валідацію і всередині контролера. Робиться це за допомогою перевірки значення властивості ModelState.IsValid. Об'єкт ModelState зберігає всі значення, які користувач ввів для властивостей моделі, а також всі помилки, пов'язані з кожним властивістю і з моделлю в цілому. Якщо в об'єкті ModelState є які-небудь помилки, то властивість ModelState.IsValid поверне false:

```
[HttpPost]
public ActionResult Create(Book book)
{

```



```

if (ModelState.IsValid)
{
    ViewBag.Message = "Valid";
    db.Books.Add(book);
    db.SaveChanges();
    return RedirectToAction("Index");
}
ViewBag.Message = "Non Valid";
return View(book);
}

```

Припустимо, значення для властивості Name не вказано, і ми відсилаємо значення моделі на сервер. Оскільки до властивості застосовується атрибут Required, то після прив'язки моделі при валідації ми отримаємо такі значення:

```

ModelState.IsValid == false
ModelState.IsValidField("Name") == false
ModelState["Name"].Errors.Count > 0

```

Ми також можемо перевіряти коректність значень окремих властивостей моделі:

```

[HttpPost]
public ActionResult Create(Book book)
{
    if (string.IsNullOrEmpty(book.Name))
    {
        ModelState.AddModelError("Name", "Некоректна назва книги");
    }
    else if (book.Name.Length > 5)
    {
        ModelState.AddModelError("Name", "Недопустима довжина рядка");
    }
    if (ModelState.IsValid)
    {
        ViewBag.Message = "Валідація пройдена";
        db.Books.Add(book);
        db.SaveChanges();
        return RedirectToAction("Index");
    }
    ViewBag.Message = "Запрос не пройшов валідацію";
    return View(book);
}

```

У разі метод ModelState.AddModelError додає для властивості, зазначеного в якості першого параметра (в даному випадку Name) помилку "Неприпустима довжина рядка". При використанні хелперів згодом ми можемо вивести дане повідомлення про помилку:

```

<div class="editor-label">
    @Html.LabelFor(model => model.Name)

```

```

</div>
<div class="editor-field">
    @Html.EditorFor(model => model.Name)
    @Html.ValidationMessageFor(model => model.Name)
</div>

```

Крім помилок для конкретної властивості ми можемо вказувати помилки на рівні моделі. Вище ми бачили, що у нас одна властивість має некоректне значення. Помилки ж на рівні моделі пов'язують декілька властивостей. Наприклад:

```

if (book.Author=="Т.Шевченко" && book.Year > 1864)
{
    ModelState.AddModelError("", "Не вірний рік");
}

```

Обидва властивості можуть бути коректними в інших ситуаціях, але не разом. Зрозуміло, що Тарас Шевченко не міг написати книги після своєї смерті. Тому ми додаємо помилку на рівні моделі і в якості першого параметра опускаємо назва властивості і вказуємо просто лапки.

3.7.5 Відображення помилок валідації

Для відображення повідомлень про помилки при валідації моделі ми можемо використовувати хелпер `Html.ValidationSummary`:

```

@using (Html.BeginForm()) {
    @Html.ValidationSummary(true)

    <div class="editor-label">
        @Html.LabelFor(model => model.Name)
    </div>
    <div class="editor-field">
        @Html.EditorFor(model => model.Name)
        @Html.ValidationMessageFor(model => model.Name)
    </div>

    <div class="editor-label">
        @Html.LabelFor(model => model.Author)
    </div>
    <div class="editor-field">
        @Html.EditorFor(model => model.Author)
        @Html.ValidationMessageFor(model => model.Author)
    </div>

    <div class="editor-label">
        @Html.LabelFor(model => model.Year)
    </div>
    <div class="editor-field">
        @Html.EditorFor(model => model.Year)
    </div>
}

```

```

        @Html.ValidationMessageFor(model => model.Year)
    </div>

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

}

```

Даний хелпер має перевантажені версії, які допомагають налаштувати більш точне відображення повідомлень про помилки:

Перевантажена версія	Опис
<code>Html.ValidationSummary()</code>	Відображає загальний список помилок зверху
<code>Html.ValidationSummary(bool)</code>	Якщо параметр дорівнює <code>true</code> , то вгорі будуть відображатися тільки повідомлення про помилки рівня моделі, а специфічні помилки будуть відображатися поруч з полями введення. Якщо ж параметр дорівнює <code>false</code> , то вгорі відображаються всі помилки.
<code>Html.ValidationSummary(string)</code>	Дана перевантажена версія хелпера відображає перед списком помилок повідомлення, яке передається в параметр <code>string</code>
<code>Html.ValidationSummary(bool, string)</code>	Поєднує дві попередні перевантажені версії

Також ми можемо використовувати хелпер `Html.ValidationMessageFor` для виведення повідомлення про помилку для окремої властивості, причому поруч з полем для введення властивості, а не у зведенні помилок зверху:

```

@Html.ValidationSummary(true)
<div class="editor-label">
    @Html.LabelFor(model => model.Name)
</div>
<div class="editor-field">
    @Html.EditorFor(model => model.Name)
    @Html.ValidationMessageFor(model => model.Name)
</div>

<div class="editor-label">
    @Html.LabelFor(model => model.Author)
</div>
<div class="editor-field">
    @Html.EditorFor(model => model.Author)
    @Html.ValidationMessageFor(model => model.Author)

```

</div>

Стилізація помилок

Ще один важливий момент відображення помилок - це їх стилізація. Те, що ми бачимо помилки в червоному кольорі і межі полів введення також в червоному кольорі, що не жорстко встановлено, і ми все це можемо змінити. У файлі стилів Site.css ми можемо знайти секцію, яка якраз і відповідає за стилізацію:

```
/* styles for validation helpers */
.field-validation-error {
    color: #b94a48;
}
.field-validation-valid {
    display: none;
}
input.input-validation-error {
    border: 1px solid #b94a48;
}
input[type="checkbox"].input-validation-error {
    border: 0 none;
}
.validation-summary-errors {
    color: #b94a48;
}
.validation-summary-valid {
    display: none;
}
```

Це класи використовуються полями, генерованими хелперами валідації. Змінюючи дані стилі, ми можемо керувати відображенням помилок. Отже, змінимо два стилі - стиль класу `.field-validation-error` (він управляє відображенням помилок поряд з текстовими полями) і стиль класу `.input-validation-error`, що управляє відображенням поля введення в режимі помилки:

```
.field-validation-error {
    color: #6699CC;
    font-weight:bold;
    text-decoration:underline;
}

input.input-validation-error {
    border: 1px solid #006699;
    background-color: silver;
}
```

В такому вигляді текст помилки буде відображений не червоним, а синім кольором.

3.7.6 Створення власної логіки валідації

Хоча вбудовані атрибути валідації охоплюють значну кількість ситуацій, які можуть виникнути, але все ж їх буває недостатньо. Проте mvc фреймворк дозволяє створювати власні атрибути валідації і закладати в них свою логіку роботи.

Усі атрибути валідації утворені від базового класу `ValidationAttribute`, який знаходиться в просторі імен `System.ComponentModel.DataAnnotations`. Тому саме від цього класу ми будемо утворювати свій атрибут. Припустимо, що нам треба, щоб деяка книга була написана обмеженим колом авторів. Створимо відповідний атрибут, який буде це перевіряти:

```
using System.ComponentModel.DataAnnotations;

namespace Mvc4BasicApplication.Annotations
{
    public class MyAuthorsAttribute : ValidationAttribute
    {
        //масив для зберігання допустимих авторів
        private static string[] myAuthors;

        public MyAuthorsAttribute(string[] Authors)
        {
            myAuthors = Authors;
        }

        public override bool IsValid(object value)
        {
            if (value != null)
            {
                string strval = value.ToString();
                for (int i = 0; i < myAuthors.Length; i++)
                {
                    if (strval == myAuthors[i])
                        return true;
                }
            }
            return false;
        }
    }
}
```

Щоб застосувати логіку валідації, треба перевизначити метод `IsValid`, наданий базовим класом. Логіка атрибуту досить проста - ми отримуємо масив допустимих авторів. А при отриманні значення валідованої властивості перевіряємо, чи є воно в цьому масиві авторів, залежно від чого повертаємо `true` або `false`.

Використання атрибуту аналогічно використанню інших атрибутів валідації:

```

public class Book
{
    [HiddenInput(DisplayValue = false)]
    public int Id { get; set; }

    [Required (ErrorMessage="Поле повинне бути встановлене")]
    [Display(Name = "Название")]
    public string Name { get; set; }

    [Required]
    [MyAuthors(new string[] { "Т.Шевченко", "Л. Українка", "В. Стефаник", "М. Вовчок" },
    ErrorMessage="Недопустимий автор")]
    [Display(Name = "Автор")]
    public string Author { get; set; }

    [Required]
    [Display(Name = "Рік")]
    [Range(1700,2000,ErrorMessage="Недопустимий рік")]
    public int Year { get; set; }
}

```

У підсумку властивість буде невалідною, якщо був введений будь-який автор, який не входить до списку.

Атрибути валідації на рівні моделі

Атрибути валідації на рівні моделі застосовуються до перевірки комбінації властивостей. Наприклад, ми не хочемо, щоб користувач міг ввести певну книгу:

```

public class NotAllowedAttribute : ValidationAttribute
{
    public override bool IsValid(object value)
    {
        Book b = value as Book;
        if (b.Name=="Кобзар" && b.Author=="Т. Шевченко" && b.Year==1867)
        {
            return false;
        }
        return true;
    }
}

```

Тоді встановлюємо атрибут для всієї моделі:

```

[NotAllowedAttribute(ErrorMessage="Недопустима книга")]
public class Book
{
    [HiddenInput(DisplayValue = false)]
    public int Id { get; set; }
}

```

```

[Required(ErrorMessage="Поле повинне бути встановлене")]
[Display(Name = "Назва")]
public string Name { get; set; }
//.....
}

```

При спробі ввести зазначені дані для книги ми отримаємо помилку.

Автовалідація і IValidatableObject

Автовалідація являє собою процес, при якому модель запускає механізм валідації з себе самої, і сама інкапсулює всю логіку валідації. Для цього клас моделі повинен реалізувати інтерфейс IValidatableObject:

```

public class Book : IValidatableObject
{
    [HiddenInput(DisplayValue = false)]
    public int Id { get; set; }

    [Display(Name = "Назва")]
    public string Name { get; set; }

    [Display(Name = "Автор")]
    public string Author { get; set; }

    [Display(Name = "Рік")]
    public int? Year { get; set; }

    public IEnumerable<ValidationResult> Validate(ValidationContext validationContext)
    {
        List<ValidationResult> errors = new List<ValidationResult>();

        if (string.IsNullOrEmpty(this.Name))
        {
            errors.Add(new ValidationResult("Введіть назву книги"));
        }
        if (string.IsNullOrEmpty(this.Author))
        {
            errors.Add(new ValidationResult("Введіть автора книги"));
        }
        if (this.Year == null || this.Year<1700 || this.Year>2000)
        {
            errors.Add(new ValidationResult("Недопустимий рік"));
        }

        return errors;
    }
}

```

В даному випадку нам треба реалізувати метод `Validate` і повернути колекцію об'єктів `ValidationResult`, які і будуть містити всі помилки валідації.

Створення свого провайдера валідації

На додаток до вищерозглянутих методів перевизначення механізму валідації ми також можемо створити свій провайдер валідації. Для цього ми повинні створити клас похідний від класу `ModelValidatorProvider` і перевизначити його метод `GetValidators`.

```
public class MyValidationProvider : ModelValidatorProvider
{
    public override IEnumerable<ModelValidator> GetValidators(ModelMetadata metadata,
ControllerContext context)
    {
        if (metadata.ContainerType == typeof(Book))
        {
            return new ModelValidator[] { new BookPropertyValidator(metadata,context)};
        }

        if (metadata.ModelType == typeof(Book))
        {
            return new ModelValidator[] { new BookValidator(metadata, context) };
        }

        return Enumerable.Empty<ModelValidator>();
    }
}
```

Класи `Book PropertyValidator` і `BookValidator` ми розглянемо трохи пізніше, а поки подивимося загальну логіку класу. Метод `GetValidators` викликається для кожного властивості і окремо для всієї моделі. Тому ми використовуємо два валідатора - `BookPropertyValidator` (для властивостей моделі) і `BookValidator` (для моделі в цілому).

За допомогою переданого як параметр об'єкта `ModelMetadata` ми отримуємо деяку інформацію стосовно об'єктів валідації. Так, ми можемо отримати доступ до наступних властивостей даного об'єкта:

- `ContainerType`. Ця властивість повертає тип валідованої моделі, яка містить вказану властивість.
- `PropertyName`. Ця властивість повертає ім'я валідованої властивості
- `ModelType`. Ця властивість повертає тип об'єкта моделі

Тепер розглянемо клас `BookPropertyValidator`, який інкапсулює логіку валідації для окремих властивостей:

```
public class BookPropertyValidator : ModelValidator
{
    public BookPropertyValidator(ModelMetadata metadata, ControllerContext context)
        : base(metadata, context)
    {}
}
```



```

public override IEnumerable<ModelValidationResult> Validate(object container)
{
    Book b = container as Book;
    if (b != null)
    {
        switch (Metadata.PropertyName)
        {
            case "Name" :
                if (string.IsNullOrEmpty(b.Name))
                {
                    return new ModelValidationResult[]{
                        new ModelValidationResult { MemberName="Name", Message="Введіть назву
книги"}
                    };
                }
                break;
            case "Author":
                if (string.IsNullOrEmpty(b.Author))
                {
                    return new ModelValidationResult[]{
                        new ModelValidationResult { MemberName="Author", Message=" Введіть автора
книги"}
                    };
                }
                break;
            case "Year":
                if (b.Year>2000 || b.Year<1700)
                {
                    return new ModelValidationResult[]{
                        new ModelValidationResult { MemberName="Year", Message="Недопустимий
рік"}
                    };
                }
                break;
        }
    }
    return Enumerable.Empty<ModelValidationResult>();
}
}

```

У методі `Validate` ми визначаємо валідовані властивості і викликаємо відповідні дії з валідації об'єкта. Потім в об'єкт `ModelValidationResult` додаємо відомості щодо виниклої помилки: властивість `MemberName` вказує на ім'я валідованої властивості, а властивість `Message` - на повідомлення про помилку валідації.

Схожим чином виглядає валідатор для всієї моделі - `BookValidator`, тільки в об'єкт `ModelValidationResult` як значення властивості `MemberName` ми передаємо порожні лапки:

```

public class BookValidator : ModelValidator
{
    public BookValidator(ModelMetadata metadata, ControllerContext context)
        : base(metadata, context)
    {}

    public override IEnumerable<ModelValidationResult> Validate(object container)
    {
        Book b = (Book)Metadata.Model;

        List<ModelValidationResult> errors = new List<ModelValidationResult>();

        if (b.Name == "Кобзар" && b.Author == "Т. Шевченко" && b.Year == 1867)
        {
            errors.Add(new ModelValidationResult { MemberName = "", Message = "Недопустима
книга" });
        }
        return errors;
    }
}

```

Останнім кроком буде реєстрація свого провайдера валідації у файлі Global.asax:

```

public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        ModelValidatorProviders.Providers.Add(new MyValidationProvider());

        Database.SetInitializer(new BookDbInitializer());

        AreaRegistration.RegisterAllAreas();
        FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
        RouteConfig.RegisterRoutes(RouteTable.Routes);
        BundleConfig.RegisterBundles(BundleTable.Bundles);
    }
}

```

Тепер ми можемо звільнити нашу модель від всіх атрибутів, оскільки вся логіка валідації у нас вже закладена в провайдері:

```

public class Book
{
    [HiddenInput(DisplayValue = false)]
    public int Id { get; set; }

    [Display(Name = "Назва")]
    public string Name { get; set; }

    [Display(Name = "Автор")]

```

```

public string Author { get; set; }

[Display(Name = "Рік")]
public int Year { get; set; }
}

```

Контрольні питання.

1. Особливості валідації.
2. Особливості валідації моделі в контролері.
3. Способи створення власної логіки валідації.

3.8 Фільтри

Іноді виникає необхідність внести в додаток деяку логіку, яка повинна відпрацьовувати до виклику дій контролера. Зокрема, наприклад, користувач викликає певну дію деякого контролера, і нам необхідно перевіряти, чи авторизувався даний користувач, і потім вже виконувати певні дії - для цього застосовуються фільтри.

Фільтри реалізовані як атрибути, завдяки чому дозволяють зменшити об'єм коду в контролері. Дані атрибути можуть застосовуватися як до всього класу, так і до окремих його методів, властивостей і полів. Наприклад, якщо у нас є деякий контролер HomeController, і ми хочемо, щоб до нього мали доступ тільки авторизовані користувачі. У цьому випадку ми можемо використовувати наступний атрибут:

```

[Authorize]
public class HomeController : Controller
{
    //.....
}

```

Атрибут [Authorize] в даному випадку буде задіювати фільтр авторизації. Якби ми не застосували даний фільтр, то нам би довелося вручну писати логіку перевірки, авторизований користувач чи ні.

Застосування фільтрів

З вищенаведеного прикладу видно, що для застосування фільтра, треба атрибут поставити над класом контролера - в цьому випадку даний фільтр буде застосовуватися до всіх методів даного контролера, однак, ми можемо застосувати фільтр на рівні окремого методу:

```

public class HomeController : Controller
{
    [Authorize]
    public ActionResult Index()

```

```

{
    ViewBag.Message = "Hello world!";
    return View();
}
}

```

Базові типи фільтрів

У MVC 5 є декілька типів фільтрів:

Тип фільтрів	Реалізований інтерфейс	Стандартна реалізація	Пояснення
Фільтри аутентифікації	IAuthenticationFilter	Відсутня	Фільтр, що визначає, чи аутентифікований клієнт. Даний фільтр запускається до виконання будь-якого іншого фільтра або методу дій
Фільтри авторизації	IAuthorizationFilter	AuthorizationAttribute	Фільтр, що визначає, чи має користувач доступ до цього ресурсу. Даний фільтр запускається після фільтра аутентифікації, але до будь-якого іншого фільтра або методу дії
Фільтри дій	IActionFilter	ActionFilterAttribute	Фільтр, застосовуваний до дій. Може запуститися як до, так і після виконання методу дій
Фільтри результатів дій	IResultFilter	ResultFilterAttribute	Фільтр, застосовуваний до результатів дій. Може запускатися як до, так і після

			виконання результату дії
Фільтри винятків	ExceptionHandler	HandleErrorAttribute	Атрибут для обробки виключень, що викидаються методом дій і результатом дій

При виклику методу дій фреймворк MVC спочатку дивиться, чи застосовуються до даного дії які-небудь атрибути, і якщо застосовуються, то на початку відпрацьовують дані атрибути. Тепер розберемо перший фільтр - фільтр аутентифікації.

3.8.1 Фільтри аутентифікації

Фільтри аутентифікації з'явилися в ASP.NET MVC 5. Їх призначення - управління аутентифікацією користувачів. Фільтри аутентифікації спрацьовують до будь-якого іншого фільтру і виконання методу, а також тоді, коли метод вже завершив виконання, але його результат - об'єкт ActionResult - не був оброблений.

Будь-який фільтр аутентифікації реалізує інтерфейс IAuthenticationFilter:

```
namespace System.Web.Mvc.Filters
{
    public interface IAuthenticationFilter
    {
        void OnAuthentication(AuthenticationContext filterContext);
        void OnAuthenticationChallenge(AuthenticationChallengeContext filterContext);
    }
}
```

Реалізація методу On Authentication покликана забезпечити перевірку користувача - чи аутентифікований він у системі. Метод OnAuthenticationChallenge використовується для обмеження доступу для аутентифікованого користувача.

Розглянемо це на прикладі і створимо свій фільтр аутентифікації. Візьмемо який-небудь проект MVC 5, який використовує аутентифікацію. Створимо в ньому для зберігання фільтра спеціальну папку Filters і додамо в неї наступний клас MyAuthAttribute:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Mvc.Filters;
```

```

namespace AuthFilterApp.Filters
{
    public class MyAuthAttribute : FilterAttribute, IAuthenticationFilter
    {
        public void OnAuthentication(AuthenticationContext filterContext)
        {
            var user = filterContext.HttpContext.User;
            if (user == null || !user.Identity.IsAuthenticated)
            {
                filterContext.Result = new HttpUnauthorizedResult();
            }
        }

        public void OnAuthenticationChallenge(AuthenticationChallengeContext filterContext)
        {
            var user = filterContext.HttpContext.User;
            if (user == null || !user.Identity.IsAuthenticated)
            {
                filterContext.Result = new RedirectToRouteResult(
                    new System.Web.Routing.RouteValueDictionary {
                        { "controller", "Account" }, { "action", "Login" }
                    });
            }
        }
    }
}

```

По-перше, щоб створити свій фільтр, нам треба наслідувати клас від класу `FilterAttribute`, а також реалізувати інтерфейс `IAuthenticationFilter`. Реалізація двох його методів досить проста. Кожен метод приймає як параметр `filterContext`, який успадкований від контексту контролера `ControllerContext` і містить всю необхідну інформацію про контекст запиту. У тому числі він дозволяє дізнатися всю інформацію про користувача, чи він аутентифікований і т.д. І якщо користувач не аутентифікований, то повертається певний результат: `filterContext.Result = new HttpUnauthorizedResult()`, який говорить про те, що доступ до даного ресурсу для користувачів заборонений. У другому методі подібним чином ми перенаправляємо користувача на сторінку логіна через `filterContext.Result = new RedirectToRouteResult(new System.Web.Routing.RouteValueDictionary {{"controller", "Account"}, {"action", "Login"}});`

Простий фільтр готовий, і тепер ми можемо його застосувати. Наприклад, у нас є за замовчуванням в проекті контролер `HomeController`. Застосуємо фільтр до одного з його дій:

```

public class HomeController : Controller
{
    [MyAuthAttribute]
    public ActionResult Index()
    {
        return View();
    }
}

```

```
}
```

Тепер, якщо ми не залогінені на сайті, то доступ до дії буде заборонений, і нас буде перенаправляти на сторінку логіна.

3.8.2 Фільтри авторизації

Фільтри авторизації спрацьовують після фільтрів аутентифікації і до запуску інших фільтрів і виклику методів дій. Мета фільтрів авторизації - розмежувати доступ користувачів, щоб до певних ресурсів додатку мали доступ тільки певні користувачі.

Фільтри авторизації реалізують інтерфейс `IAuthorizationFilter`:

```
namespace System.Web.Mvc
{
    public interface IAuthorizationFilter
    {
        void OnAuthorization(AuthorizationContext filterContext);
    }
}
```

І якщо при отриманні запиту виявиться, що до запитуваної дії контролера застосовується даний фільтр, то спочатку спрацьовує метод `OnAuthorization` даного інтерфейсу. І якщо фільтр схвалить запит, то далі викликається дія. Інакше дія не буде працювати. У ASP.NET MVC 5 є вбудована реалізація даного фільтра - `AuthorizeAttribute`. Він застосовується за допомогою установки атрибута `[Authorize]` над контролером або методом контролера. Наприклад:

```
[Authorize]
public class HomeController : Controller
{
    //.....
}
```

Атрибут `AllowAnonymous`

Атрибут `AllowAnonymous` дозволяє дозволити доступ до ресурсів для анонімних, не авторизованих користувачів. Наприклад, якщо до контролера застосовується атрибут `Authorize`, то фреймворк спочатку буде дивитися, чи є у користувача необхідні права для доступу до методу контролера. Однак у цьому випадку ми потрапляємо в замкнуте коло - щоб авторизуватися, треба звернутися до методу `Login`, але щоб звернутися до цього методу, вже треба бути авторизованим. Тому застосовується атрибут `AllowAnonymous`, який відкриває публічний доступ до методу контролера:

Використання ролей при авторизації

Система авторизації в mvc дозволяє настроїти доступ до ресурсів сайту для окремих користувачів або груп користувачів. Для подібного налаштування можна використовувати дві властивості атрибута `AuthorizeAttribute`:

- Users - містить перерахування імен користувачів, яким дозволено вхід
- Roles - містить перерахування імен ролей, яким дозволено вхід

Використання властивостей Users і Roles:

```
[Authorize (Users="eugene, sergey")]
public ActionResult Index()
{
    .....
}
[Authorize (Roles="admin")]
public ActionResult Create()
{
    .....
}
[Authorize (Roles="admin, moderator", Users="eugene, sergey")]
public ActionResult Edit()
{
    .....
}
```

У разі якщо ролі і користувачі не вказані, то доступ до методів контролера має будь аутентифіцирований користувач.

Створення власного фільтра авторизації

Вище ми подивилися, як працює фільтр авторизації. Однак за кадром залишилися принципи роботи даного фільтра. Для цього ми створимо свій фільтр. Найбільш простий і безпечний спосіб створення свого фільтра авторизації - це спадкування класу від атрибуту `AuthorizeAttribute` і перевизначення методу `AuthorizeCore`:

```
using System;
using System.Web;
using System.Web.Mvc;
using System.Linq;

namespace FiltersApp.Filters
{
    public class MyAuthorizeAttribute : AuthorizeAttribute
    {
        private string[] allowedUsers = new string[] { };
        private string[] allowedRoles = new string[] { };

        public MyAuthorizeAttribute()
        {}

        protected override bool AuthorizeCore(HttpContextBase httpContext)
        {
            if (!String.IsNullOrEmpty(base.Users))
```



```

    {
        allowedUsers = base.Users.Split(new char[] { ',' });
        for (int i = 0; i < allowedUsers.Length; i++)
        {
            allowedUsers[i] = allowedUsers[i].Trim();
        }
    }
    if(!String.IsNullOrEmpty(base.Roles))
    {
        allowedRoles = base.Roles.Split(new char[] { ',' });
        for (int i = 0; i < allowedRoles.Length; i++)
        {
            allowedRoles[i] = allowedRoles[i].Trim();
        }
    }

    return httpContext.Request.IsAuthenticated &&
        User(httpContext) && Role(httpContext);
}

private bool User(HttpContextBase httpContext)
{
    if(allowedUsers.Length>0)
    {
        return allowedUsers.Contains(httpContext.User.Identity.Name);
    }
    return true;
}

private bool Role(HttpContextBase httpContext)
{
    if (allowedRoles.Length > 0)
    {
        for (int i = 0; i < allowedRoles.Length; i++)
        {
            if (httpContext.User.IsInRole(allowedRoles[i]))
                return true;
        }
        return false;
    }
    return true;
}
}
}

```

Масиви користувачів і ролей допомагають провести детальну авторизацію. У базовому класі атрибуту визначені змінні для ролей і користувачів, яких ми передаємо в атрибут. І, звернувшись до цих змінним через базовий клас - `base.Roles` і `base.Users`, ми їх можемо отримати. Так, якщо у нас використовуються наступні ролі: `Roles = "admin, moderator"`, то властивість

base.Roles також матиме значення "admin, moderator". І, виконуючи маніпуляції з рядком - розбиття по коми, відсікання початкових і кінцевих пробілів, ми можемо помістити всі передані ролі в масив.

Іншим моментом використання атрибуту є застосування об'єкта контексту HttpContextBase. З його допомогою можна отримати контекст запиту та інформацію про користувача, наприклад, аутентифікований користувач в системі. Застосування фільтра подібно до застосування стандартної версії:

```
[MyAuthorize (Roles="admin, moderator", Users="eugene, sergey")]
public ActionResult Index()
{
    .....
}
```

Перевизначення атрибуту AuthorizeAttribute найбільш безпечний спосіб створення механізму авторизації, у порівнянні, скажімо, з реалізацією в своєму класі інтерфейсу IAuthorizationFilter. Тому у випадках, якщо вам треба створити свій клас атрибута авторизації, рекомендується саме цей спосіб. Але можна і поєднувати спадкування атрибуту і перевизначення інтерфейсу IAuthorizationFilter:

```
public class CustomAuthAttribute : AuthorizeAttribute, IAuthorizationFilter
{
    public void OnAuthorization(AuthorizationContext filterContext)
    {
        // якщо користувач не належить до ролі admin, то він перенаправляється на Home/About
        bool auth = filterContext.HttpContext.User.IsInRole("admin");
        if(!auth)
        {
            filterContext.Result = new RedirectToRouteResult(
                new System.Web.Routing.RouteValueDictionary {
                    { "controller", "Home" }, { "action", "About" }
                });
        }
    }
}
```

Хоча в попередніх прикладах перевизначення авторизації по суті не мало сенсу, однак все-таки може знадобитися реалізація свого атрибута. Наприклад, щоби адміністраторам сервера сайту не доводилося щоразу набирати пароль при вході на сайт, ми можемо зробити наступну реалізацію атрибута:

```
public class LocalAuthorizeAttribute : AuthorizeAttribute
{
    protected override bool AuthorizeCore(HttpContextBase httpContext)
    {
        return httpContext.Request.IsLocal || base.AuthorizeCore(httpContext);
    }
}
```

Перевизначення атрибут також дозволяє використовувати ролі і користувачів:

```
[LocalAuthorize (Roles="admin", Users="eugene, sergey")]
public ActionResult Index()
{
    .....
}
```

Однак у цьому випадку заходити на сайт зможуть не тільки зазначені ролі і користувачі, а й локальні користувачі для даної машини (що ми дізнаємося за допомогою властивості `HttpContext.Request.IsLocal`).

3.8.3 Фільтри виключень

Фільтри виключень спрацьовують, якщо при виконанні методу дії буде викинуто необроблене виключення. З одного боку, ми могли помістити всю логіку виконання методу в блок `try ... catch` і відстежити виключення. Однак область роботи фільтрів виключення дещо ширша. Вони дозволяють відстежити не тільки виключення, що виникають в самому методі, але й винятки, що генеруються результатами дій, а також іншими застосовуваними до даного дії фільтрами. У цьому і полягає особливість даного типу фільтрів. Всі фільтри виключень повинні застосовувати інтерфейс `IFilter`:

```
public interface IFilter
{
    void OnException(ExceptionContext filterContext);
}
```

І якщо раптом додаток генерує необроблюваних винятків, то фільтр викликає метод `OnException`. Переданий в цей метод параметр - `ExceptionContext` є об'єктом, похідним від `ControllerContext`. Тому з нього можна отримати як специфічну для фільтра інформацію, так і загальну інформацію про запит. Зокрема клас `ExceptionContext` має такі властивості, які дозволяють отримати інформацію про виключення:

Властивість	Тип	Опис
ActionDescriptor	ActionDescriptor	Містить інформацію про метод дії, на якому було викинуто виключення
Exception	Exception	Представляє необроблене виключення
ExceptionHandled	bool	Значення, що показує, чи вважається виключення обробленим. Якщо ми на фільтрі помічаємо його значення в <code>true</code> , то

		виключення вважається обробленим
Result	ActionResult	Результат методу дії, до якого застосовується фільтр виключення

За допомогою властивості `Exception` ми можемо отримати доступ до викидає виключення. Встановивши властивість `ExceptionHandled` в `true`, фільтр тим самим позначає виняток як оброблений. За допомогою властивості `Result` фільтр управляє результатом дій. Загальнопоширеною практикою в даному випадку є перенаправлення користувача на сторінку помилки або відтворення помилки на екрані.

Тепер створимо фільтр, який буде обробляти виключення `IndexOutOfRangeException`:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace Filters.Filters
{
    public class IndexException : FilterAttribute, IExceptionHandler
    {
        public void OnException(ExceptionContext exceptionContext)
        {
            if (!exceptionContext.ExceptionHandled && exceptionContext.Exception is
                IndexOutOfRangeException)
            {
                exceptionContext.Result = new RedirectResult("/Content/ExceptionFound.html");
                exceptionContext.ExceptionHandled = true;
            }
        }
    }
}
```

Тут в методі `OnException` насамперед ми перевіряємо, чи не встановлено значення властивості `ExceptionHandled`. Якщо воно встановлено в `true` - якийсь інший фільтр виключень вже обробив дане виключення. Також перевіряється тип винятку, оскільки ми відловлюємо тільки виключення типу `IndexOutOfRangeException`.

Далі ми встановлюємо результат методу, до якого застосований фільтр. Передбачається, що в проекті в каталозі `Content` у нас знаходиться деяка сторінка `ExceptionFound.html`, яка відображає користувачеві повідомлення про помилку. В даному випадку важливо помітити виняток як оброблений: `exceptionContext.ExceptionHandled = true`, в іншому випадку ми можемо побачити

в браузері діагностичне повідомлення про помилку, яке зазвичай посилає фреймворк у відповідь клієнту.

Приклад використання:

```
[IndexException]
public ActionResult Index()
{
    int[] mas = new int[2];
    mas[6] = 4;
    return View();
}
```

У даному разі метод `Index` викине необроблене виключення, і воно буде об'єктом типу `IndexOutOfRangeException`, а користувач буде перенаправлений на сторінку `ExceptionFound.html`. Подібним чином ми можемо обробити ті чи інші типи винятків.

HandleErrorAttribute. Вбудована обробка виключень.

Створювати свої фільтри винятків необов'язково, так як у фреймворку є вбудована реалізація інтерфейсу `IExceptionHandler` - атрибут `HandleErrorAttribute`. Він має ряд властивостей, за допомогою яких ми можемо виробити гнучку настройку фільтра:

Властивість	Тип	Опис
ExceptionType	Type	Являє тип оброблюваного винятку. За замовчуванням використовується <code>System.Exception</code>
View	string	Ім'я представлення, яке рендериться даними фільтром. Якщо значення не задане, то за замовчуванням використовуються наступні шляхи: <code>/Views/Ім'я_контроллера/Error.cshtml</code> або <code>/Views/Shared/Error.cshtml</code>
Master	string	Ім'я використовуваної майстер-сторінки

При обробці виключення фільтр виключень посилає статусний код HTTP 500 і генерує зазначене у властивості `View` представлення. Наприклад, використаємо попередній приклад з фільтром виключень, застосувавши вбудовану реалізацію:

```
[HandleError(ExceptionType = typeof(System.IndexOutOfRangeException), View =
"ExceptionFound")]
public ActionResult About()
{
    int[] mas = new int[2];
    mas[6] = 4;
    return View();
}
```

```
}
```

У даному випадку очевидно, що на рядку `mas [6] = 4;` буде викинутий виключення. У режимі налагодження призупиниться виконання програми, тоді користувач може натиснути на кнопку Continue на панелі інструментів. Тут знову обробляється виключення типу `IndexOutOfRangeException`, і при виникненні такого виключення генерується у відповідь представлення `ExceptionFound.cshtml`. Дане представлення повинно знаходитися в проекті в каталозі `Views / Імя_контроллера /`.

Одразу треба сказати, що якщо ви хочете при розробці бачити оброблювані фільтром `HandleErrorAttribute`, то треба включити дану функціональність у файлі конфігурації `web.config` за допомогою тега `<customErrors mode = "On" />`:

```
<system.web> <customErrors mode = "On" />
<! - далі решту вмісту вузла system.web ->
</system.web>
```

3.8.4 Фільтри дій і результатів

Фільтр дій

Фільтри результатів дозволяють нам проконтролювати вхідний контекст запиту при доступі до дії, а також виконати певні дії по завершенню роботи методу дій. Наприклад, ми можемо змінити вихідний результат методу.

Фільтр дій повинен реалізувати інтерфейс `IActionFilter`:

```
public interface IActionFilter
{
    void OnActionExecuting(ActionExecutingContext filterContext);
    void OnActionExecuted(ActionExecutedContext filterContext);
}
```

Інакше ми повинні наслідувати новий фільтр від абстрактного класу `ActionFilterAttribute`, який вже реалізує інтерфейс `IResultFilter`:

```
public abstract class ActionFilterAttribute : FilterAttribute, IActionFilter, IResultFilter
```

Метод `OnActionExecuting` викликається перед викликом методу дій. А метод `OnActionExecuted` - після. За допомогою методу `OnActionExecuting` можна отримати інформацію про запит, змінити її або зовсім скасувати запит. Переданий в метод об'єкт `ActionExecutingContext` є похідним від класу `ControllerContext` і визначає наступні дві додаткових властивості:

Властивість	Тип	Опис
<code>ActionDescriptor</code>	<code>ActionDescriptor</code>	Надає інформацію про

		метод дії, що викликається
Result	ActionResult	Результат методу дій

За допомогою властивості Result ми можемо скасувати результат дії, присвоїти цій властивості інший результат дії. У метод OnActionExecuted як параметр передається об'єкт ActionExecutedContext, який також є похідним від класу ControllerContext. Він визначає деякі додаткові властивості:

Властивість	Тип	Опис
ActionDescriptor	ActionDescriptor	Надає інформацію про метод дії, що викликається
Canceled	bool	Зберігає значення, що показує, скасований чи ні виклик дії. Має значення true, якщо виклик дії скасували іншим фільтром
Exception	Exception	Повертає виключення, яке повертається даним методом дій чи іншим фільтром
ExceptionHandled	bool	Зберігає значення, що показує, чи оброблений виняток. Якщо true, то виключення оброблене
Result	ActionResult	Результат методу дій

Як приклад ми можемо створити наступний фільтр дії:

```
public class MyActionAttribute : FilterAttribute, IActionFilter
{
    public void OnActionExecuted(ActionExecutedContext filterContext)
    {
        filterContext.HttpContext.Response.Write("Дія виконана");
    }

    public void OnActionExecuting(ActionExecutingContext filterContext)
    {
        if (filterContext.HttpContext.Request.Browser.Browser=="Opera")
        {
            filterContext.Result = new HttpNotFoundResult();
        }
    }
}
```

І використання:

```
[MyAction]
public ActionResult Index()
{
    return View();
}
```

Тут в методі `OnActionExecuting` у випадку, якщо клієнт використовує браузер Опера (наприклад, ми хочемо заборонити підтримку свого сайту для цього браузера), то ми повертаємо клієнту статусний код 404. А в методі `OnActionExecuted` ми додаємо у вихідний потік рядок.

Фільтр результатів

Фільтри результатів багато в чому схожі на фільтри дій, оскільки так само можуть спрацьовувати як до повернення результату дії, так і після. Фільтри результатів реалізують інтерфейс `IResultFilter`:

```
public interface IResultFilter
{
    void OnResultExecuting(ResultExecutingContext filterContext);
    void OnResultExecuted(ResultExecutedContext filterContext);
}
```

Метод `OnResultExecuting` викликається перед тим, як метод дії почне повертати результат дії. А метод `OnResultExecuted` - після того, як метод дії повернув результат. Ці методи в якості параметрів приймають об'єкти `ResultExecutingContext` і `ResultExecutedContext`, які аналогічні об'єктам `OnActionExecuting` і `OnActionExecuted` відповідно. Наприклад, ми можемо виводити у вихідний потік ім'я користувача (якщо він авторизований) і поточний час запиту до сервера:

```
using System;
using System.Web;
using System.Web.Mvc;
namespace Filters.Filters
{
    public class MyResultAttribute: FilterAttribute, IResultFilter
    {
        public void OnResultExecuted(ResultExecutedContext filterContext)
        {
            filterContext.HttpContext.Response.Write("Час поточного запиту HTTP: " +
            filterContext.HttpContext.Timestamp);
        }

        public void OnResultExecuting(ResultExecutingContext filterContext)
        {

```



```

        filterContext.HttpContext.Response.Write("Поточний користувач: " +
filterContext.HttpContext.User.Identity.Name);
    }
}
}

```

Вбудована реалізація фільтрів дій і фільтрів результатів

Як вже було сказано, фільтри дій і фільтри результатів об'єднані в одну реалізацію - абстрактний клас `ActionFilterAttribute`, який поєднує риси обох фільтрів:

```

public abstract class ActionFilterAttribute : FilterAttribute, IActionFilter, IResultFilter
{
    public virtual void OnActionExecuting(ActionExecutingContext filterContext){}
    public virtual void OnActionExecuted(ActionExecutedContext filterContext){}
    public virtual void OnResultExecuting(ResultExecutingContext filterContext) {}
    public virtual void OnResultExecuted(ResultExecutedContext filterContext) {}
}

```

Ми можемо реалізувати тільки ті методи, які нам потрібні, або відразу все, що робить даний клас зручним для практичного використання.

3.8.5 Приклади фільтрів дій

Розглянемо деякі фільтри дій, які можуть використовуватися в різноманітних ситуаціях.

Фільтр кешування

```

using System.Web;
using System.Web.Mvc;
// .....
public class Cache Attribute: ActionFilterAttribute
{
    // Час кешування, за замовчуванням - 3600 секунд
    public int Duration {get; set; }
    Public CacheAttribute ()
    {
        Duration = 3 600;
    }

    public override void OnActionExecuted (ActionExecutedContext filterContext)
    {
        // якщо встановили тривалість в 0, то кешування не застосовується
        if (Duration <= 0) return;
        HttpCachePolicyBase cache = filterContext.HttpContext.Response.Cache;
        TimeSpan cacheDuration = TimeSpan.FromSeconds (Duration);
    }
}

```

```
// Задаємо публічний кеш
cache.SetCacheability (HttpCacheability.Public);
// Установка тривалості кешування
    cache.SetExpires (DateTime.Now.Add (cacheDuration));
// Установка параметра max-age cache.SetMaxAge (cacheDuration);
// Додаємо додаткові параметри для кешування
cache.AppendCacheExtension ("must-revalidate, proxy-revalidate");
}
}
```

Даний фільтр в багато чому аналогічний вбудованому фільтру кешування `OutputCache` за тим винятком, що в даному випадку сам програміст може визначити всі параметри кешування при різних умовах. Наприклад, визначити різну логіку кешування для різних браузерів і т.д. Для управління часом задається властивість `Duration`, що приймає час в секундах.

Застосування фільтру:

```
[Cache (Duration=300)]
public ActionResult Index()
{
    //.....
}
```

У підсумку при зверненні до методу ми отримаємо наступні заголовки:

```
HTTP/1.1 200 OK
Cache-Control: public, must-revalidate, proxy-revalidate, max-age=300
//.....
```

Фільтр компресії

```
using System.Web;
using System.Web.Mvc;
using System.IO.Compression;
//.....
public class CompressAttribute : ActionFilterAttribute
{
    public override void OnActionExecuting(ActionExecutingContext filterContext)
    {
        HttpRequestBase request = filterContext.HttpContext.Request;
        //одержуємо заголовок Accept-Encoding, який вказує
        //які алгоритми стиску він підтримує
        string acceptEncoding = request.Headers["Accept-Encoding"];
        if (string.IsNullOrEmpty(acceptEncoding)) return;
        acceptEncoding = acceptEncoding.ToUpperInvariant();
        HttpResponseBase response = filterContext.HttpContext.Response;
        if (acceptEncoding.Contains("GZIP"))
        {
            response.AppendHeader("Content-encoding", "gzip");
            response.Filter = new GZipStream(response.Filter, CompressionMode.Compress);
        }
    }
}
```

```

    }
    else if (acceptEncoding.Contains("DEFLATE"))
    {
        response.AppendHeader("Content-encoding", "deflate");
        response.Filter = new DeflateStream(response.Filter, CompressionMode.Compress);
    }
}
}

```

За допомогою заголовка Accept-Encoding у фільтрі отримуємо, який алгоритм стиснення підтримується для браузерів: deflate або gzip (нерідко підтримуються обидва, тоді вибираємо gzip). Потім в залежності від умов використовуємо або клас GZipStream, або DeflateStream для стиснення вихідного потоку.

Фільтр пробілів

```

using System;
using System.Collections.Generic;
using System.IO;
using System.Text;
using System.Text.RegularExpressions;
using System.Web.Mvc;
//.....
public class WhitespaceAttribute : ActionFilterAttribute
{
    public override void OnActionExecuted(ActionExecutedContext filterContext)
    {
        var response = filterContext.HttpContext.Response;
        // Якщо sitemap, то нічого не робимо
        if (filterContext.HttpContext.Request.RawUrl == "/sitemap.xml") return;

        if (response.ContentType != "text/html" || response.Filter == null) return;

        response.Filter = new SpaceCleaner(response.Filter);
    }
    // додатковий клас для усунення пробілів
    private class SpaceCleaner : Stream
    {
        private readonly Stream outputStream;
        StringBuilder _s = new StringBuilder();

        public SpaceCleaner(Stream filterStream)
        {
            if (filterStream == null)
                throw new ArgumentNullException("filterStream is not determined");
            outputStream = filterStream;
        }

        public override void Write(byte[] buffer, int offset, int count)
        {

```

```

        var html = Encoding.UTF8.GetString(buffer, offset, count);
        //регулярний вираз для пошуку пробілів між тегами
        var reg = new Regex(@"(?<=\s)\s+(?![^\<>]*</pre>)");
        html = reg.Replace(html, string.Empty);
        buffer = Encoding.UTF8.GetBytes(html);
        outputStream.Write(buffer, 0, buffer.Length);
    }
    // нереалізовані методи Stream
    public override int Read(byte[] buffer, int offset, int count)
    {
        throw new NotSupportedException();
    }
    public override bool CanRead { get { return false; } }
    public override bool CanSeek { get { return false; } }
    public override bool CanWrite { get { return true; } }
    public override long Length { get { throw new NotSupportedException(); } }
    public override long Position
    {
        get { throw new NotSupportedException(); }
        set { throw new NotSupportedException(); }
    }
    public override void Flush()
    {
        outputStream.Flush();
    }
    public override long Seek(long offset, SeekOrigin origin)
    {
        throw new NotSupportedException();
    }
    public override void SetLength(long value)
    {
        throw new NotSupportedException();
    }
}
}

```

Даний фільтр отримує вихідний потік у вигляді об'єкта `filterContext.HttpContext.Response.Filter` і передає його для обробки в об'єкт `SpaceCleaner`. `SpaceCleaner` за допомогою регулярних виразів знаходить всі пробіли, видаляє їх і змінені дані записує у вихідний потік. Таким чином, відбувається мінімізація html-контенту, обсяг даних зменшується.

Фільтр логування

Спочатку визначимо модель, яка буде описувати користувача сайту, і контекст даних:

```

public class Visitor
{
    public int Id { get; set; }
    public string Login { get; set; }
    public string Ip { get; set; }
}

```

```

    public string Url { get; set; }
    public DateTime Date { get; set; }
}

```

```

public class LogContext : DbContext
{
    public DbSet<Visitor> Visitors { get; set; }
}

```

Тоді фільтр, який продукує логування в базу даних, буде виглядати наступним чином:

```

using System.Web.Mvc;
using SomeActionFiltersApp.Models;
using System;
//.....
public class LogAttribute : ActionFilterAttribute
{
    public override void OnActionExecuting(ActionExecutingContext filterContext)
    {
        var request = filterContext.HttpContext.Request;

        Visitor visitor= new Visitor()
        {
            Login = (request.IsAuthenticated) ? filterContext.HttpContext.User.Identity.Name : "null",
            Ip = request.ServerVariables["HTTP_X_FORWARDED_FOR"] ?? request.UserHostAddress,
            Url = request.RawUrl,
            Date = DateTime.UtcNow
        };

        using(LogContext db = new LogContext())
        {
            db.Visitors.Add(visitor);
            db.SaveChanges();
        }
        base.OnActionExecuting(filterContext);
    }
}

```

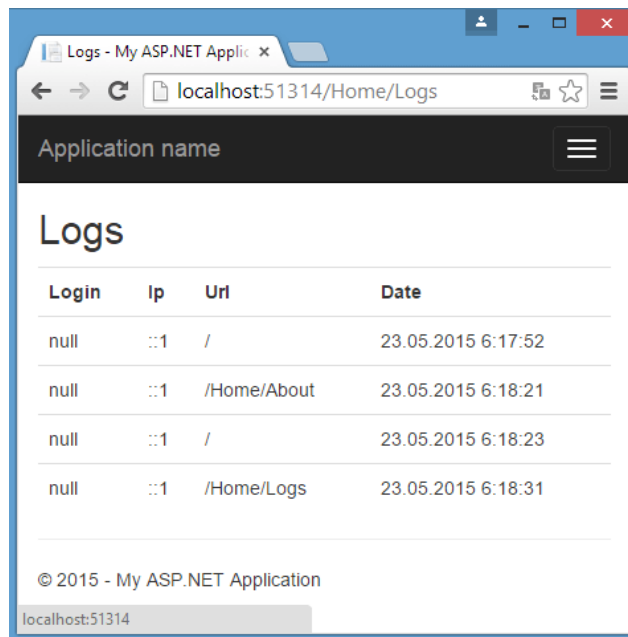


Рисунок 3.41 - Логги

3.8.6 Глобальні фільтри і перевизначення фільтрів

Глобальні фільтри

Ми подивилися на різні види фільтрів. Ми можемо застосувати фільтри як до окремих дій, так і до всього контролеру - тобто до всіх його дій. Однак якщо ми хочемо застосувати фільтр до всіх контролерів, а контролерів у нас багато? В даному випадку в MVC реалізована функціональність глобальних фільтрів. Ці фільтри застосовуються до всіх дій всіх контролерів.

Відкриємо файл FilterConfig.cs, який знаходиться в папці App_Start. У цьому файлі визначено клас FilterConfig, який в методі RegisterGlobalFilters здійснює реєстрацію глобальних фільтрів в колекцію filters. Ця колекція являє собою об'єкт GlobalFilterCollection, який передається як параметр при виклику методу у файлі Global.asax.

Отже, ми можемо зареєструвати глобальні фільтри наступним чином:

```
public class FilterConfig
{
    public static void RegisterGlobalFilters(GlobalFilterCollection filters)
    {
        filters.Add(new HandleErrorAttribute());

        filters.Add(new MyExceptionAttribute());
        filters.Add(new MyResultAttribute());
    }
}
```

Перевизначення фільтрів

Одним з нововведень MVC 5 стала така функціональність як перевизначення фільтрів. Щоб зрозуміти її суть, розглянемо таку ситуацію. Наприклад, у нас є контролер з обмеженням доступу:

```
[Authorize(Users = "eugene")]
public class HomeController : Controller
{
    public ActionResult Index()
    {
        return View();
    }

    [Authorize(Users = "Sergiy")]
    public string Test()
    {
        return "Test";
    }
}
```

Атрибут авторизації на рівні контролера `[Authorize (Users = "eugene")]` обмежує доступ до методів контролера: в даному випадку вони доступні тільки користувачу з логіном `eugene`. І очевидно, що другий атрибут авторизації для методу `Test` працювати не буде. Однак така можливість була б дуже корисна, особливо при використанні глобальних фільтрів, коли нам треба відкрити або перевизначити доступ. Проблема стосується не тільки фільтрів авторизації, а й будь-яких інших. І щоб вирішити дану проблему в MVC 5 була введена така функціональність, як перевизначення фільтрів. Вона має на увазі реалізацію інтерфейсу `IOVERRIDEFilter`:

```
namespace System.Web.Mvc.Filters
{
    public interface IOVERRIDEFilter : IFilter
    {
        Type FiltersToOverride{get;};
    }
}
```

Єдина властивість інтерфейсу дозволяє вказати тип фільтрів, які треба перевизначити. Отже, створимо новий клас, який реалізує даний інтерфейс:

```
public class OverrideFilterAttribute : FilterAttribute, IOVERRIDEFilter
{
    public Type FiltersToOverride
    {
        get { return typeof(IAuthorizationFilter); }
    }
}
```

Так як нам треба перевизначити поведінку фільтрів авторизації, то в якості типу вказується `IAuthorizationFilter`. Тепер ми можемо переписати контролер наступним чином:

```
[Authorize(Users = "eugene")]
public class HomeController : Controller
{
    public ActionResult Index()
    {
        return View();
    }

    [OverrideFilter]
    [Authorize(Users = "Sergiy")]
    public string Test()
    {
        return "Test";
    }
}
```

Таким чином, фільтр авторизації `[Authorize (Users = "eugene")]` буде застосовуватися до всіх дій контролера, крім методу `Test`.

3.8.7 Додаткові вбудовані фільтри

Фільтр `RequireHttps`

Фільтр `RequireHttps` змушує використовувати протокол `HTTPS`, а браузер перенаправить користувача на ту ж дію, тільки з префіксом `https`. Даний фільтр застосовується тільки до `GET`-запитів.

Фільтр `OutputCache`

Даний фільтр вказує `mvc`-фреймворку кешувати висновок методу дії, щоб отриманий контент можна було в подальшому використовувати повторно. Кешування може збільшити продуктивність, особливо коли йдеться про вибірку з бази даних, яка може займати значний час. За допомогою параметра `Duration` ми можемо налаштувати час (у секундах):

```
[OutputCache (Duration=360)]
public ActionResult Index()
{
    //.....
}
```

Атрибут `ValidateAntiforgeryToken`

Фільтр `ValidateAntiforgeryToken` призначений для протидії підробці міжсайтових запитів, виробляючи верифікацію токенів при зверненні до методу дії. Найбільш частим випадком є застосування даного фільтра до методів, що відповідає за авторизацію, наприклад:


```

[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Login(LoginViewModel model, string returnUrl)
{
    if (ModelState.IsValid)
    {
        //.....
    }
}

```

Крім того, нам треба використовувати хелпер `@Html.AntiForgeryToken()` для використання токена:

```

@using (Html.BeginForm("Login", "Account", FormMethod.Post))
{
    @Html.AntiForgeryToken()
    //.....
}

```

Контрольні питання.

1. Особливості фільтрації аутентифікації.
2. Фільтрація виключень.
3. Перевизначення фільтрів.

3.9 Прив'язка моделі

3.9.1 Введення в прив'язку моделей

У попередніх розділах ми розглянули моделі і дізналися, як передавати в методи контролера значення з представлень. Але за кадром залишився весь механізм, який сприяє зіставленню значень з використовуваними в методах контролера параметрами. Цей механізм називається прив'язкою моделі.

Щоб зрозуміти сенс прив'язки, розглянемо приклад. Припустимо, ми відправляємо на сервер форму з деякими даними для моделі `Book`. І в певному методі контролера ми можемо отримати передані дані з форми:

```

[HttpPost]
public ActionResult Edit()
{
    Book b = new Book();
    b.Name = Request.Form["Name"];
    b.Author = Request.Form["Author"];
    // .....
}

```

Однак властивостей моделі Book, які треба отримати із запиту, може бути безліч. Крім того, ви можете помітити, що поля для введення властивостей моделі мають ім'я (атрибут name), що збігається з ім'ям властивості моделі. Щоб полегшити процес отримання надісланих даних і був створений механізм прив'язки. Завдяки чому ми можемо просто написати:

```
[HttpPost]
public ActionResult Edit(Book b)
{
    //.....
}
```

При цьому прив'язка моделі не обмежена POST-запитами і складними параметрами типу об'єкта Book. Прив'язка може також працювати і з більш простими параметрами, наприклад з наступним дією Edit, оброблювальним HTTP-запит GET:

```
public ActionResult Edit(int id)
{
    // ...
}
```

Прив'язка моделей

Після зіставлення запиту з деяким маршрутом контролерів називаються спеціальні компоненти - активатори дій (action invoker), які викликають потрібну дію контролера і передають у нього значення з контексту запиту. Щоб зіставити отримані значення з конкретними параметрами активатори дій, зокрема, активатор за замовчуванням - ControllerActionInvoker, використовують прив'язчики моделі (model binder). Прив'язчики моделей і здійснюють власне прив'язку.

Всі прив'язчики моделей повинні реалізувати інтерфейс IModelBinder:

```
public interface IModelBinder
{
    object BindModel (ControllerContext controllerContext,
                     ModelBindingContext bindingContext);
}
```

В принципі для кожного окремого типу може існувати свій прив'язка моделі. При перегляді параметрів методу дії активатор дій шукає для кожного типу параметра відповідний прив'язчик і викликає його метод BindModel. У випадку, якщо відповідного даному типу прив'язчика не виявиться, то використовується прив'язка за замовчуванням - DefaultModelBinder. Потім прив'язчик використовує спеціальні компоненти - поставщики значень (value provider) для пошуку значень в різних частинах запиту.

DefaultModelBinder

Прив'язка DefaultModelBinder використовується за замовчуванням, якщо для даного типу не визначений інший прив'язчик. Щоб отримати значення для параметрів, прив'язчик переглядає наступні об'єкти строго по порядку:

- Request.Form. В даному випадку прив'язчик отримує значення, відправлені за допомогою форм
- RouteData.Values. Тут отримуються значення за допомогою маршрутів додатку
- Request.QueryString. В даному випадку прив'язчик одержує значення з рядка запиту
- Request.Files. У даному випадку використовуються завантажені на сервер файли

Якщо в одному з цих об'єктів буде виявлений відповідний елемент, то на ньому пошук припиняється, а параметру передається знайдене значення. Наприклад, прив'язчик шукає значення для параметрів наступного дії:

```
public ActionResult Edit(int id)
{
    // ...
}
```

Щоб DefaultModelBinder міг зв'язати значення з параметром, елемент даних запиту повинен обов'язково мати те ж ім'я, що і параметр. Тобто в даному випадку вони обидва повинні мати ім'я id. Отже, прив'язка DefaultModelBinder переглядає в пошуку значення для параметра id наступні шляхи:

```
Request.Form ["id"]
RouteData.Values ["id"]
Request.QueryString ["id"]
Request.Files ["id"]
```

При прив'язці простих типів DefaultModelBinder перетворює рядковий вираз параметра з даних запиту до типу параметра за допомогою класу System.ComponentModel.TypeDescriptor. Якщо ж значення не можна перетворити, то прив'язка закінчується невдачею. Тому є сенс використовувати параметри, що допускають значення null, а в метод дії вкладати відповідну логіку з перевірки значення:

```
public ActionResult Edit(int? id)
{
    // ...
}
```

При отриманні значень для складних типів, наприклад, для тієї ж моделі Book, DefaultModelBinder за допомогою рефлексії переглядає об'єкт і знаходить всі властивості об'єкта, які можуть використовуватися при прив'язці. Наприклад,

DefaultModelBinder бачить, що об'єкт Book має властивість Author, то прив'язчик буде шукати в запиті параметр "Author". Якщо властивість представляє простий тип, то для нього використовується, як було зазначено вище перетворення за допомогою класу класу System.ComponentModel.TypeDescriptor. Якщо ж властивість представляє складний тип - то знову повторюється рефлексія і весь цикл.

3.9.2 Вибіркова прив'язка

Іноді виникає можливість виключити деякі властивості з прив'язки моделі. Ми можемо це зробити за допомогою атрибуту Bind. Для включення тільки певних властивостей ми можемо використовувати властивість Include даного атрибута:

```
public ActionResult Create([Bind (Include="Name, Author")] Book b)
{
    // ...
}
```

У даному випадку ми вказуємо, що в прив'язці братимуть участь тільки властивості Name і Author. Решта ж властивостей з прив'язки виключаються. Інакше ми можемо використовувати властивість Exclude атрибута Bind, щоб виключити властивість з прив'язки:

```
public ActionResult Create([Bind (Exclude="Year")] Book b)
{
    // ...
}
```

В даному випадку вибіркова прив'язка застосована до методу Create. Однак, якщо нам треба здійснити вибірку прив'язку глобально у всьому додатку? Тоді ми можемо застосувати атрибут безпосередньо до моделі, і в цьому випадку атрибут буде застосований за замовчуванням до всіх методів дій контролерів проекту:

```
[Bind (Exclude="Year")]
public class Book
{
    [ScaffoldColumn(false)]
    public int Id { get; set; }

    [Required (ErrorMessage="Поле повинне бути встановлене")]
    [Display(Name = "Назва")]
    public string Name { get; set; }

    [Required]
    [Display(Name = "Автор")]
    public string Author { get; set; }
```

```

[Display(Name = "Рік")]
public int Year { get; set; }
}

```

3.9.3 Явна прив'язка моделі

При використанні параметра в методі дії прив'язка моделі працює неявно. Але ми можемо викликати в контролері і явну прив'язку моделі за допомогою методів `UpdateModel` і `TryUpdateModel`. Якщо модель не пройшла валідацію, то метод `UpdateModel` генерує виключення. Нижче показаний приклад використання методу `UpdateModel` в дії `Edit` замість застосування параметра:

```

[HttpPost]
public ActionResult Edit()
{
    var book = new Book();
    try
    {
        UpdateModel(book);
        db.Entry(book).State = EntityState.Modified;
        db.SaveChanges();
        return RedirectToAction("Index");
    }
    catch
    {
        ViewBag.Message="Під час редагування виникли помилки";
        return RedirectToAction("Index");
    }
}

```

`TryUpdateModel` також викликає прив'язку моделі, але не викидає виключення. Цей метод повертає значення типу `bool` - якщо це значення дорівнює `true`, модель пройшла прив'язку, якщо `false`, то валідація пройшла невдало.

```

[HttpPost]
public ActionResult Edit()
{
    var book = new Book();
    if (TryUpdateModel(book))
    {
        db.Entry(book).State = EntityState.Modified;
        db.SaveChanges();
        return RedirectToAction("Index");
    }
    else
    {
        ViewBag.Message=" Під час редагування виникли помилки ";
        return RedirectToAction("Index");
    }
}

```

```
}  
}
```

3.9.4 Постачальники значень

Як вже говорилося, для пошуку значень прив'язка моделі використовує спеціальні компоненти - постачальники значень (value providers). Постачальники значень надають доступ до інформації, яка потім використовується в прив'язці моделі. Постачальники значень поставляються фабриками постачальників значень (value provider factories) і системою пошуку даних для цих постачальників значень в тому порядку, в якому вони зареєстровані (реєстрація за принципом черзі - вгорі першого, внизу останній). Розробники можуть створювати свої власні фабрики і постачальники значень і додавати їх в список фабрик `ValueProviderFactories.Factories`. Для надання додаткового джерела даних для прив'язки моделі розробники можуть застосувати фабрику або постачальник значень.

Усі постачальники значень реалізують інтерфейс `IValueProvider`:

```
public interface IValueProvider  
{  
    bool ContainPrefix (string prefix);  
    ValueProviderResult GetValue (string key);  
}
```

У конкретних реалізаціях метод `ContainPrefix` зіставляє деякі дані з вказаним префіксом, а метод `GetValue` отримує дані для певного ключа. Створимо свій постачальник значень. Нехай, наприклад, він буде отримувати для нас назву браузера користувача. Визначимо постачальник:

```
public class BrowserValueProvider : IValueProvider  
{  
    public bool ContainsPrefix(string prefix)  
    {  
        return string.Compare("browser", prefix, true) == 0;  
    }  
  
    public ValueProviderResult GetValue(string key)  
    {  
        return ContainsPrefix(key) ? new ValueProviderResult("Ваш браузер:  
"+HttpContext.Current.Request.Browser.Browser, null,  
            CultureInfo.InvariantCulture) : null;  
    }  
}
```

У методі `GetValue`, якщо ключ збігається з префіксом, то ми повертаємо об'єкт `ValueProviderResult`, в конструктор якого передаємо три параметри - перший параметр приймає значення, асоційоване з ключем. Другий параметр використовується для відстеження помилок, а в третьому параметрі передається

інформація про культуру, з яким асоціюється значення. В даному випадку другий і третій параметр нам не настільки важливі. Як уже писалося вище, щоб задіяти свій постачальник значень, нам треба визначити фабрику постачальника:

```
public class BrowserValueProviderFactory : ValueProviderFactory
{
    public override IValueProvider GetValueProvider(ControllerContext controllerContext)
    {
        return new BrowserValueProvider();
    }
}
```

І в кінці нам треба зареєструвати фабрику постачальника значень у файлі Global.asax:

```
public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        ValueProviderFactories.Factories.Add(new BrowserValueProviderFactory());

        //.....
    }
}
```

У даному випадку ми додаємо наш постачальник в кінець колекції ValueProviderFactories.Factories. Фреймворк переглядає послідовно всі фабрики постачальників значень в порядку реєстрації. Якщо буде знайдено відповідний постачальник, то наступні після нього вже не використовуються. У нашому випадку це не має значення. Однак якщо виникне необхідність поставити свою фабрику постачальника значень на перше місце в списку, то ми можемо зробити це таким чином:

```
ValueProviderFactories.Factories.Insert (0, new BrowserValueProviderFactory ());
```

Тепер ми можемо визначити метод дії, в якому протестуємо роботу постачальника значень:

```
public string BrowserInfo(string browser)
{
    return browser;
}
```

Отже, відправивши запит даному методу дій, ми отримаємо в якості відповіді назва нашого браузера. Ми не відправляємо у формі або в рядку запиту параметр browser і його значення. Його знаходить сам наш постачальник.

3.9.5 Створення прив'язки моделі

Щоб створити прив'язку моделі, нам треба реалізувати інтерфейс `IModelBinder`:

```
public class BookModelBinder : IModelBinder
{
    public object BindModel(ControllerContext controllerContext, ModelBindingContext
bindingContext)
    {
        // Одержуємо поставщик значень
        var valueProvider = bindingContext.ValueProvider;

        // Одержуємо дані по одному полю
        ValueProviderResult vprId = valueProvider.GetValue("Id");

        // Одержуємо дані по остальним полям
        string name = (string)valueProvider.GetValue("Name").ConvertTo(typeof(string));
        string author = (string)valueProvider.GetValue("Author").ConvertTo(typeof(string));
        int year = (int)valueProvider.GetValue("Year").ConvertTo(typeof(int));
        Book book = new Book() { Name = name + " (new)", Author = author, Year = year };

        // якщо поле Id визначено(редагування)
        if (vprId != null)
        {
            book.Name = name; // без new
            book.Id=(int)vprId.ConvertTo(typeof(int));
        }
        return book;
    }
}
```

Даний інтерфейс містить єдиний метод `BindModel`, який ми і реалізуємо в новому класі.

Спочатку ми отримуємо постачальник значень, а потім з його допомогою отримуємо значення для окремих властивостей майбутнього об'єкта `Book`. Щоб коректним чином отримати значення, ми перетворимо його до належного типу: `int id = (int) valueProvider.GetValue ("Id").ConvertTo (typeof (int)) ;`

І в кінці повертаємо створений об'єкт `Book`. Тепер нам треба зареєструвати прив'язчик моделі. Це можна зробити двома способами. Перший спосіб - реєстрація у файлі `Global.asax`:

```
public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        ModelBinders.Binders.Add(typeof(Book), new BookModelBinder());

        //.....
    }
}
```



```
}
```

Альтернативний спосіб реєстрації прив'язки - використання відповідного атрибуту для моделі:

```
[ModelBinder(typeof(BookModelBinder))]  
public class Book  
{  
    //.....  
}
```

Тепер для моделі Book використовуватиметься прив'язчик Book ModelBinder. Ну і в будь-якому методі, який приймає як параметр об'єкт Book, можна протестувати прив'язчик:

```
[HttpPost]  
public ActionResult Index(Book book)  
{  
    //.....  
}
```

Контрольні питання.

1. Особливості явної прив'язки моделей.
2. Особливості вибіркової прив'язки моделей.
3. Робота постачальника значень.

3.10 JavaScript і AJAX

3.10.1 Використання JavaScript

Сучасні веб-додатки практично неможливо уявити без мови клієнтської частини - JavaScript. Навіть при використанні таких серверних мов і технологій, як PHP, ASP.NET, важко обійтися без JavaScript. Проте чистий JavaScript в реальності використовується все менше. Йому на зміну приходять спеціальні бібліотеки, зокрема, jQuery. Стосовно до ASP.NET MVC при створенні веб-додатків бібліотеки jQuery відіграють дуже велику роль.

Підключення файлів JavaScript / jQuery

За замовчуванням проект ASP.NET MVC 5 вже містить необхідний базовий набір скриптів, у тому числі бібліотеки jQuery:

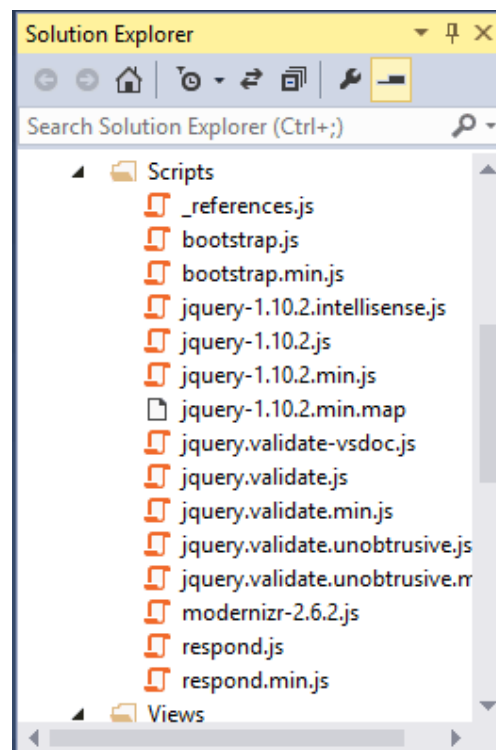


Рисунок 3.42 - Підключення JavaScript

Більшість скриптів за замовчуванням мають свої двійники з суфіксом min, наприклад, jquery-1.10.2.js і jquery-1.10.2.min.js. Обидва скрипти представляють одну і ту ж функціональність. Але друга версія являє мінімізовану версію (тому і йде з суфіксом min). Подібні мінімізовані скрипти набагато менше за обсягом (у середньому на 60%), тому в реальних додатках краще використовувати саме мінімізовані скрипти, так як користувач витрачає менше часу і трафіку на їх завантаження. У той же час їх не дуже зручно читати. Тому для більшої зручності розробників повні та мінімізовані скрипти базових бібліотек йдуть разом. Подивимося, навіщо потрібні більшість скриптів, що йдуть за замовчуванням у проекті MVC 5 з типом аутентифікації Individual User Accounts:

- jquery-1.10.2.js - базова бібліотека jQuery, на яку спираються більшість інших скриптів. У даному випадку використовується версія 1.10.2. Однак бібліотека постійно оновлюється, тому можна використовувати більш нові версії, які можна додати вручну або через NuGet.
- modernizr-2.6.2.js - бібліотека, що дозволяє визначити, чи підтримує браузер ті або інші можливості HTML5 та CSS3
- bootstrap.js - бібліотека, що дозволяє створювати адаптивні веб-додатки з використанням css-фреймворку bootstrap
- respond.js - дозволяє використовувати правила media queries CSS3 в старих браузерах, які напряду не підтримують дану можливість
- jquery.validate.js - представляє функціонал для валідації на стороні клієнта
- jquery.validate.unobtrusive.js - надає підтримку ненав'язливою валідації моделі
- jquery-1.10.2.intellisense.js і jquery.validate-vsdoc.js - використовуються для підтримки документації та IntelliSense за відповідними бібліотекам у Visual Studio

У деяких типах проектів ряд поширених скриптів може бути відсутнім, наприклад, якщо робити проект MVC 5 з типом аутентифікації No Authentication, то в ньому будуть відсутні скрипти валідації. У цьому випадку ми можемо знайти відсутній скрипт в NuGet і додати його в проект:

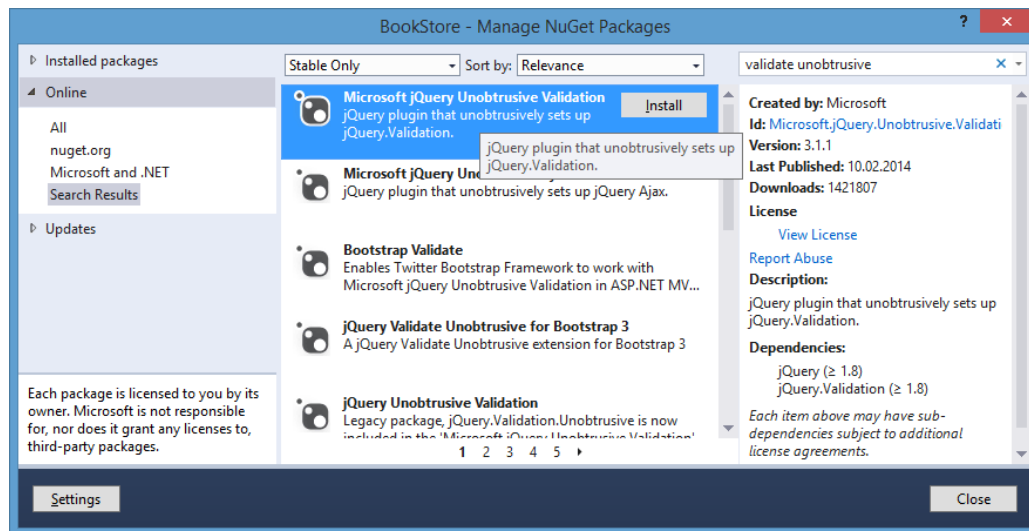


Рисунок 3.43 - Приєднання скрипта

Щоб підключити файл javascript використовується метод Render класу System.Web.Optimization.Scripts:

@ Scripts.Render ("~/ scripts / jquery.validate.min.js")

Цей метод приймає як параметр рядок - повний шлях до скрипту. Також для підключення скриптів ми можемо використовувати хелпер `Url.Content`:

```
<script src="@Url.Content("~/scripts/jquery.validate.min.js")"
type="text/javascript"> </script>
```

Якщо файли javascript будуть використовуватися більшістю представлень додатку, то найзручніше додати їх відразу на майстер-сторінку.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width" />
  <title>@ViewBag.Title</title>
  @Styles.Render("~/Content/css")
  @Scripts.Render("~/scripts/jquery-1.10.2.min.js")
  @Scripts.Render("~/scripts/validate.min.js")
</head>
<body>
  @RenderBody()
  @RenderSection("scripts", required: false)
</body>
</html>
```

Якщо ж скрипти використовуються не на всіх, а на окремих представленнях, то зручно визначити у майстер-сторінці спеціальну секцію для скриптів, як у прикладі вище. А потім у представленні підключати скрипти в цій секції:

```
@{
  ViewBag.Title = "Index";
}
<! - Далі основний зміст представлення ->
<!--секція скриптів-->
@section Scripts {
  @Scripts.Render("~/scripts/validate.min.js")
}
```

3.10.2 Вступ в AJAX

AJAX (Асинхронний JavaScript і XML) являє собою технологію гнучкої взаємодії між клієнтом і сервером. Завдяки її використанню ми можемо здійснювати асинхронні запити до сервера без перезавантаження всієї сторінки. Правда, в даний час все більше замість формату XML використовується формат JSON для взаємодії між клієнтом і сервером. Стосовно до ASP.NET MVC використання AJAX вилилося в цілу концепцію під назвою "ненав'язливого AJAX" і ненав'язливого JavaScript (unobtrusive Ajax/JavaScript). Сенс цієї концепції полягає в тому, що весь необхідний код JavaScript використовується не на самій веб-сторінці, а розміщується в окремі файли з розширенням *.js. А

потім за допомогою тега `<script>` ми в веб-станиці посилаємося на даний файл коду. Таким чином ми відокремлюємо візуалізацію від логіки додатка. Так, виділення скрипта в окремий файл завантаження збільшує продуктивність сайту, оскільки файл зберігається в кеші і потім від туди подгружається. Налаштування ненав'язливого JavaScript / AJAX. По-перше, відповідні налаштування повинні бути вказані у файлі Web.config:

```
<appSettings>
  <add key="webpages:Version" value="3.0.0.0" />
  <add key="webpages:Enabled" value="false" />
  <add key="ClientValidationEnabled" value="true" />
  <add key="UnobtrusiveJavaScriptEnabled" value="true" />
</appSettings>
```

По-друге, нам треба підключити відповідні файли JavaScript:

```
@Scripts.Render("~/scripts/jquery-1.10.2.js")
@Scripts.Render("~/scripts/jquery.unobtrusive-ajax.js")
```

Якщо у нас в проєкті відсутня бібліотека `jquery.unobtrusive-ajax.js`, то ми її знову можемо додати через NuGet:

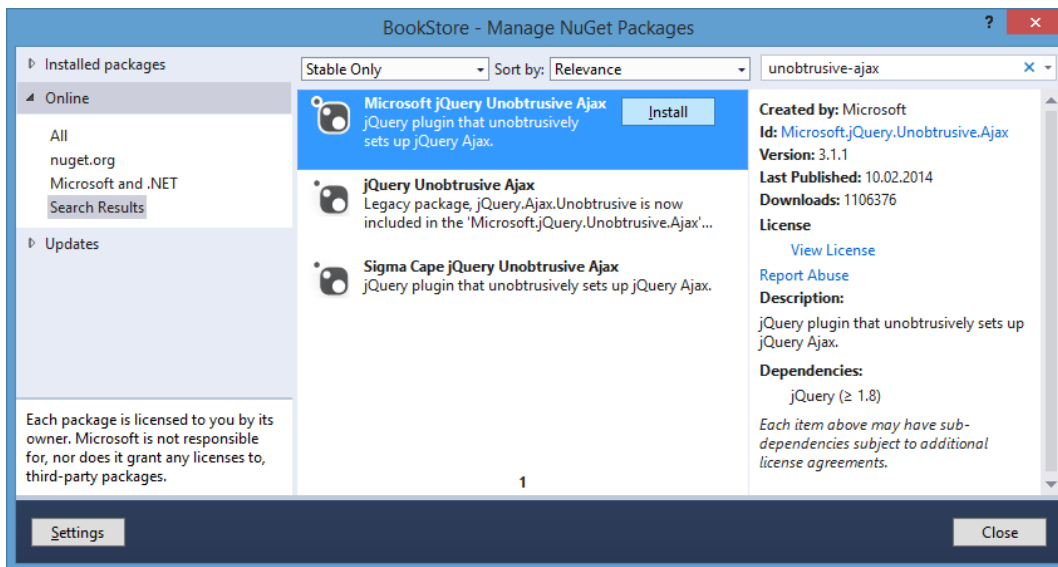


Рисунок 3.44 - Приєднання бібліотеки `jquery.unobtrusive-ajax.js`

Перший файл - загальна бібліотека jQuery. Другий файл (`jquery.unobtrusive-ajax.js`) підключає до додатка функціональність Ajax-хелперів, наприклад, Ajax-форм. Головне підключити основну бібліотеку jQuery перед іншими файлами, так як всі інші скрипти, як правило, залежать від неї.

AJAX-хелпери

Для здійснення ajax-запитів в ASP.NET MVC присутня така функціональність, як ajax-хелпери:

- Ajax.ActionLink: гіперлінк на дію контролера, після натискання на який відбувається аjax-запит до цієї дії
- Ajax.RouteLink: схожий на хелпер Ajax.ActionLink, тільки посилання створюється на певний маршрут, а не на дію контролера
- Ajax.BeginForm: створює html-форму, яка відправляє аjax-запити до певної дії певного контролера
- Ajax.BeginRouteForm: схожий на Ajax.BeginForm, тільки аjax-запити спрямовуються не до дії контролера, до по певному маршруту
- Ajax.GlobalizationScript: створює посилання на скрипт, який містить інформацію про культуру
- Ajax.JavaScriptStringEncode: кодує рядок для використання в JavaScript

Найбільш часто використовуються Ajax.ActionLink і Ajax.BeginForm, тому далі ми розглянемо ці хелпери.

3.10.3 Ajax-форми

Отже, після налаштування і підключення всіх необхідних скриптів ми можемо приступити безпосередньо до роботи з Ajax. Припустимо, у нас є клас Book, що містить дані про книгу, а в БД у нас може перебувати кілька книг одного автора. І нам треба реалізувати пошук всіх книг певного автора.

Здавалося б, навіть у даному випадку Ajax, якщо ми можемо, наприклад, у форму вводити ім'я автора і відправляти на сервер, а сервер в якості відповіді поверне нам сторінку з потрібним результатом. Але, як вище вже говорилося, AJAX допоможе нам уникнути перезавантаження всієї сторінки і виконати завантаження даних в асинхронному режимі, що безсумнівно підвищує продуктивність програми.

Для початку визначимо дію контролера, яке відповідатиме за витяг з БД потрібної інформації і передавати витягнуту інформацію в часткове представлення:

```
[HttpPost]
public ActionResult BookSearch(string name)
{
    var allbooks = db.Books.Where(a => a.Author.Contains(name)).ToList();
    if(allbooks.Count<=0)
    {
        return HttpNotFound();
    }
    return PartialView(allbooks);
}
```

Отже, дія отримує як параметр ім'я автора і по ньому здійснює пошук в БД. Тепер додамо до представлень даного контролера часткове представлення BookSearch.cshtml. оскільки часткові представлення досить зручні для роботи з AJAX:

```

@model IEnumerable<AjaxMvcApplication.Models.Book>
<div id="searchresults">
    <h3>Всі книги автора : @Model.First().Author</h3>
    <ul>
        @foreach (var item in Model)
        {
            <li>@item.Name</li>
        }
    </ul>
</div>

```

В даному випадку представлення типізується для моделі `IEnumerable<AjaxMvcApplication.Models.Book>`, яка і буде передаватися в представлення. А потім в елемент `div` виводитимуться результати пошуку у вигляді списку за умови, звичайно, якщо модель не дорівнює `null`. Тепер перейдемо до самого представлення, яке і буде відображатися користувачеві:

```

@{
    ViewBag.Title = "Index";
}

<div>
@using (Ajax.BeginForm("BookSearch", new AjaxOptions { UpdateTargetId = "results"}))
{
    <input type="text" name="name" />
    <input type="submit" value="Пошук" />
}
<div id="results"></div>
</div>
@Scripts.Render("~/scripts/jquery-1.10.2.min.js")
@Scripts.Render("~/scripts/jquery.unobtrusive-ajax.min.js")

```

Хелпер `Ajax.BeginForm` схожий на хелпер `Html.BeginForm` - він також створює елемент `form`, який використовується для відправки запиту на сервер. Перший параметр приймає ім'я дії, до якого буде звернуто запит. В даному випадку це створене вище дія `BookSearch`, яка повертає часткове представлення з даними. Якщо дія знаходиться не в поточному контролері, а в іншому, ми також можемо вказати ім'я контролера: `Ajax.BeginForm ("BookSearch", "Home", new AjaxOptions`

Другий параметр більш цікавий. Він представляє об'єкт `AjaxOptions`, який впливає на відображення результатів. Він приймає ряд параметрів, з яких ми в даному випадку використовували тільки `UpdateTargetId`. Цей параметр вказує, що у нас при отриманні результатів від сервера на сторінці буде оновлюватися елемент з `id results`: в нього буде завантажуватися `html`-код часткового представлення. Тепер ми можемо запустити додаток і здійснити пошук за допомогою `AJAX`.

Зверніть увагу на створювану розмітку для даної форми:

```

<form action="/Home/BookSearch" data-ajax="true" data-ajax-mode="replace" data-ajax-
update="#results" id="form0" method="post">
    <input type="text" name="name" />
    <input type="submit" value="Поиск" />
</form>
<div id="results"></div>

```

Створюється як і у випадку з хелперів `Html.BeginForm` елемент `form`. Але тепер в спеціальному атрибуті вказується, що це Ajax-форма: `data-ajax = "true"`. Інші параметри форми є передачею в `html` параметрів об'єкта `AjaxOptions`.

При використанні ajax-запитів треба враховувати ще одну невелику деталь: в нашому випадку при зверненні до дії `BookSearch` в блок `id = "results"` буде завантажуватися часткове представлення. Але якби в даному випадку ми використовували не `POST`, а `GET`-запити Ajax, то ми могли б і напяму звернутися до цього методу в адресному рядку браузера, написавши, наприклад, запит `Home/BookSearch`. Однак не завжди це бажано. І ми можемо закрити метод від прямого звертання в браузері, використавши атрибут `ChildActionOnly`:

```

[ChildActionOnly]
public ActionResult BookSearch(string name)
{
    // решта вмісту методу
}

```

3.10.4 Параметри об'єкту `AjaxOptions`

Вище ми використовували тільки один параметр об'єкта `AjaxOptions`. Але він має ще ряд параметрів, які дозволяють налаштувати відображення результатів запиту:

- `Confirm` - налаштовує повідомлення про підтвердження відправлення запиту на сервер. Якщо користувач не підтвердить, то запит не буде відправлено
- `HttpMethod` - встановлює метод (`Get` або `Post`), за допомогою якого виконується запит
- `InsertionMode` - встановлює, як отримані результати будуть відображатися на сторінці. Може приймати одне з трьох значень перерахування `InsertionMode`: `InsertAfter`, `InsertBefore` і `Replace` (за замовчуванням).
- `LoadingElementId` - встановлює `id` елемента `html`-сторінки, який буде відображатися під час запиту. Зазвичай це якась анімація, яка дає знати, що деяка робота виконується у фоновому режимі
- `LoadingElementDuration` - встановлює кількість мілісекунд, через яке з'явиться елемент, зазначений у параметрі `LoadingElementId`
- `OnBegin` - задає зворотний виклик перед відправкою запиту. Співвідноситься з подією `beforeSend` бібліотеки `jQuery`

- **OnComplete** - задає зворотний виклик, який викликається після виконання запиту (як вдалого, так і невдалого). Співвідноситься з подією **complete** бібліотеки **jQuery**
- **OnFailure** - задає зворотний виклик, який викликається після невдалого виконання запиту. Співвідноситься з подією **error** бібліотеки **jQuery**
- **OnSuccess** - задає зворотний виклик, який викликається після вдалого виконання запиту. Співвідноситься з подією **success** бібліотеки **jQuery**
- **UpdateTargetId** - вказує на **id** елемента, в якому будуть виводитися результати запиту
- **Url** - встановлює адресу **Url** сервера, на який відправляється запит. Встановивши дану властивість, можна не використовувати назву ім'я контролера і його дія в якості параметрів **Ajax.BeginForm**

LoadingElementId i LoadingElementDuration

Ці параметри використовуються для створення якоїсь анімації або візуалізації, яка сповіщає користувачів про процес запиту. Якщо запит іде довго, то користувач може довго не отримувати результати запиту і може подумати, що сторінка зависла. Щоб користувач бачив, що процес виконується, і потрібно налаштовувати дані параметри.

```
@using (Ajax.BeginForm("BookSearch", new AjaxOptions
    { UpdateTargetId = "results",
      LoadingElementId="loading",
      LoadingElementDuration=1000}))
{
    <input type="text" name="name" />
    <input type="submit" value="Поиск" />
}
<div id="loading" style="display:none; color:Red; font-weight:bold;">
    <p>Іде завантаження...</p>
</div>
<div id="results"></div>
```

Даний елемент відображається на сторінці тільки під час запиту, тому для нього встановлюється атрибут **display: none**. І після виконання запиту даний елемент знову ховається. Дуже популярний спосіб індикації процесу за допомогою картинки:

```
@using (Ajax.BeginForm("BookSearch", new AjaxOptions {
    UpdateTargetId = "results", LoadingElementId="loading", LoadingElementDuration=1000}))
{
    <input type="text" name="name" />
    <input type="submit" value="Пошук" />
}

```

```
<div id="results"></div>
```

Confirm

Цей параметр налаштовує повідомлення про підтвердження відправлення запиту на сервер. Якщо користувач не підтвердить, то запит не буде відправлено.

```
@using (Ajax.BeginForm("BookSearch", new AjaxOptions
    { UpdateTargetId = "results",
      Confirm="Виконати AJAX-запит?")))
{
    <input type="text" name="name" />
    <input type="submit" value="Пошук" />
}
<div id="results"></div>
```

InsertionMode

Цей параметр вказує на спосіб вставки результатів запиту на сторінку. Він приймає одне з трьох значень однойменного перерахування InsertionMode: InsertAfter (після решти результатів), InsertBefore (перед іншими результатами) і Replace (повністю заміщає результати попереднього AJAX-запиту - використовується за замовчуванням).

Зворотні виклики

Параметри OnBegin, OnComplete, OnFailure і OnSuccess допомагають задати зворотні виклики, які будуть виконуватися перед запитом, або у разі вдалого чи невдалого виконання запиту. Щоб їх використовувати, визначимо відповідні обробники javascript в представленні і вкажемо їх в параметрах об'єкта AjaxOptions:

```
<script type="text/javascript">
    function OnBegin() {
        alert("Відбувається відправлення запиту");
    }
    function OnSuccess(data) {
        alert("Запит був успішно виконаний. Одержані наступні дані: \n" + data);
    }
    function OnFailure(request, error) {
        $("#results").html("Книги вказаного автора не містяться в базі даних.");
    }
    function OnComplete(request, status) {
        alert("Статус запиту : " + status);
    }
</script>
```

```
@using (Ajax.BeginForm("BookSearch", new AjaxOptions
```

```

{
    UpdateTargetId = "results",
    OnBegin = "OnBegin",
    OnFailure = "OnFailure",
    OnSuccess = "OnSuccess",
    OnComplete = "OnComplete"
}))
{
    <input type="text" name="name" />
    <input type="submit" value="Пошуку" />
}

<div id="results"></div>

```

Таким чином, об'єкт `AjaxOptions` дозволяє налаштувати різні параметри аях-запиту.

3.10.5 AJAX-посилання

Іншим часто використовуваним AJAX-хелперами є `Ajax.ActionLink`. Він багато в чому схожий на хелпер `Ajax.BeginForm` за тим винятком, що генерує спеціальні аях-посилання. Розглянемо на прикладі. Додамо в представлення наступну розмітку:

```

<div id="bestbook">
    @Ajax.ActionLink("Найкраща книга", "BestBook",
        new AjaxOptions{ UpdateTargetId="bestbook"})
</div>

```

У даному випадку ми також використовуємо об'єкт `AjaxOptions` для установки параметрів асинхронного посилання, який приймає всі ті ж параметри, що і у випадку з аях-формою.

Ми поміщаємо посилання в елемент з `id`, який вказується в параметрі `UpdateTargetId`. Таким чином, результат запиту буде заміщати посилання. Але для роботи нам також необхідно, як і у випадку з аях-формою, дія контролера і викликаного їм часткове представлення.

Метод контролера, в даному випадку `BestBook`, буде просто повертати першу книгу:

```

public ActionResult BestBook()
{
    Book book = db.Books.First();
    return PartialView(book);
}

```

А часткове представлення, яке викликається `BestBook.cshtml` буде виводити інформацію про цю книгу:

```
@model AjaxMvcApplication.Models.Book
<h3>Найкраща книга</h3>
@Html.DisplayForModel()
```

AJAX-посилання буде відображена при запуску як звичайне посилання, а після натискання вона буде замінена отриманим від сервера контентом. Як приклад можна також змінити код представлення з попереднього розділу про аїах-форми, застосувавши аїах-посилання:

```
@using (Ajax.BeginForm("BookSearch", new AjaxOptions { UpdateTargetId = "results" }))
{
    <input type="text" name="name" />
    <input type="submit" value="Поиск" />
}

<div id="results"></div>
<br />
@foreach (string s in ViewBag.Authors)
{
    <div style="margin-right:8px;float:left;">
        @Ajax.ActionLink(s, "BookSearch", new { name=s }, new AjaxOptions { UpdateTargetId="results" })
    </div>
}
@Scripts.Render("~/scripts/jquery-1.10.2.min.js")
@Scripts.Render("~/scripts/jquery.unobtrusive-ajax.min.js")
```

У підсумку всі автори, що передаються в представлення через ViewBag (наприклад, так: ViewBag.Authors = db.Books.Select(s=>s.Author).Distinct();), виводяться у вигляді посилань вниз сторінки. А натискання на одне з цих посилань буде мати те ж дію, що і пошук по кнопці.

3.10.6 Формат JSON

Технологія AJAX все більше замість передачі даних у форматі XML, JSON використовує формат (JavaScript Object Notation). JSON не залежить від мови програмування, він більш зручний і легше обробляється. Наприклад, об'єкт Книга у форматі JSON міг би виглядати б так: {"Ім'я": "Кобзар", "Автор": "Т. Шевченко", "Рік", "1867"}. JSON У кожний окремий об'єкт полягає у фігурні дужки і являє собою набір пар ключ-значення, розділених комами, де ключем є назва властивості об'єкта, а значенням відповідно значення цієї властивості.

Раніше ми в Аїах-формі зверталися до методу, який повертав дані у вигляді звичайної розмітки HTML. Тепер змінимо його або визначимо новий, щоб він повертав дані у форматі JSON:

```
public JsonResult JsonSearch(string name)
{
```

```

var jsondata = db.Books.Where(a => a.Author.Contains(name)).ToList<Book>();
return Json(jsondata, JsonRequestBehavior.AllowGet);
}

```

Нова дія тепер повертає об'єкт JsonResult, який приймає об'єкт з результатами запиту (в даному випадку об'єкт jsondata). Другий необов'язковий параметр представляє значення перерахування JsonRequestBehavior і може приймати два значення: AllowGet (дозволити Get-запити) і DenyGet (заборонити Get-запити). В даному випадку ми дозволяємо дії посилати результати в JSON-форматі у відповідь на запити Get. Тепер змінимо аях-форму в нашому представленні:

```

<script type="text/javascript">
function OnSuccess(data) {
    var results = $('#results'); // одержуємо потрібний елемент
    results.empty(); //очищаємо елемент
    for (var i = 0; i < data.length; i++) {
        results.append('<li>' + data[i].Name + '</li>'); // додаємо в список
    }
}
</script>

@using (Ajax.BeginForm("JsonSearch", new AjaxOptions {OnSuccess = "OnSuccess"}))
{
    <input type="text" name="name" />
    <input type="submit" value="Пошук" />
}
<br />
<div id="results"></div>
@Scripts.Render("~/scripts/jquery-1.10.2.min.js")
@Scripts.Render("~/scripts/jquery.unobtrusive-ajax.min.js")

```

Оскільки параметр UpdateTargetId не братиметься до уваги при отриманні даних в форматі JSON, тому вся логіка щодо виведення даних на сторінку закладається в зворотний виклик OnSuccess. У обробнику OnSuccess ми послідовно додаємо отримані дані в потрібний елемент html-сторінки.

3.10.7 AJAX-запити за допомогою jQuery

Крім використання AJAX-форм і AJAX-посилань ми можемо виконати запит засобами самої бібліотеки jQuery, не вдаючись до аях-хелперів. Для прикладу створимо представлення, в якому і будуть здійснюватися запити AJAX:

```

<input type="text" name="name" id="search" />
<input type="button" id="submit" value="Пошук" />
<div id="results"></div>
<script type="text/javascript">
$(document).ready(function () {
    $('#submit').click(function (e) {
        e.preventDefault();
    });
});

```

```

        var name = $('#search').val();
        name=encodeURIComponent(name);
        $('#results').load('@Url.Action("BookSearch", "Home")?name='+name)
    });
});
</script>

```

В даному випадку ми в функції jQuery приєднуємо до елемента з id = submit (а це кнопка) обробник події click. Рядком var name = \$ ('# search').val(); ми отримуємо введені в текстове поле значення. Наступною рядком обробляємо його – кодуємо пробільні символи, так як інакше, якщо введений вираз буде містити пробільні символи, то передане в дію контролера значення буде некоректне. В рядку \$ ('# results'). Load ('@ Url.Action ("Book Search", "Home")? Name =' + name) ми завантажуюмо в елемент з id = results контент, отриманий за допомогою методу load .

Метод, до якого ми звертаємося, буде отримувати рядок, здійснювати за ним пошук і передавати знайдені дані в часткове представлення:

```

public ActionResult BookSearch(string name)
{
    var allbooks = db.Books.Where(a => a.Author.Contains(name)).ToList();
    return PartialView(allbooks);
}

```

Часткове представлення, що викликається може виглядати наступним чином:

```

@model IEnumerable<AjaxMvcApplication.Models.Book>
<div id="searchresults">
    @if (Model.Count()>0)
    {
        <h3>Всі книги автора : @Model.First().Author</h3>
        <ul>
            @foreach (var item in Model)
            {
                <li>@item.Name</li>
            }
        </ul>
    }
</div>

```

Тепер, ми можемо здійснити пошук, ввівши будь-яке значення і натиснувши на кнопку пошуку.

3.10.8 Приклад. Створення чату на AJAX і jQuery.

Розглянемо більш складний приклад застосування AJAX в ASP.NET MVC, який представляє собою створення чату за допомогою jQuery. Для початку визначимо модель, яку ми будемо використовувати:

```

public class ChatModel
{
    public List<ChatUser> Users; // Всі користувачі

    public List<ChatMessage> Messages; // всі повідомлення

    public ChatModel()
    {
        Users = new List<ChatUser>();
        Messages = new List<ChatMessage>();

        Messages.Add(new ChatMessage()
        {
            Text = "Чат запущений " + DateTime.Now
        });
    }
}

public class ChatUser
{
    public string Name;
    public DateTime LoginTime;
    public DateTime LastPing;
}

public class ChatMessage
{
    // автор повідомлення, якщо null – то автор сервер
    public ChatUser User;
    // час повідомлення
    public DateTime Date = DateTime.Now;
    // текст повідомлення
    public string Text = "";
}

```

Модель чату - Chat Model використовує для зберігання повідомлень і користувачів дві додаткові моделі ChatMessage і ChatUser.

Тепер перейдемо до контролера. У проекті MVC 5 вже є контролер HomeController за замовчуванням. Змінимо його в такий спосіб:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using AjaxChat.Models;

namespace AjaxChat.Controllers
{
    public class HomeController : Controller
    {

```

```

static ChatModel chatModel;

public ActionResult Index(string user, bool? logOn, bool? logOff, string chatMessage)
{
    try
    {
        if (chatModel == null) chatModel = new ChatModel();

        //залишаємо тільки останні 90 повідомлень
        if (chatModel.Messages.Count > 100)
            chatModel.Messages.RemoveRange(0, 90);

        // якщо звичайний запит, просто повертаєм представлення
        if (!Request.IsAjaxRequest())
        {
            return View(chatModel);
        }
        // если передан параметр logOn
        else if (logOn != null && (bool)logOn)
        {
            //перевіряємо, чи існує вже такий користувач
            if (chatModel.Users.FirstOrDefault(u => u.Name == user) != null)
            {
                throw new Exception("Користувач вже існує");
            }
            else if (chatModel.Users.Count > 10)
            {
                throw new Exception("Чат заповнений");
            }
            else
            {
                // додаємо в список нового користувача
                chatModel.Users.Add(new ChatUser()
                {
                    Name = user,
                    LoginTime = DateTime.Now,
                    LastPing = DateTime.Now
                });

                // додаємо в список повідомлень повідомлення про нового користувача
                chatModel.Messages.Add(new ChatMessage()
                {
                    Text = user + " зайшов в чат",
                    Date = DateTime.Now
                });
            }

            return PartialView("ChatRoom", chatModel);
        }
        // якщо переданий параметр logOff
        else if (logOff != null && (bool)logOff)
    }
}

```



```

    {
        LogOff(chatModel.Users.FirstOrDefault(u => u.Name == user));
        return PartialView("ChatRoom", chatModel);
    }
    else
    {
        ChatUser currentUser = chatModel.Users.FirstOrDefault(u => u.Name == user);

        //для кожного користувача запам'ятовуємо час останнього оновлення
        currentUser.LastPing = DateTime.Now;

        // видаляємо неактивних користувачів, якщо час простою більше 15 сек
        List<ChatUser> toRemove = new List<ChatUser>();
        foreach (Models.ChatUser usr in chatModel.Users)
        {
            TimeSpan span = DateTime.Now - usr.LastPing;
            if (span.TotalSeconds > 15)
                toRemove.Add(usr);
        }
        foreach (ChatUser u in toRemove)
        {
            LogOff(u);
        }

        // додаємо в список повідомлень нове повідомлення
        if (!string.IsNullOrEmpty(chatMessage))
        {
            chatModel.Messages.Add(new ChatMessage()
            {
                User = currentUser,
                Text = chatMessage,
                Date = DateTime.Now
            });
        }

        return PartialView("History", chatModel);
    }
}
catch (Exception ex)
{
    //в випадку помилки надсилаємо статусний код 500
    Response.StatusCode = 500;
    return Content(ex.Message);
}
}

// при виході користувача видаляємо його зі списку
public void LogOff(ChatUser user)
{
    chatModel.Users.Remove(user);
    chatModel.Messages.Add(new ChatMessage()

```

```

    {
        Text = user.Name + " покинув чат.",
        Date = DateTime.Now
    });
}
}
}

```

Всі дані - користувачі і повідомлення про них будуть зберігатися у змінній chatModel. І так як вона буде загальною для всіх користувачів, вона оголошена статично.

Метод Index приймає чотири параметри. Перший параметр ідентифікує користувача. Параметр logOn при значенні true вказує, що користувач здійснив вхід. Параметр logOff при значенні true, навпаки, вказує, що користувач вийшов з чату. І останній параметр містить повідомлення користувача. І в залежності від значень параметрів повертаємо або звичайне представлення Index.cshtml, або одне з часткових представлень.

Якщо запит не є AJAX-запитом, передається звичайне представлення. Якщо встановлено параметр logOn або logOff, то повертаємо часткове представлення ChatRoom. І якщо встановлений останній параметр з повідомленням користувача то повертається часткове представлення History. Тепер визначимо всі ці представлення. Код головного представлення Index.cshtml буде виглядати наступним чином:

```

@model AjaxChat.Models.ChatModel
@{
    Layout = null;
}

<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Чат на Ajax и JQuery</title>
    @Styles.Render("~/Content/Site.css")
</head>
<body>
    <div id="Username">
    </div>
    <div id="LastRefresh">
    </div>
    <div id="container">
        <div id="loginBlock">
            Введіть ім'я :
            <input type="text" id="txtUserName" />
            <button id="btnLogin" value="Ввійти">Ввійти</button>
        </div>
    </div>
    <div id="Error">

```

```

</div>
@Ajax.ActionLink("Login", "Index", new { user = "" },
    new AjaxOptions { UpdateTargetId = "container", OnFailure = "LoginOnFailure", OnSuccess =
"LoginOnSuccess" },
    new { @id = "LoginButton", @style = "visibility:hidden;" })
@Scripts.Render("~/bundles/jquery")
@Scripts.Render("~/scripts/jquery.unobtrusive-ajax.min.js")
@Scripts.Render("~/Scripts/chat.js")
</body>
</html>

```

Це представлення без майстер-сторінки. Елемент div з id="Username" буде відображати нік користувача. Блок з id = "LastRefresh" призначений для відображення часу останнього оновлення чату, а div с id="Error"- для виведення помилок, які можуть виникнути.

І для власне логіна у нас є текстове поле і кнопка. Однак при введенні ніка і натисканні на кнопку дані відразу не будуть передаватися на сервер, так як це не форма. Реальна відправка даних на сервер буде відбуватися за допомогою аяк-посилання, яка буде мати id = "LoginButton":

```

@Ajax.ActionLink("Login", "Index", new { user = "" },
    new AjaxOptions { UpdateTargetId = "container", OnFailure = "LoginOnFailure", OnSuccess =
"LoginOnSuccess" },
    new { @id = "LoginButton", @style = "visibility:hidden;" })

```

При натисканні на посилання буде відбуватися оновлення блоку з id = "container". За рахунок цього буде відбуватися часткове оновлення поточної сторінки за допомогою аяк, а не звичайне перезавантаження сторінки. Але самі ми на посилання натискати не будемо. Ми його навіть бачити не будемо, так як у нього встановлено стиль "visibility: hidden;" Натискання відбуватиметься автоматично в підключеному в поданні файлі chat.js:

```

$(document).ready(function () {

    // логін
    $("#btnLogin").click(function () {
        var nickName = $("#txtUserName").val();
        if (nickName) {
            // формуємо посилання з параметрами, за якими йде звертання
            var href = "/Home?user=" + encodeURIComponent(nickName);
            href = href + "&logOn=true";
            $("#LoginButton").attr("href", href).click();

            //установка поля с ніком пользователя
            $("#Username").text(nickName);
        }
    });
});

```

```

//при успішному вході завантажуюємо повідомлення
function LoginOnSuccess(result) {

    Scroll();
    ShowLastRefresh();

    //кожні п'ять секунд оновлюємо чат
    setTimeout("Refresh()", 5000);

    //відправка повідомлень після натискання Enter
    $('#txtMessage').keydown(function (e) {
        if (e.keyCode == 13) {
            e.preventDefault();
            $('#btnMessage').click();
        }
    });

    //установка обробника натискання кнопки для відправки повідомлень
    $('#btnMessage').click(function () {
        var text = $('#txtMessage').val();
        if (text) {

            //звертаємось до методу Index і передаємо параметр "chatMessage"
            var href = "/Home?user=" + encodeURIComponent($('#Username').text());
            href = href + "&chatMessage=" + encodeURIComponent(text);
            $('#ActionLink').attr("href", href).click();
        }
    });

    //обробник кнопки виходу з чату
    $('#btnLogOff').click(function () {

        //звертаємось до методу Index и передаємо параметр "logOff"
        var href = "/Home?user=" + encodeURIComponent($('#Username').text());
        href = href + "&logOff=true";
        $('#ActionLink').attr("href", href).click();
        document.location.href = "Home";
    });
}

//при помилці відображаєм повідомлення про помилку при вході в систему
function LoginOnFailure(result) {
    $('#Username').val("");
    $('#Error').text(result.responseText);
    setTimeout("$('#Error').empty();", 2000);
}

// кожні 5 секунд оновлюємо поле чата
function Refresh() {
    var href = "/Home?user=" + encodeURIComponent($('#Username').text());
    $('#ActionLink').attr("href", href).click();
}

```

```

        setTimeout("Refresh();", 5000);
    }

    //Відображаємо повідомлення про помилку
    function ChatOnFailure(result) {
        $('#Error').text(result.responseText);
        setTimeout("$('#Error').empty();", 2000);
    }

    // при успішному одержанні відповіді з серверу
    function ChatOnSuccess(result) {
        Scroll();
        ShowLastRefresh();
    }

    //скрол до низу вікна
    function Scroll() {
        var win = $('#Messages');
        var height = win[0].scrollHeight;
        win.scrollTop(height);
    }

```

```

    //відображення часу останнього оновлення чату
    function ShowLastRefresh() {
        var dt = new Date();
        var time = dt.getHours() + ":" + dt.getMinutes() + ":" + dt.getSeconds();
        $('#LastRefresh').text("Останнє оновлення було в " + time);
    }

```

Для надсилання даних за логіном і натиснути на посилання, в обробнику натиснення ми прописуємо наступні рядки:

```

// формуємо посилання з параметрами, за якими йде звернення
var href = "/Home?user=" + encodeURIComponent(nickName);
href = href + "&logOn=true";
// автоклік по ajax-ссылке
$('#LoginButton').attr("href", href).click();

```

Оскільки в запиті до сервера передається параметр `logIn = true`, то метод `Index` при обробці цього запиту буде відправляти клієнту часткове представлення `ChatRoom`:

`return PartialView("ChatRoom", chatModel);`. Створимо це представлення:

```

@using AjaxChat.Models
@model ChatModel
<div id="RefreshArea">
@{
    Html.RenderPartial("History", Model);
}
</div>

```

```
@Ajax.ActionLink("ActionLink", "Index", new { user = "", logOn = "", logOff = "", chatMessage = "" },
new AjaxOptions { UpdateTargetId = "RefreshArea", OnSuccess = "ChatOnSuccess", OnFailure =
"ChatOnFailure" },
new { @id = "ActionLink", @style = "visibility:hidden;" })
```

```
<div id="inputForm">
  <table>
    <tr>
      <td rowspan="2"><textarea id="txtMessage" rows="3"></textarea></td>
      <td><button id="btnMessage" value="Відправити" > Відправити </button></td>
    </tr>
    <tr>
      <td><button id="btnLogOff" value="Вийти">Вийти</button></td>
    </tr>
  </table>
</div>
```

Це часткове представлення, типізоване моделлю Chat Model, яка передається з методу Index. Воно містить поля для введення і відправки повідомлень, а також кнопку виходу з чату.

Крім того, тут є блок `div id = "RefreshArea"`, призначений безпосередньо для виведення повідомлень і користувачів. Для рендеринга цієї інформації, в ньому існує звернення до часткового представлення History, яке ми зараз створимо. Також тут є невидиме аїах-посилання, оновлююче блок `div id = "RefreshArea"`:

```
@Ajax.ActionLink("ActionLink", "Index", new { user = "", logOn = "", logOff = "", chatMessage = "" },
new AjaxOptions { UpdateTargetId = "RefreshArea", OnSuccess = "ChatOnSuccess", OnFailure =
"ChatOnFailure" },
new { @id = "ActionLink", @style = "visibility:hidden;" })
```

У вищерозглянутому javascript файлі при вдалому логіні викликається функція `LoginOnSuccess`, яка в свою чергу викликає функцію `Refresh ()`:

```
function Refresh() {
  var href = "/Home?user=" + encodeURIComponent($("#Username").text());

  $("#ActionLink").attr("href", href).click();
  setTimeout("Refresh()", 5000);
}
```

Ця функція знову формує рядок запиту з параметрами для аїах-посилання і виробляє автоклік на неї. В результаті блок `div id = "RefreshArea"` буде оновлюватися новими даними кожні п'ять секунд. Подібна дія відбуватиметься також і при натисканні на кнопку відправки повідомлення в чат:

```
var href = "/Home?user=" + encodeURIComponent($("#Username").text());
href = href + "&chatMessage=" + encodeURIComponent(text);
$("#ActionLink").attr("href", href).click();
```

Оскільки в обох випадках при передачі параметрів в метод Index контролера параметри logOn і logOff будуть не встановлені, то метод Index у відповідь буде повертати часткове представлення History.cshtml: return PartialView ("History", chatModel);

Тепер створимо саме часткове представлення History.cshtml:

```
@using AjaxChat.Models
@model ChatModel
<div id="Users">
    <p><b>В чаті онлайн: @Model.Users.Count</b></p>
    @foreach (ChatUser user in Model.Users)
    {
        <p>@user.Name</p>
    }
</div>
<div id="Messages">
    @foreach (ChatMessage msg in Model.Messages)
    {
        <p>
            @msg.Date<br />
            @if (msg.User != null)
            {
                <b>@(msg.User.Name + ":")</b> @msg.Text
            }
            else
            {
                <span class="serverMes">@msg.Text</span>
            }
        </p>
    }
</div>
```

Таким чином нами було реалізовано чат з використанням засобів C# та MVC5.

Контрольні питання.

1. Особливості явної прив'язки моделей.
2. Особливості вибіркової прив'язки моделей.
3. Робота постачальника значень.

3.11 OWIN і Katana

Одним з нововведень в .NET 4.5.1 стало впровадження специфікації OWIN і проекту Katana. Завдяки цим нововведенням стають доступними нові способи побудови веб-додатків. Розглянемо, що вони представляють. OWIN або Open Web Interface for .NET являє собою специфікацію, визначальну взаємодію між веб-додатком і веб-сервером. Сучасним веб-додатків потрібно більше гнучкості.

І відповіддю на ці виклики стала поява специфікації OWIN. Але OWIN - це специфікація, яка створювалася як відкритий проект поза стінами Microsoft. Безпосередньою реалізацією цієї специфікації став проект Katana.

Всю архітектуру проекту Katana можна представити таким чином:

Програма (додаток)
Проміжне програмне забезпечення (Middleware)
Сервер
Хост

Розглянемо основні шари цієї архітектури:

- Хост: процес, який обробляє додаток. Якщо в якості сервера додатки використовується IIS, то подібним процесом є процес самого сервера IIS. Однак їм може бути і сам файл програми. Основне його завдання - запуск, завантаження компонентів OWIN і коректне завершення.
- Сервер: реалізує специфікацію OWIN, виконує наповнення словника оточення з даних запиту і обробляє запит на конвеєрі OWIN.
- Проміжне ПЗ або шар Middleware: набір компонентів, що обробляють запити в конвеєрі OWIN.
- Додаток, що виконує певне завдання.

Що нам дає використання Katana в своїх проектах? Katana дозволяє створювати додаток, що використовує різні веб-технології, і запускати його де завгодно. Подивимося на прикладі. У Visual Studio 2013 створимо новий додаток. У нашому випадку воно буде називатися KatanaApp. Як шаблон додатку виберемо шаблон Empty:

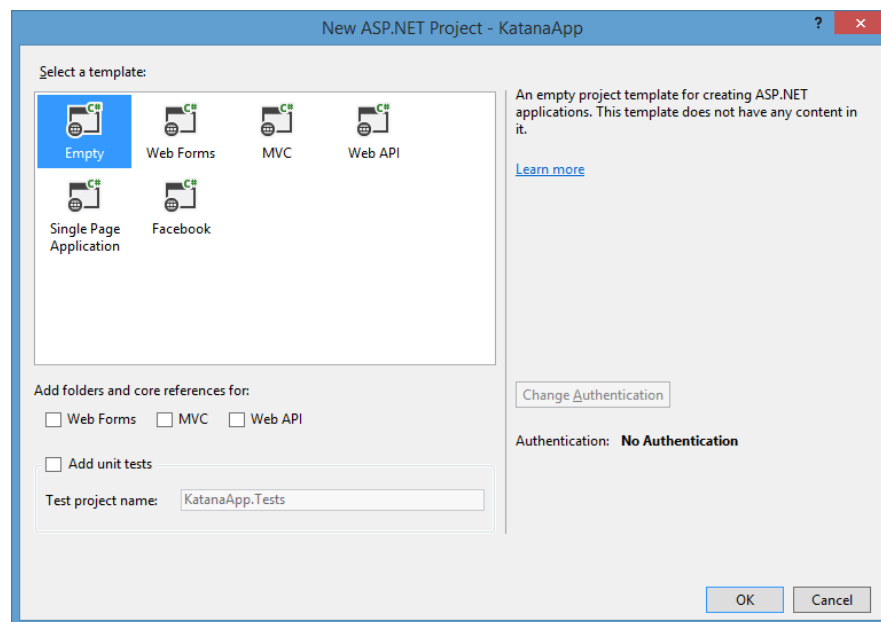


Рисунок 3.45 - Створення проекту

Отже, Visual Studio створює абсолютно порожній проект. Функціонал інфраструктури OWIN міститься в бібліотеці Microsoft.Owin.Host.SystemWeb. Для додавання бібліотеки в проект скористаємося менеджером NuGet:

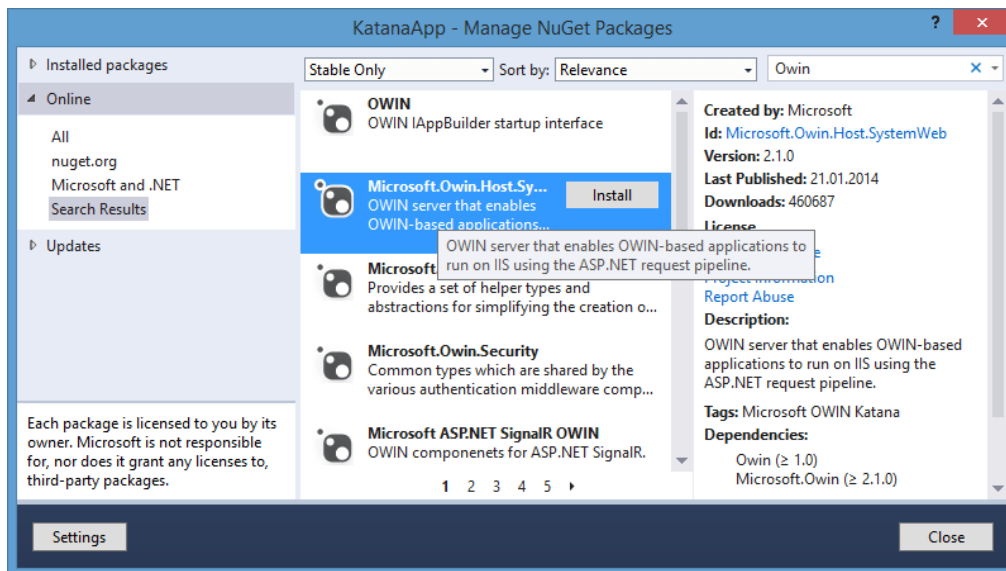


Рисунок 3.46 - Додавання системи OWIN

Тепер додамо в проект клас запуску OWIN. Всі додатки OWIN повинні мати подібний клас, в якому визначаються компоненти конвеєра додатку. Є різні способи визначення даного класу. Ми скористаємося найпростішим - просто додамо в проект відповідного класу. Для цього натиснемо правою кнопкою миші на проект у вікні Solution Explorer, в меню виберемо Add-> New Item ... і у вікні додавання нового компонента виберемо OWIN Startup class:

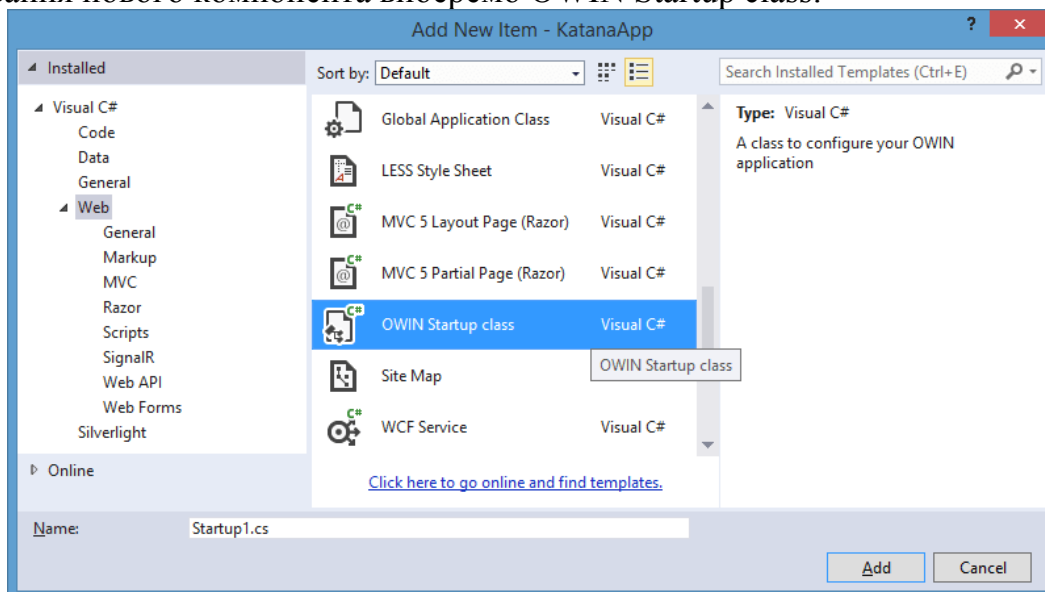


Рисунок 3.47 - Додавання нового класу

Тепер змінимо доданий клас наступним чином:

```
using System;
using System.Threading.Tasks;
using Microsoft.Owin;
using Owin;
```

```
[assembly: OwinStartup(typeof(KatanaApp.Startup1))]
```

```

namespace KatanaApp
{
    public class Startup1
    {
        public void Configuration(IAppBuilder app)
        {
            app.Run(context =>
            {
                context.Response.ContentType = "text/html; charset=utf-8";
                return context.Response.WriteAsync("<h2>Hello world!</h2>");
            });
        }
    }
}

```

По-перше, щоб додаток знав, де у нас знаходиться клас Owin Startup, ми використовуємо атрибут [assembly: OwinStartup (typeof (KatanaApp.Startup1))] і тепер він знає, що таким класом є KatanaApp.Startup1.

Цей клас містить один метод Configuration, який в якості параметра приймає інтерфейс IAppBuilder. Цей інтерфейс і є тим проміжним шаром middleware, який бере участь в обробці запитів. Для обробки запиту об'єкт IAppBuilder виконує функцію Run, яка має в якості параметра делегат Func<IOwinContext, Task>. За допомогою об'єкта context.Response створюємо відповідь і посилаємо його клієнту.

В даному випадку в якості веб-сервера виступив IIS, він же виступає в якості хоста. Однак однією з сторін проекту Katana і OWIN є те, що додаток сам буде виступати і в якості хоста, і в якості сервера. В основі такої організації лежить клас HttpListener, який може прослуховувати вхідні http-запити і відповідати на них. Тобто фактично може виступати в якості веб-сервера. І хоча ми більшою мірою говоримо про веб-додатки в рамках ASP.NET MVC, але щоб подивитися переваги OWIN, створимо простий консольний додаток в Visual Studio, наприклад, в Visual Studio 2013 for Desktop.

Спочатку додамо через NuGet бібліотеку Microsoft.Owin.SelfHost:

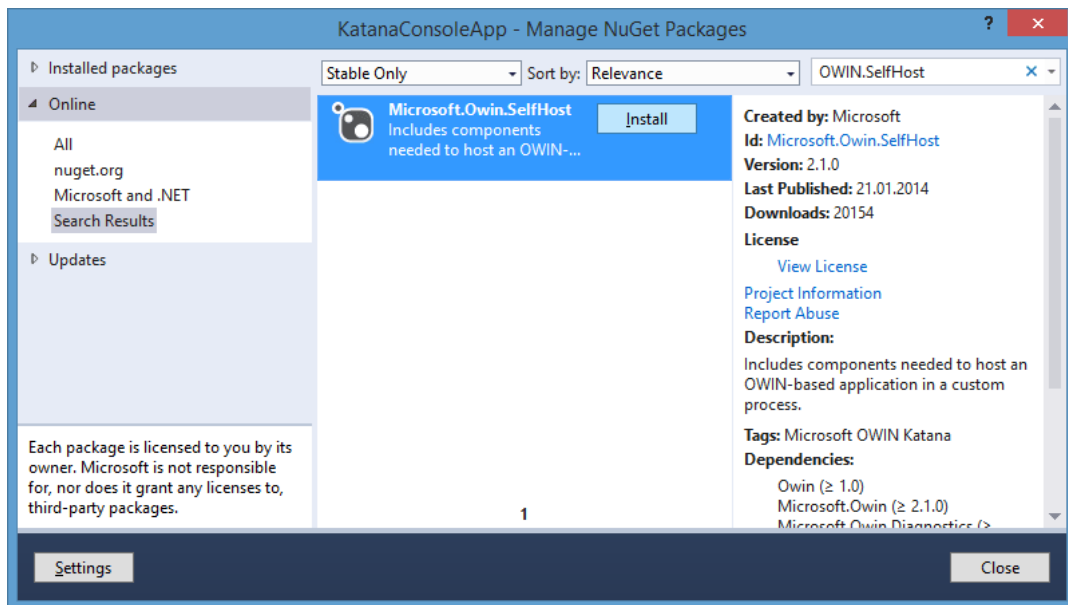


Рисунок 3.48 - Менеджер пакетів

Тепер додамо в консольний додаток певний клас Startup1 (не забуваючи змінити простір імен класу на простір імен нашого застосування). І змінимо код класу Program наступним чином:

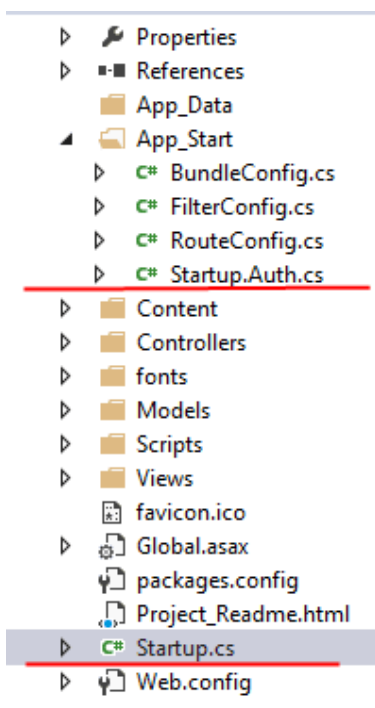
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace KatanaConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            using (Microsoft.Owin.Hosting.WebApp.Start<Startup1>("http://localhost:8000"))
            {
                Console.WriteLine("Сервер запущений. Натисніть будь-яку клавішу для завершення роботи...");
                Console.ReadLine();
            }
        }
    }
}
```

Тепер ми можемо запустити додаток. Він буде фактично працювати як веб-сервер на 8000 порті, і в браузері ми можемо звернутися до нього, набравши в адресному рядку `http://localhost: 8000`.

3.11.1 OWIN і Katana в проектах MVC 5

Ми розглянули загальну концепцію проектів OWIN і Katana, але як же вони застосовуються в MVC5? При створенні проекту ASP.NET MVC5, що використовує ідентифікацію користувачів, ми зможемо помітити застосування OWIN і Katana в додатку:



По-перше, тому що проект використовує Katana, то в ньому є файл Startup.cs з однойменним класом:

```
using Microsoft.Owin;  
using Owin;  
[assembly: OwinStartupAttribute(typeof(IdentityMvc.Startup))]  
namespace IdentityMvc  
{  
    public partial class Startup  
    {  
        public void Configuration(IAppBuilder app)  
        {  
            ConfigureAuth(app);  
        }  
    }  
}
```

Цей такий же клас, що ми розглядали раніше, тільки в даному випадку він частковий (partial) і викликає метод ConfigureAuth. Цей метод визначено в іншому файлі - в файлі Startup.Auth.cs, який знаходиться в папці App_Start. У цьому файлі також знаходиться визначення часткового класу Startup:

```

using Microsoft.AspNet.Identity;
using Microsoft.Owin;
using Microsoft.Owin.Security.Cookies;
using Owin;

namespace IdentityMvc
{
    public partial class Startup
    {
        public void ConfigureAuth(IAppBuilder app)
        {
            app.UseCookieAuthentication(new CookieAuthenticationOptions
            {
                AuthenticationType = DefaultAuthenticationTypes.ApplicationCookie,
                LoginPath = new PathString("/Account/Login")
            });
            app.UseExternalSignInCookie(DefaultAuthenticationTypes.ExternalCookie);

            //app.UseMicrosoftAccountAuthentication(
            //    clientId: "",
            //    clientSecret: "");

            //app.UseTwitterAuthentication(
            //    consumerKey: "",
            //    consumerSecret: "");

            //app.UseFacebookAuthentication(
            //    appId: "",
            //    appSecret: "");

            //app.UseGoogleAuthentication();
        }
    }
}

```

У методі `ConfigureAuth` проводиться аутентифікація за допомогою куків. В закоментованих рядках ви можете побачити, що за допомогою OWIN проводиться аутентифікація за допомогою зовнішніх сервісів, таких як facebook, twitter та ін. Крім того, як ми далі побачимо, вся базова система аутентифікації і авторизації AspNet Identity базується на компонентах OWIN і Katana.

Контрольні питання.

1. Особливості ідентифікації за допомогою зовнішніх сервісів.
2. Призначення компонентів OWIN і Katana.

3.12 Авторизація та аутентифікація в MVC 5

3.12.1 ASP.NET Identity

Реліз ASP.NET MVC5 ознаменувався виходом нової системи авторизації і аутентифікації в .NET додатках під назвою ASP.NET Identity. Ця система прийшла на зміну провайдерам Simple Membership, які були введені в ASP.NET MVC 4.

Розглянемо систему авторизації та аутентифікації ASP.NET Identity на прикладі додатка MVC5. Отже, при створенні програми MVC 5 Visual Studio пропонує нам вибрати один з типів аутентифікації:

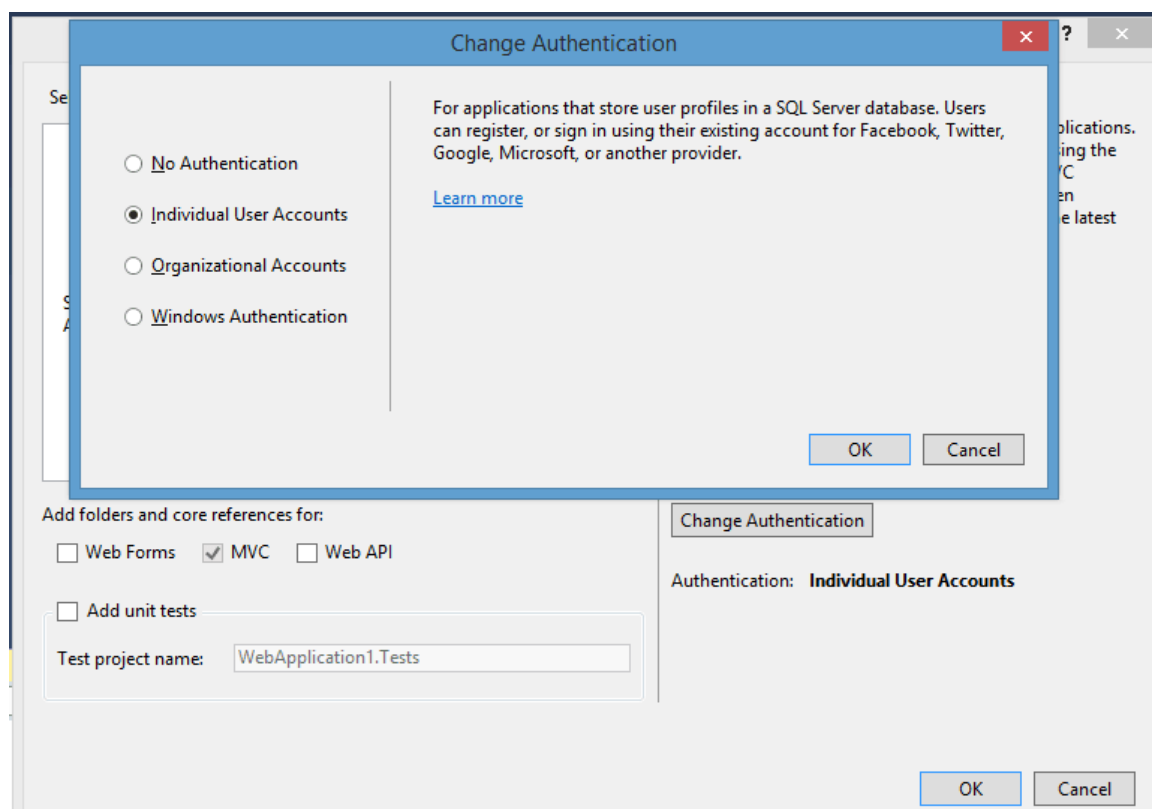
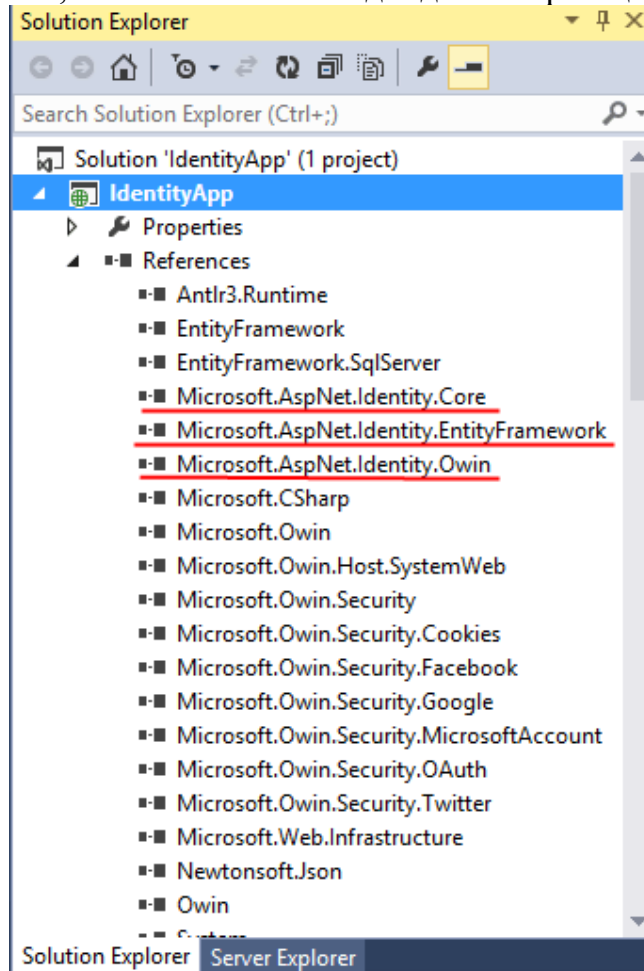


Рисунок 3.49 - Зміна аутентифікації

Натиснувши на кнопку Change Authentication, ми можемо змінити тип аутентифікації, вибравши одне з наступних:

- No Authentication: ASP.NET Identity і вбудована система аутентифікації відсутня
- Individual User Accounts: проект за замовчуванням включає систему ASP.NET Identity, яка дозволяє авторизувати як користувачів всередині програми, так і за допомогою зовнішніх сервісів, як google, твіттер і т.д.
- Organizational Accounts: підходить для сайтів і веб-додатків окремих компаній і організацій
- Windows Authentication: система аутентифікації для мереж intranet за допомогою облікових записів Windows

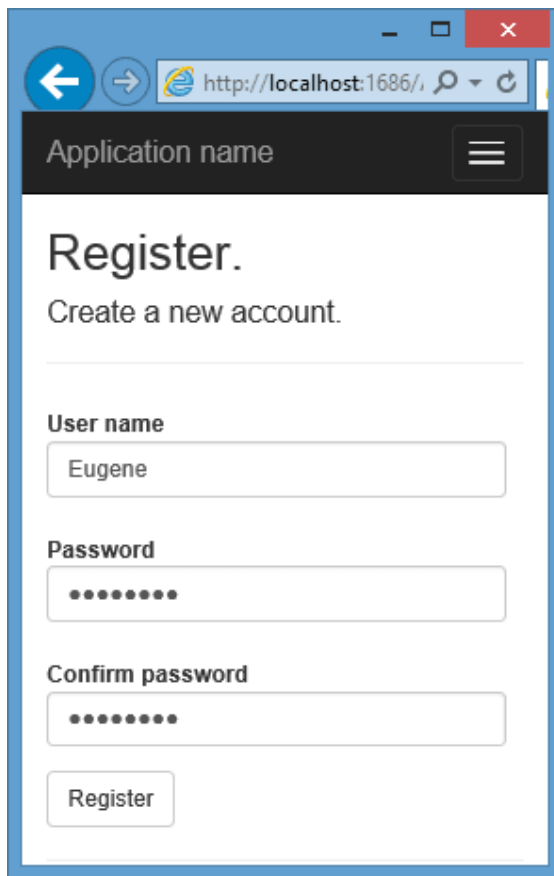
Залишимо значення за замовчуванням, тобто Individual User Accounts і створимо проект. Створений проект вже за замовчуванням має всю необхідну для авторизації інфраструктуру: моделі, контролери, представлення. Якщо ми заглянемо в вузол References (Бібліотеки), то побачимо там ряд ключових бібліотек, які і містить необхідні для авторизації і аутентифікації класи:



Це ряд бібліотек OWIN, які додають функціональність OWIN в проект, а також три бібліотеки власне ASP.NET Identity:

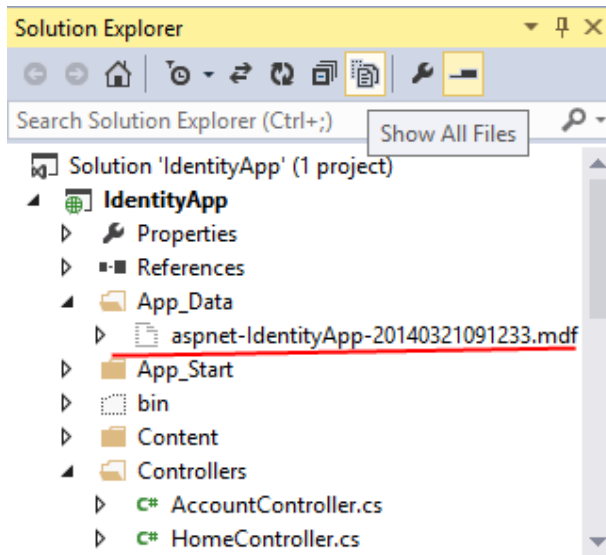
- Microsoft.AspNet.Identity.EntityFramework: містить класи Entity Framework, які застосовують ASP.NET Identity і здійснюють зв'язок з SQL Serverом
- Microsoft.AspNet.Identity.Core: містить ряд ключових інтерфейсів ASP.NET Identity. Реалізація цих інтерфейсів дозволить вийти за рамки MS SQL Server і використовувати в якості сховища облікових записів інші СУБД, в тому числі системи NoSQL
- Microsoft.AspNet.Identity.OWIN: привносить в додаток ASP.NET MVC аутентифікацію OWIN за допомогою ASP.NET Identity

Оскільки вся інфраструктура вже є в проекті, запустимо проект, на сторінці що відображається в браузері знайдемо посилання Register і натиснемо на неї. На сторінці реєстрації введемо деякі дані:

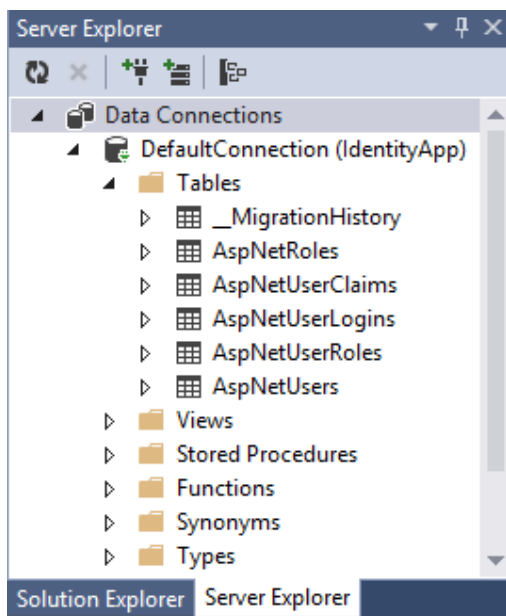


Після реєстрації логін буде відображатися в правому верхньому куті веб-сторінки веб-додатку. Здійснивши реєстрацію, ми можемо разлогініться, натиснувши на LogOff, і знову увійти в систему. Таким чином, ми можемо вже почати користуватися вбудованою системою аутентифікації в додатку ASP.NET MVC 5. Тепер же розглянемо її основні моменти.

По-перше, де це все зберігається? Куди потрапляють дані зареєстрованих користувачів? В даному випадку використовується підхід Code First. У файлі web.config вже є рядок підключення за замовчуванням, яка задає каталог бази даних. Якщо ми розкриємо папку App_Data, то зможемо побачити створену базу даних:



Якщо раптом у папці база даних не видно, натиснемо вгорі вікна Solution Explorer на кнопку Show All Files (Показати всі файли). Ми можемо відкрити цю базу даних у вікні Server Explorer і побачити її вміст:



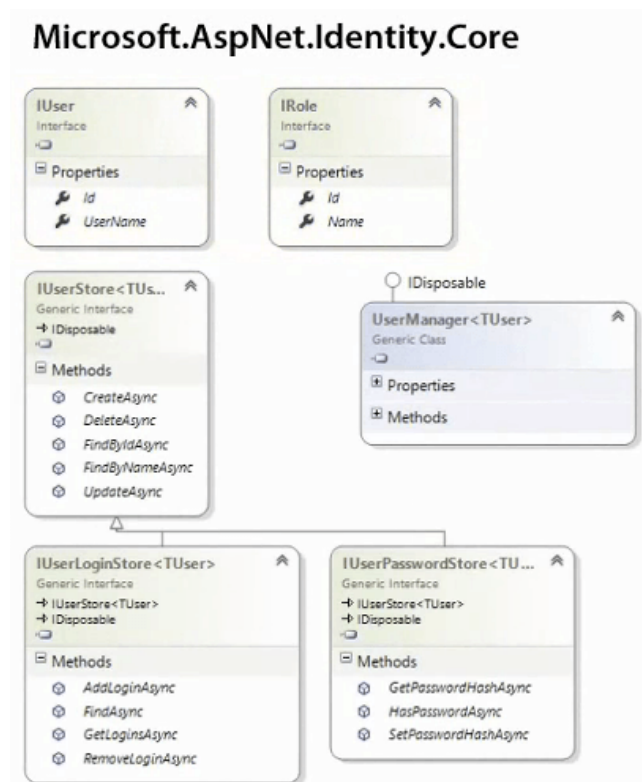
За замовчуванням при реєстрації першого користувача створюється наступний набір таблиць:

- **_MigrationHistory**: Використовується EntityFramework для міграцій БД
- **AspNetRoles**: містить визначення ролей
- **AspNetUserClaims**: таблиця, що зберігає набір клейм (claim). Claim представляє іншу модель авторизації в порівнянні з ролями. Грубо кажучи, claim містить деяку інформацію про користувача, наприклад, адреса електронної пошти, логін, вік і т.д. І ця інформація дозволяє

ідентифікувати користувача і наділити його відповідними правами доступу.

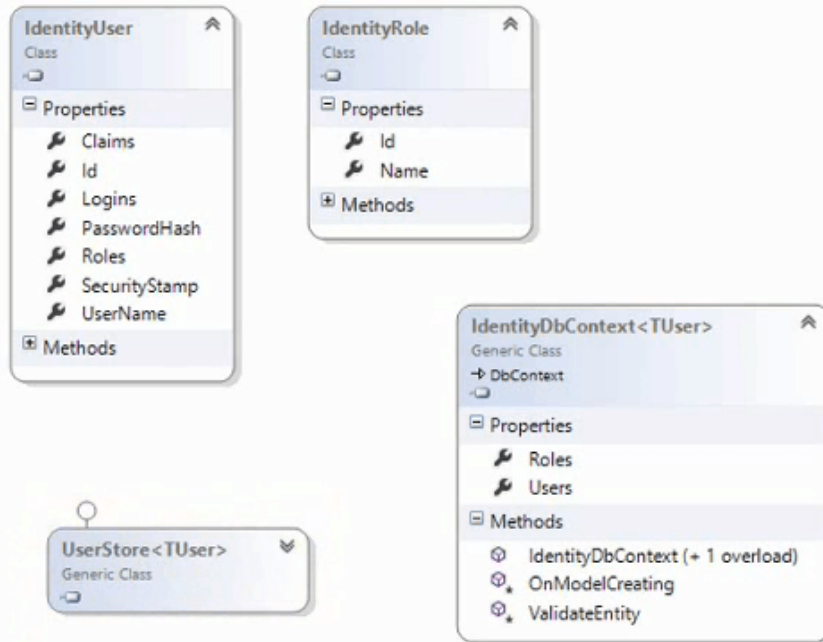
- AspNetUserLogins: таблиця логінів користувача
- AspNetUserRoles: таблиця, що встановлює для користувачів певні ролі
- AspNetUsers: власне таблиця користувачів. Якщо ми її відкріємо, то побачимо дані зареєстрованого користувача

Ключовими об'єктами в AspNet Identity є користувачі та ролі. Вся функціональність по створенню, видаленню користувачів, взаємодії зі сховищем користувачів зберігається в класі UserManager. Для роботи з ролями і їх управлінням в AspNet Identity визначено клас RoleManager. Класи UserManager і RoleManager знаходяться в бібліотеці Microsoft.AspNet.Identity.Core.



Кожен користувач для UserManager представляє об'єкт інтерфейсу IUser. А всі операції по управлінню користувачами виробляються через сховище, представлене об'єктом IUserStore. Кожна роль представляє реалізацію інтерфейсу IRole, а управління ролями класом RoleManager відбувається через сховище IRoleStore. Безпосередню реалізацію інтерфейсів IUser, IRole, IUserStore і IRoleStore надає простір імен Microsoft.AspNet.Identity.EntityFramework:

Microsoft.AspNet.Identity.EntityFramework



Клас IdentityUser є реалізацією інтерфейсу IUser. А клас сховища користувачів - UserStore реалізує інтерфейс IUserStore. Подібним чином клас IdentityRole реалізує інтерфейс IRole, а клас сховища ролей - RoleStore реалізує інтерфейс IRoleStore. А для взаємодії з базою даних в просторі імен Microsoft.AspNet.Identity.EntityFramework визначено клас контексту IdentityDbContext. У додатку ASP.NET MVC ми не будемо працювати безпосередньо з класами IdentityUser і IdentityDbContext. За замовчуванням в проект в папку Models додається файл IdentityModels.cs, який містить визначення класів користувачів і контексту даних:

```

public class ApplicationUser : IdentityUser
{
    public async Task<ClaimsIdentity> GenerateUserIdentityAsync(
        UserManager<ApplicationUser> manager)
    {
        var userIdentity = await manager.CreateIdentityAsync(this,
            DefaultAuthenticationTypes.ApplicationCookie);
        return userIdentity;
    }
}

public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
{
    public ApplicationDbContext()
        : base("DefaultConnection", throwIfV1Schema: false)
    {
    }

    public static ApplicationDbContext Create()
    {
    }
}

```

```

{
    return new ApplicationDbContext();
}
}

```

У додатку ми не працюємо безпосередньо з класами `IdentityUser` і `IdentityDbContext`, а маємо справу з класами-спадкоємцями. Клас `ApplicationUser` успадковує від `IdentityUser` все властивості. І крім того додає метод `GenerateUserIdentityAsync()`, в якому за допомогою виклику `UserManager.CreateIdentityAsync` створюється об'єкт `ClaimsIdentity`. Даний об'єкт містить інформацію про даного користувача.

Подібна архітектура дозволяє взяти вже готовий функціонал і при необхідності додати новий, наприклад, додати для користувача нову властивість або додати нову таблицю в БД. Не має потреби детально розписувати весь функціонал `AspNet Identity`, який за замовчуванням додається в проект, розглянемо коротко лише основні можливості.

Щоб задіяти `AspNet Identity`, в проект в папку `App_Start` додаються два файли. Файл `Startup.Auth.cs` містить клас запуску додатка `OWIN`. Оскільки `AspNet Identity` використовує інфраструктуру `OWIN`, то даний клас є одним з ключових і необхідних для роботи.

Файл `IdentityConfig.cs` містить ряд додаткових допоміжних класів: сервіси для двохфакторної валідації за допомогою email і телефону `EmailService` і `SmsService`, клас менеджера користувачів `ApplicationUserManager`, який додає до `UserManager` ряд додаткових функцій, і клас `ApplicationSignInManager`, використовуваний для входу і виходу з сайту.

Базова функціональність системи аутентифікації і управління обліковими записами розташована в двох контролерів: `AccountController` і `ManageController`

У `AccountController` визначені методи для логіна, реєстрації, верифікації коду, відправленого по email або по смс, скидання пароля, нагадування пароля, вхід на сайт за допомогою зовнішніх сервісів. Контролер `ManageController` використовується для управління своїм обліковим записом та передбачає можливості по зміні пароля і управління телефонними номерами в системі. Для обох контролерів вже за замовчуванням генеруються всі необхідні подання та спеціальні моделі представлень.

Незважаючи на те, що за замовчуванням нам вже надається готовий функціонал, однак в ряді випадків, наприклад, для відправки смс або електронної пошти необхідна додаткова настройка. Тепер розглянемо основні моменти системи `AspNet Identity`.

3.12.2 Базові класи `AspNet Identity`

Систему `AspNet Identity` утворює безліч різних класів, призначених для різних завдань. Розглянемо основні класи.

Контекст даних IdentityDbContext

Для взаємодії з базою даних в просторі імен Microsoft.AspNet.Identity.EntityFramework визначено клас IdentityDbContext. Це звичайний контекст даних, похідний від DbContext, який вже містить властивості, необхідні для управління користувачами і ролями: властивості Users і Roles. Хоча в реальному додатку ми будемо мати справу з класами, похідними від IdentityDbContext. Наприклад, у нас є такий контекст даних:

```
public class ApplicationContext : IdentityDbContext<ApplicationUser>
{
    public ApplicationContext() : base("IdentityDb") { }
    public static ApplicationContext Create()
    {
        return new ApplicationContext();
    }
}
```

Виведемо всіх користувачів з БД:

```
public ActionResult Index()
{
    List<ApplicationUser> users = new List<ApplicationUser>();
    using(ApplicationContext db = new ApplicationContext())
    {
        users = db.Users.ToList();
    }
    return View(users);
}
```

Користувачі

Функціональність користувачів в Asp Net Identity зосереджена в класі IdentityUser, який визначений в просторі імен Microsoft.AspNet.Identity.EntityFramework. IdentityUser реалізує інтерфейс IUser і визначає наступні властивості:

- Claims: повертає колекцію claims - спеціальних атрибутів, якими володіє користувач і які зберігають про користувача певну інформацію
- Email: email користувача
- Id: унікальний ідентифікатор користувача
- Logins: повертає колекцію логінів користувача
- PasswordHash: повертає хеш пароля
- Roles: повертає колекцію ролей користувача
- PhoneNumber: повертає номер телефону

- SecurityStamp: повертає деяке значення, яке змінюється при кожній зміні налаштувань аутентифікації для даного користувача
- UserName: повертає нік користувача
- AccessFailedCount: число спроб невдалого входу в систему
- EmailConfirmed: повертає true, якщо email був підтверджений
- PhoneNumberConfirmed: повертає true, якщо телефонний номер був підтверджений
- TwoFactorEnabled: якщо дорівнює true, то для даного користувача включена двухфакторная авторизація

Як правило, для управління користувачами визначають клас, похідний від IdentityUser:

Менеджер користувачів User Manager

Безпосереднє управління користувачами здійснюється з допомогою класу UserManager <T>, яке знаходиться в просторі імен Microsoft.AspNet.Identity.

Цей клас визначає безліч методів, які мають як синхронні, так і асинхронні версії. Перелічимо основні:

- Change Password (id, old, new) / Change Password Async (id, old, new): змінює пароль користувача
- Create (user) / CreateAsync (user): створює нового користувача
- Delete (user) / DeleteAsync (user): видаляє користувача
- Find (user, pass) / FindAsync (user, pass): шукає користувача за певним логіном і паролем
- FindById (id) / FindByIdAsync (id): шукає користувача по id
- FindByEmail (email) / FindByEmailAsync (email): шукає користувача по email
- FindByName (name) / FindByNameAsync (name): шукає користувача по ніку
- Update (user) / UpdateAsync (user): оновлює користувача
- Users: повертає всіх користувачів
- AddToRole (id, name) / AddToRoleAsync (id, name): додає для користувача з певним id роль name
- GetRoles (id) / GetRolesAsync (id): отримує всі ролі користувача по id
- IsInRole (id, name) / IsInRoleAsync (id, name): повертає true, якщо користувач з даними id має роль name
- RemoveFromRole (id, name) / RemoveFromRoleAsync (id, name): видаляє роль name у користувача по id

Створюємо клас для управління користувачами:

```
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.EntityFramework;
```

```

using Microsoft.AspNet.Identity.Owin;
using Microsoft.Owin;

namespace AspNetIdentityApp.Models
{
    public class ApplicationManager : UserManager<ApplicationUser>
    {
        public ApplicationManager(IUserStore<ApplicationUser> store) : base(store)
        {
        }
        public static ApplicationManager Create (IdentityFactoryOptions<ApplicationManager>
options,
                                IOwinContext context)
        {
            ApplicationContext db = context.Get<ApplicationContext>();
            ApplicationManager manager = new ApplicationManager(new
UserStore<ApplicationUser>(db));
            return manager;
        }
    }
}

```

Для створення класу використовується об'єкт User Store <ApplicationUser>. Клас UserStore представляє реалізацію інтерфейсу IUserStore <T>. У свою чергу, щоб створити об'єкт UserStore, необхідно використовувати контекст даних ApplicationContext.

Ролі в AspNet Identity

Кожна роль в ASP.NET Identity представляє об'єкт інтерфейсу IRole. У Entity Framework є вбудована реалізація даного інтерфейсу у вигляді класу IdentityRole.

Клас IdentityRole має такі властивості:

- Id: унікальний ідентифікатор ролі
- Name: назва ролі
- Users: колекція об'єктів IdentityUserRole, який пов'язують користувача і роль

Для управління ролями в ASP.NET Identity є клас RoleManager <T>, де T представляє реалізацію інтерфейсу IRole. Клас RoleManager управляє ролями за допомогою наступних синхронних і асинхронних методів:

- Create (role) / CreateAsync (role): створює нову роль з ім'ям role
- Delete (role) / DeleteAsync (role): видаляє роль з ім'ям role
- FindById (id) / FindByIdAsync (id): шукає роль по id
- FindByName (name) / FindByNameAsync (name): шукає роль за назвою

- RoleExists (name) / RoleExistsAsync (name): повертає true, якщо роль з даною назвою існує
- Update (role) / UpdateAsync (role): оновлює роль
- Roles: повертає набір всіх наявних ролей

Тепер розглянемо застосування цих класів в додатку на ASP.NET MVC.

3.12.3 Авторизація через зовнішні сервіси.

Інфраструктура OWIN і AspNet Identity дозволяють виробляти авторизацію через зовнішні сервіси за допомогою акаунтів в Google, Twitter, Facebook і т.д. Хоча подібна функціональність була доступна і попередньої версії - в MVC4, в MVC5 її механізм істотно змінився, і за аутентифікацію відповідають компоненти OWIN.

Візьмемо стандартний проект MVC5, що використовує AspNet Identity, тобто має в якості типу аутентифікації Individual User Accounts. У проекті в папці App_Start ми можемо побачити файл Startup.Auth.cs, який відповідає за настройку конфігурації OWIN:

```
using System;
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.Owin;
using Microsoft.Owin;
using Microsoft.Owin.Security.Cookies;
using Microsoft.Owin.Security.Google;
using Owin;
using GoogleAuthApp.Models;

namespace GoogleAuthApp
{
    public partial class Startup
    {
        public void ConfigureAuth(IAppBuilder app)
        {
            app.CreatePerOwinContext(ApplicationDbContext.Create);
            app.CreatePerOwinContext<ApplicationUserManager>(ApplicationUserManager.Create);
            app.CreatePerOwinContext<ApplicationSignInManager>(ApplicationSignInManager.Create);

            app.UseCookieAuthentication(new CookieAuthenticationOptions
            {
                AuthenticationType = DefaultAuthenticationTypes.ApplicationCookie,
                LoginPath = new PathString("/Account/Login"),
                Provider = new CookieAuthenticationProvider
                {
                    OnValidateIdentity =
                        SecurityStampValidator.OnValidateIdentity<ApplicationUserManager, ApplicationUser>(
```



```

        validateInterval: TimeSpan.FromMinutes(30),
        regenerateIdentity: (manager, user) => user.GenerateUserIdentityAsync(manager))
    }
});
app.UseExternalSignInCookie(DefaultAuthenticationTypes.ExternalCookie);

app.UseTwoFactorSignInCookie(DefaultAuthenticationTypes.TwoFactorCookie,
TimeSpan.FromMinutes(5));

app.UseTwoFactorRememberBrowserCookie(DefaultAuthenticationTypes.TwoFactorRememberBrowserCookie);

//app.UseMicrosoftAccountAuthentication(
//    clientId: "",
//    clientSecret: "");

//app.UseTwitterAuthentication(
//    consumerKey: "",
//    consumerSecret: "");

//app.UseFacebookAuthentication(
//    appId: "",
//    appSecret: "");

//app.UseGoogleAuthentication(new GoogleOAuth2AuthenticationOptions()
//{
//    ClientId = "",
//    ClientSecret = ""
//});
}
}
}

```

В кінці методу `ConfigureAuth` можна побачити ряд закоментуваних за замовчуванням рядків, призначених для аутентифікації через зовнішні сервіси. Наприклад:

```

//app.UseFacebookAuthentication(
//    appId: "",
//    appSecret: "");

```

При реєстрації нашого застосування в соціальній мережі Facebook ми отримуємо ідентифікатор додатку в рамках цієї соцмережі і секретне слово. Ці значення ми даємо для параметрів `appId` і `appSecret` відповідно. І після цього, якщо у нас є аккаунт в мережі Facebook, ми можемо авторизуватися в додатку через цей аккаунт. Досить зручно - всього три рядки, а все інше за нас зробить інфраструктура ASP.NET MVC.

Для прикладу виберемо найпростіший варіант - аутентифікацію через Google. Щоб підключатися до провайдерів аутентифікації Google або Facebook, нам треба налаштувати проект на використання SSL. Для цього перейдемо до

панелі властивостей проекту і встановимо там параметр SSL Enabled рівним True.

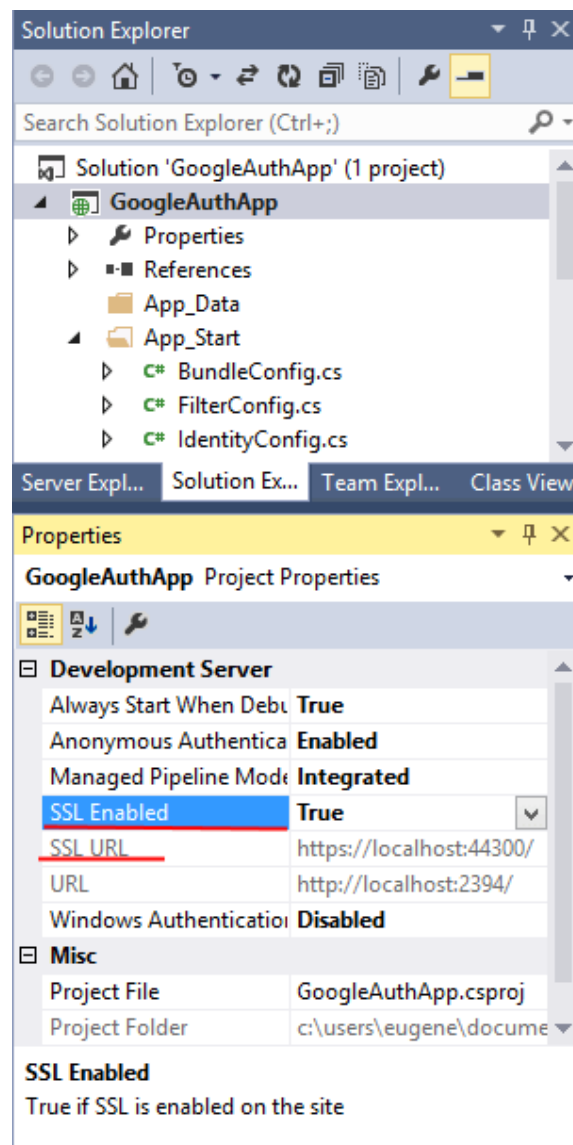


Рисунок 3.50 - Підключення провайдерів

Разом з установкою SSL також Visual Studio встановлює і нову адресу для SSL-з'єднання: <https://localhost:44300/>, який ми будемо використовувати для аутентифікації Google.

Тепер нам треба змінити адресу запуску проекту. Для цього скопіюємо ssl-адресу і перейдемо в настройки проекту на вкладку Web. Вставимо скопійований адресу в поле Project Url:

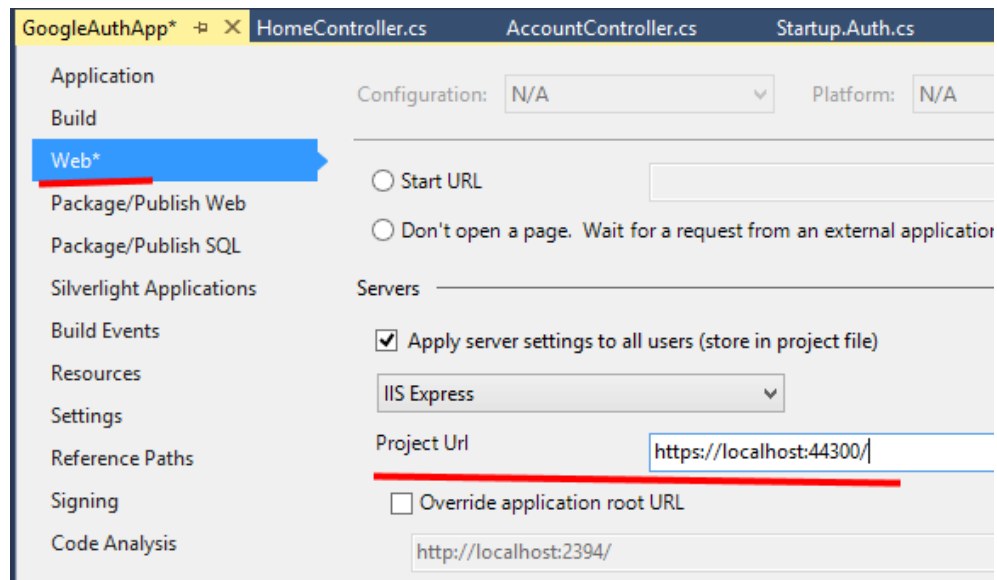


Рисунок 3.51 - Зміна локальної адреси

Перш за все нам треба буде створити в акаунті Google деякі настройки, які будуть використовуватися нашим додатком. Для цього авторизуємося в Google і перейдемо на сторінку Google Developers Console:

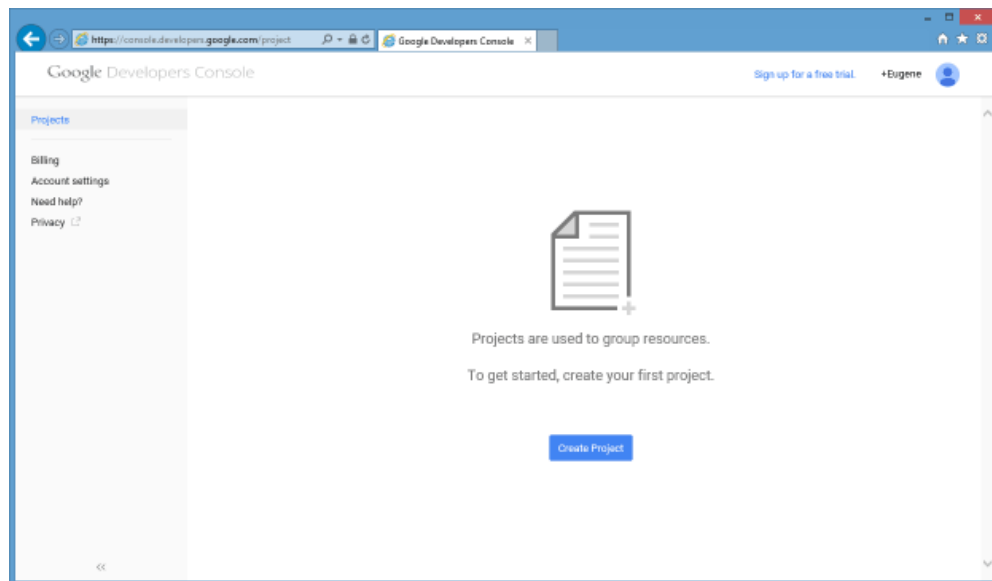


Рисунок 3.52 - сторінка Google Developers Console

На сторінці натиснемо на кнопку Create Project. У вікні введемо назву проекту і його ID:

New Project

PROJECT NAME ⓘ

My Project ✕

PROJECT ID ⓘ

snappy-beaker-853 ↻

[Show advanced options...](#)

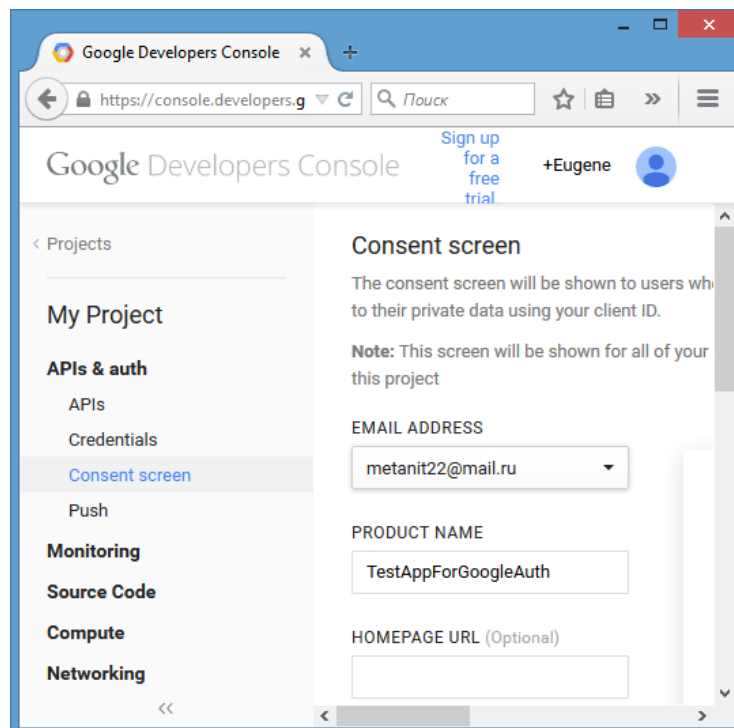
☒ I have read and agree to all [Terms of Service](#) for the Google Cloud Platform products.

Create Cancel

Рисунок 3.53 – Введення параметрів проекту

Google автоматично генерує деякі значення, як на рисунку вище. І в принципі ми можемо ці значення залишити, проте через деякий час буде створено новий проект, а перед нами відкриється панель налаштувань проекту.

У лівій колонці меню виберемо пункт **APIs & auth** і далі виберемо **Consent screen**. Тут нам треба встановити назву додатку в поле **Product Name**, а також вказати email:



Google Developers Console

Sign up for a free trial

+Eugene

< Projects

My Project

APIs & auth

APIs

Credentials

Consent screen

Push

Monitoring

Source Code

Compute

Networking

Consent screen

The consent screen will be shown to users who to their private data using your client ID.

Note: This screen will be shown for all of your this project

EMAIL ADDRESS

metanit22@mail.ru

PRODUCT NAME

TestAppForGoogleAuth

HOMEPAGE URL (Optional)

Рисунок 3.54 – Встановлення додаткових даних

У нашому випадку назва програми **TestAppForGoogleAuth**. Збережемо, натиснувши внизу сторінки на кнопку **Save**. Після цього перейдемо в лівій

панелі меню до пункту Credentials і на сторінці натиснемо на кнопку Create New Client ID. Нам відобразиться наступне діалогове вікно:

Create Client ID

APPLICATION TYPE

☒ **Web application**
Accessed by web browsers over a network.

☐ **Service account**
Calls Google APIs on behalf of your application instead of an end-user.
[Learn more](#)

☐ **Installed application**
Runs on a desktop computer or handheld device (like Android or iPhone).

AUTHORIZED JAVASCRIPT ORIGINS
Cannot contain a wildcard (http://*.example.com) or a path (http://example.com/subdir).

https://localhost:44300/

AUTHORIZED REDIRECT URIS
One URI per line. Needs to have a protocol, no URL fragments, and no relative paths. Can't be a non-private IP Address.

https://localhost:44300/signin-google

Create Client ID **Cancel**

Рисунок 3.55 - Встановлення додаткових даних

Тут є кілька опцій, які треба встановити. По-перше, в полі APPLICATION TYPE залишимо Web Application. В поле Authorized JavaScript введемо ssl-адреса нашого застосування, який був встановлений вище. Як правило, це https: // localhost: 44300 /

Після це в поле Authorized redirect URI введемо https: // localhost: 44300 / signin-google

Після введення всіх даних натиснемо на кнопку Create Client ID. Після цього Google згенерує нам ряд аутентифікаційних налаштувань, які ми будемо використовувати в додатку:

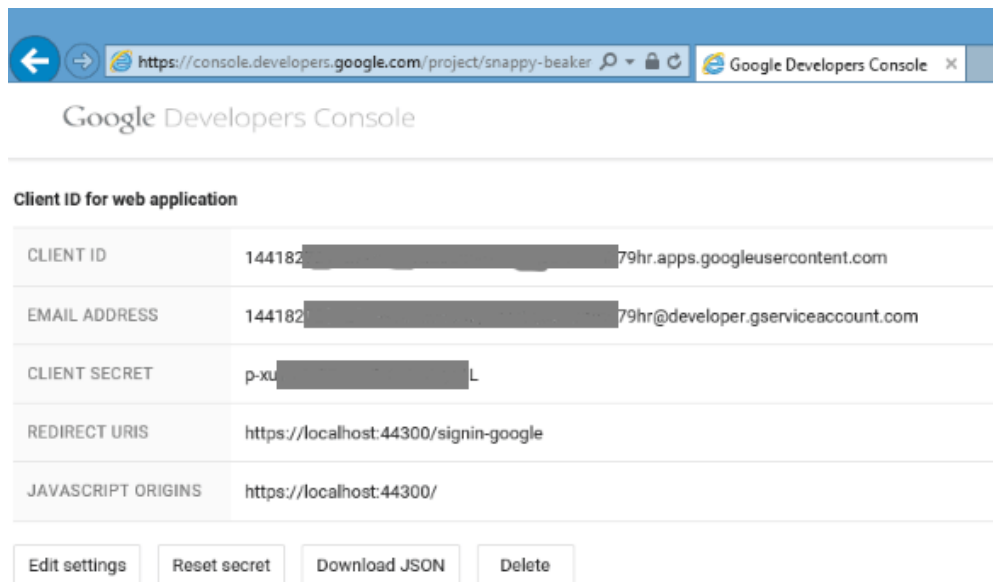


Рисунок 3.56 - Встановлення додаткових даних

Після отримання ідентифікаційних налаштувань оновимо файл Startup.Auth.cs:

```
app.UseGoogleAuthentication(new GoogleOAuth2AuthenticationOptions()
{
    ClientId = "144182.....79hr.apps.googleusercontent.com",
    ClientSecret = "p-xu.....L"
});
```

Потім переходимо в меню на вкладку APIs. Знайдемо пункт Google+ API і натиснемо на кнопку OFF:

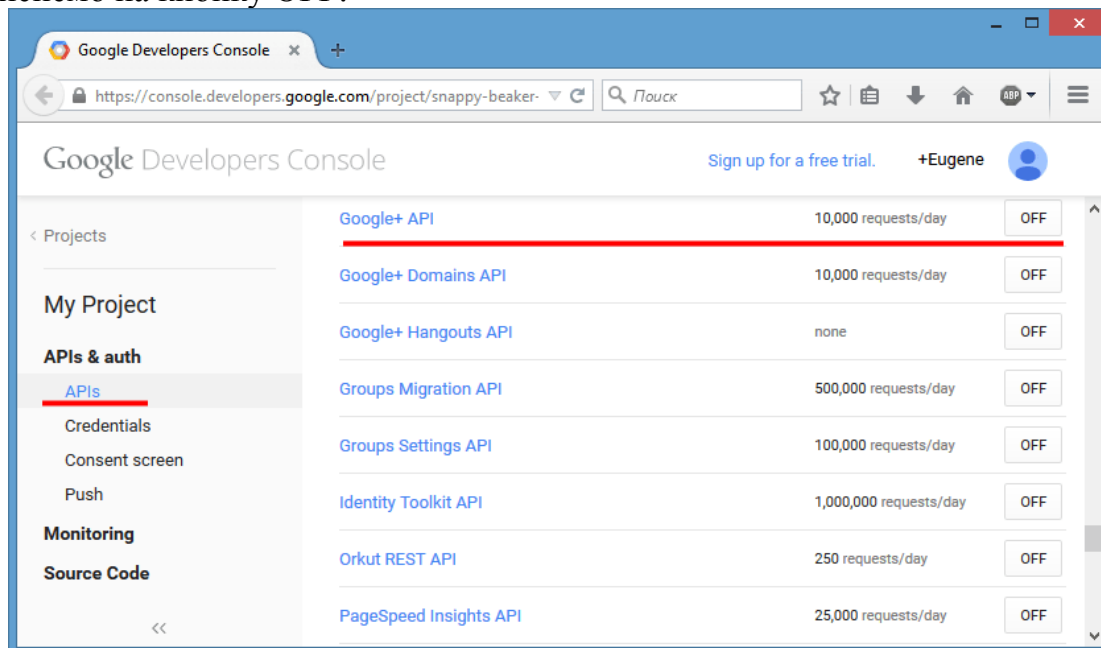


Рисунок 3.57 - Встановлення додаткових даних

Після всіх налаштувань встановимо відповідний атрибут ssl для контролера HomeController:

```
[RequireHttps]
public class HomeController : Controller
{
}
```

Запустимо проект і звернемося до методу Login, побачимо, що тепер нам доступно два способи входу на сайт: стандартний і через Google:

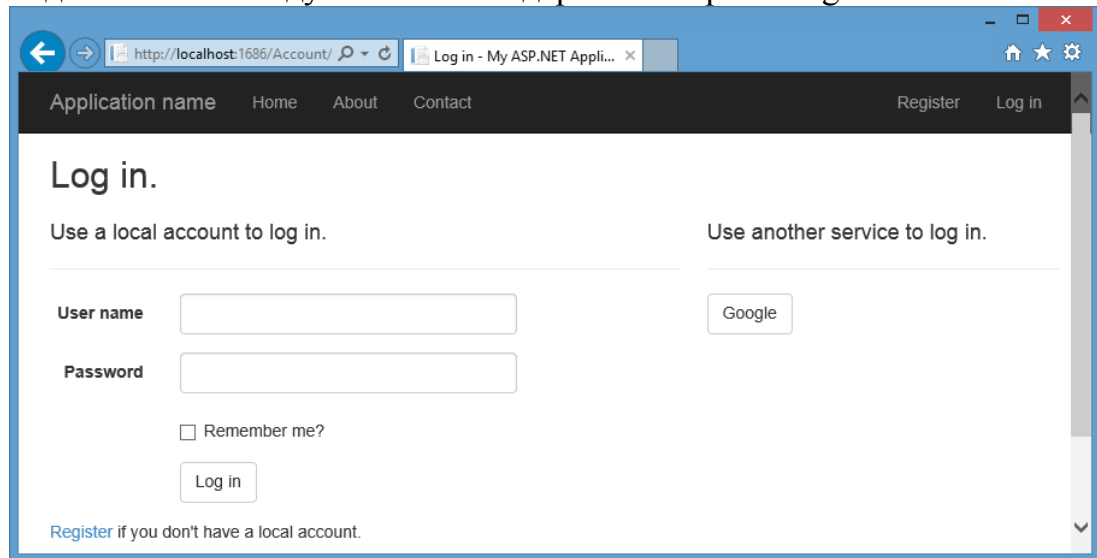


Рисунок 3.58 – Зміна способу входу

Після цього додаток просить прийняти деякі дозволи для доступу до нашого аккаунту :

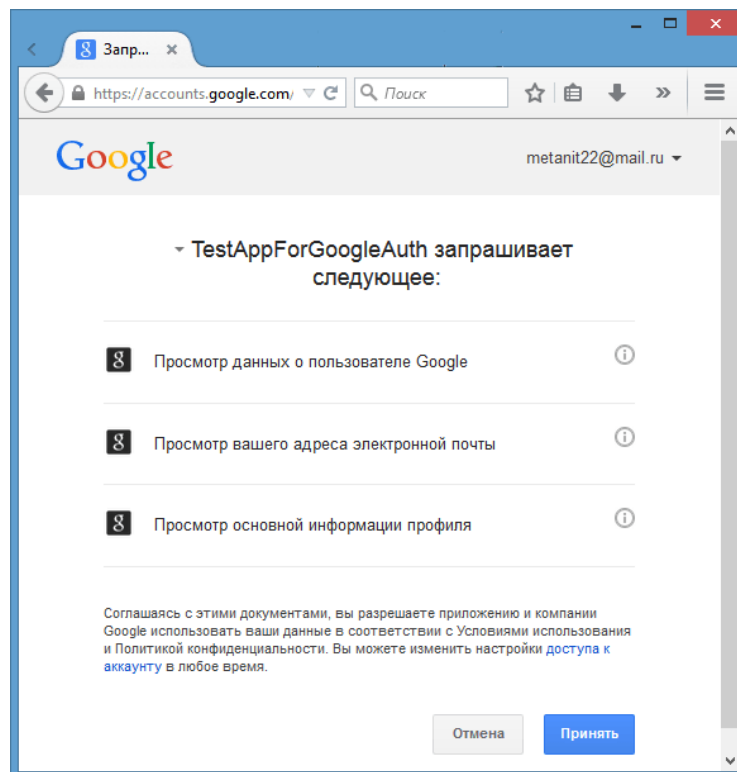


Рисунок 3.59 – Прийняття додаткових дозволів доступу

Натиснемо на кнопку **Прийняти**, і після цього ми перейдемо до поведінкового кроку - реєстрації зовнішнього логіна, який буде відображатися на сайті для даного облікового запису:

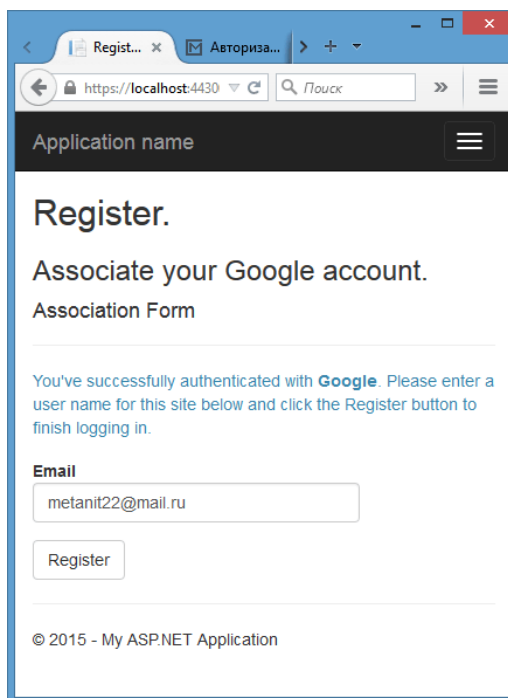


Рисунок 3.60 – Успішна ідентифікація

І після цього в меню почне відображатися логін:

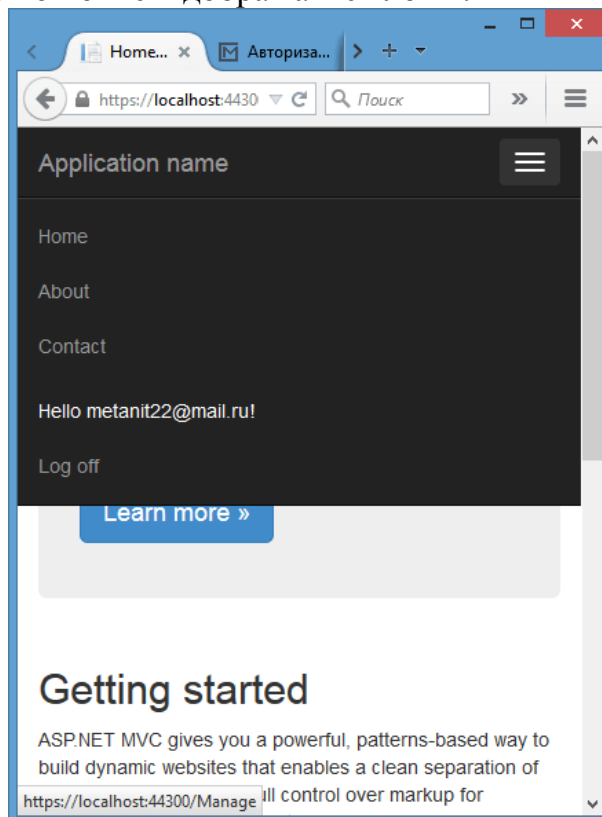
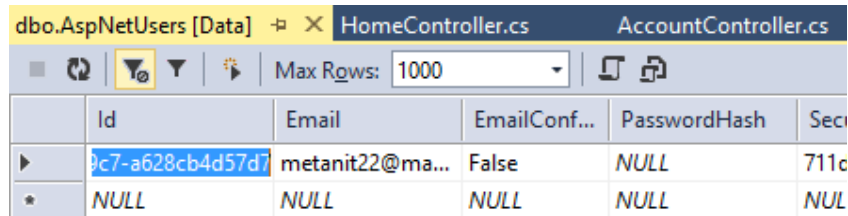


Рисунок 3.61 – Відображення логіна

За відображення логіна в поданні відповідає часткове представлення `_ExternalLoginsListPartial`. Дані користувача після завершення візиту на сайт не

потрапляють. Якщо ми відкриємо у вікні Server Explorer використовувану базу даних, то ми зможемо побачити, що в таблицю `AspNetUserLogins`, яка відповідає за зв'язок логінів сайту з зовнішніми акаунтами, був доданий новий запис:



	Id	Email	EmailConf...	PasswordHash	Sec...
▶	9c7-a628cb4d57d7	metanit22@ma...	False	NULL	711c
✱	NULL	NULL	NULL	NULL	NUL

Рисунок 3.62 – Відображення бази даних

Крім того, так як був доданий новий користувач, відповідний запис додається в таблицю `AspNetUsers`. Таким чином, ми можемо скористатися перевагами OWIN і OAuth2 і додати в додаток ідентифікацію за допомогою зовнішніх сервісів.

3.12.4 Перевизначення системи `AspNet Identity`

Раніше ми розглянули основні моменти `AspNet Identity` і то, як її використовувати. Однак вбудована система не завжди в повній мірі задовольняє потребам розробників. І може знадобитися перевизначити її: додати нові поля до моделі, додати нові таблиці і т.д. Розглянемо, як перевизначати базовий механізм `AspNet Identity`.

Як ми вже бачили, при використанні `AspNet Identity` за замовчуванням створюється база даних зі стандартним набором таблиць. Для взаємодії з базою даних в проекті визначено контекст даних `ApplicationDbContext`, який можна знайти в файлі `IdentityModels.cs` в папці `Models`:

```
public class ApplicationUser : IdentityUser
{
    public async Task<ClaimsIdentity> GenerateUserIdentityAsync(UserManager<ApplicationUser>
manager)
    {
        var userIdentity = await manager.CreateIdentityAsync(this,
DefaultAuthenticationTypes.ApplicationCookie);
        return userIdentity;
    }
}

public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
{
    public ApplicationDbContext()
        : base("DefaultConnection", throwIfV1Schema: false)
    {
    }

    public static ApplicationDbContext Create()
```

```

    {
        return new ApplicationDbContext();
    }
}

```

Крім того, тут також описується клас користувачів `ApplicationUser`, який успадковує всю логіку від `IdentityUser`. При реєстрації контекст даних заносить в БД логін і хеш пароля користувача. Однак, що якщо ми хочемо визначити ще ряд полів, наприклад, вік і т.д.? Змінимо моделі у файлі `IdentityModels.cs` наступним чином:

```

public class ApplicationUser : IdentityUser
{
    public int Age { get; set; } // додаємо властивість Age

    public async Task<ClaimsIdentity> GenerateUserIdentityAsync(UserManager<ApplicationUser>
manager)
    {
        var userIdentity = await manager.CreateIdentityAsync(this,
DefaultAuthenticationTypes.ApplicationCookie);
        return userIdentity;
    }
}
// додаємо модель Book
public class Book
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Author { get; set; }
}

public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
{
    public DbSet<Book> Books { get; set; }

    public ApplicationDbContext()
        : base("DefaultConnection", throwIfV1Schema: false)
    {
    }

    public static ApplicationDbContext Create()
    {
        return new ApplicationDbContext();
    }
}

```

По-перше, ми додали пару нових властивостей в клас `ApplicationUser`. По-друге, ми також додали новий клас `Book` і властивість `Books` в контекст даних. Оскільки у нас змінився контекст даних і модель користувача, то створена раніше база даних вже не буде відображати наші моделі, і її треба перебудувати.

У цьому нам допоможуть міграції. Отже, внизу Visual Studio в вікні Package Manager Console введемо команду: enable-migrations і натиснемо Enter:

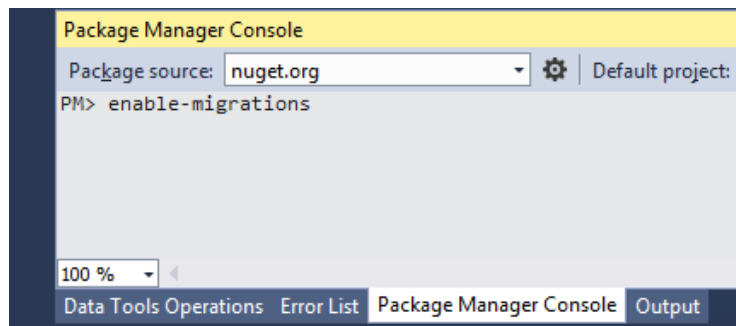


Рисунок 3.63 - Виконання міграцій

В результаті виконання цієї команди Visual Studio створює в проекті папку Migrations, в якій можна знайти файл Configuration.cs. Цей файл містить оголошення однойменного класу Configuration, який встановлює налаштування конфігурації:

```
namespace CustomIdentity.Migrations
{
    using System;
    using System.Data.Entity;
    using System.Data.Entity.Migrations;
    using System.Linq;
    internal sealed class Configuration :
    DbMigrationsConfiguration<CustomIdentity.Models.ApplicationDbContext>
    {
        public Configuration()
        {
            AutomaticMigrationsEnabled = false;
            ContextKey = "CustomIdentity.Models.ApplicationDbContext";
        }

        protected override void Seed(CustomIdentity.Models.ApplicationDbContext context)
        {
        }
    }
}
```

У методі Seed можна проініціалізувати базу даних початковими даними. Тепер нам треба створити саму міграцію. Там же в консолі Package Manager Console введемо команду:

```
PM> Add-Migration "DataMigration"
```

Після цього Visual Studio автоматично генерує клас міграції:

```
namespace CustomIdentity.Migrations
{
    using System;
    using System.Data.Entity.Migrations;
```

```

public partial class DataMigration : DbMigration
{
    public override void Up()
    {
        CreateTable(
            "dbo.Books",
            c => new
            {
                Id = c.Int(nullable: false, identity: true),
                Name = c.String(),
                Author = c.String(),
            })
            .PrimaryKey(t => t.Id);

        AddColumn("dbo.AspNetUsers", "Age", c => c.Int(nullable: false));
    }

    public override void Down()
    {
        DropColumn("dbo.AspNetUsers", "Age");
        DropTable("dbo.Books");
    }
}

```

І щоб виконати міграцію, застосуємо цей клас, набравши в тій же консолі команду: **PM> Update-Database**. Після цього, якщо ми подивимося на склад бази даних, то побачимо, що до неї були застосовані зміни відповідно до виконаної міграції:

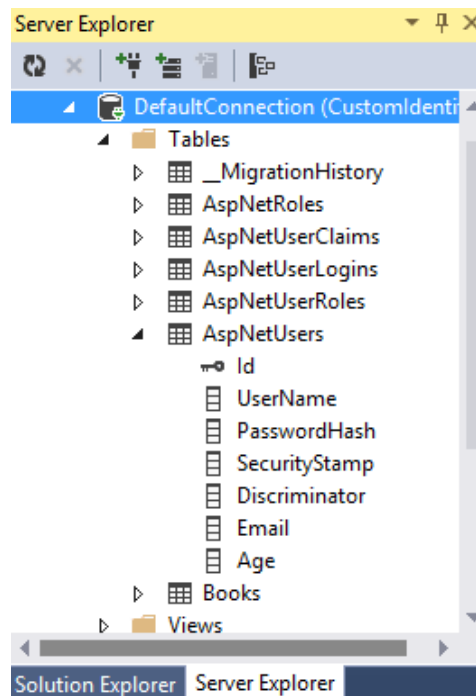


Рисунок 3.64 – Структура бази даних

Так як у нас з'явилося дві нових властивості, змінимо модель реєстрації в файлі AccountViewModels.cs, який знаходиться в папці Models:

```
public class RegisterViewModel
{
    [Required]
    public string Email { get; set; }

    [Required]
    public int Age { get; set; }

    [Required]
    [StringLength(100, ErrorMessage = "The {0} must be at least {2} characters long.",
    MinimumLength = 6)]
    [DataType(DataType.Password)]
    [Display(Name = "Password")]
    public string Password { get; set; }

    [DataType(DataType.Password)]
    [Display(Name = "Confirm password")]
    [Compare("Password", ErrorMessage = "The password and confirmation password do not
    match.")]
    public string ConfirmPassword { get; set; }
}
```

І також додамо в представлення для реєстрації Register.shtml пару нових полів:

```
<Div class = "form-group">
    @ Html.LabelFor (m => m.Age, new {class = "col-md-2 control-label"})
    <Div class = "col-md-10">
        @ Html.TextBoxFor (m => m.Age, new {class = "form-control"})
    </ Div>
</ Div>
```

Зовнішній вигляд сторінки реєстрації в підсумку буде виглядати так:

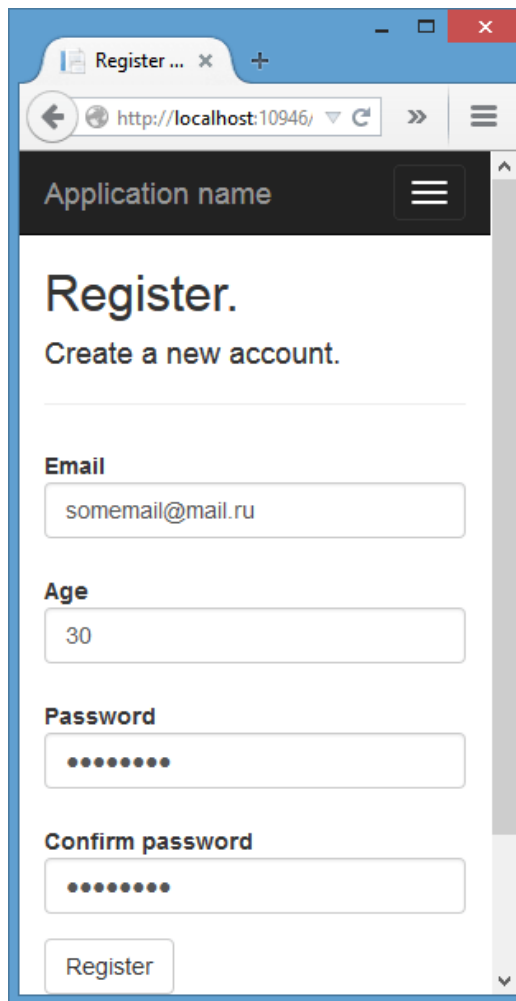


Рисунок 3.65 – Форма реєстрації

І змінимо код додавання нового користувача в методі Register контролера AccountController. Зараз він виглядає так:

```
var user = new ApplicationUser { UserName = model.Email, Email = model.Email };
```

змінимо його на:

```
var user = new ApplicationUser { UserName = model.Email, Email = model.Email, Age=model.Age };
```

Тепер ми можемо додавати нових користувачів, вказуючи у них додаткові властивості, а також при бажанні можемо зробити функціонал по додаванню об'єктів Book в таблицю Books та створити нові властивості моделей і нові таблиці.

3.12.5 Робота з ролями в ASP.NET Identity

Крім логінів і паролів система авторизації і аутентифікації пропонує ще такий компонент для розмежування доступу, як ролі. Ролі дозволяють створити групи користувачів з певними правами і в залежності від приналежності до тієї чи іншої групи, розмежувати доступ до ресурсів додатки. Візьмемо стандартний

проект ASP.NET MVC5 з моделлю аутентифікації Individual User Accounts. За замовчуванням система ASP.NET Identity включає всі необхідні засоби для управління ролями. На початку ми вже розбиралися з механізмом ASP.NET Identity. У стандартному проекті MVC 5 кожна роль представлена об'єктом класу IdentityRole, який реалізує інтерфейс IRole. За аналогією з управлінням користувачами і класом UserManager для управління ролями призначений клас RoleManager, який використовує в якості сховища ролей об'єкт RoleStore. Задіємо систему ролей в проекті. Наприклад, нам треба, щоб у нас в базі даних вже був один користувач, що виконує роль адміністратора. Отже, проініціалізуємо базу даних початковими значеннями для ролей і користувачів. Для цього додамо в папку Models новий клас:

```
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.EntityFramework;
using System.Data.Entity;
namespace RolesIdentityApp.Models
{
    public class AppDbInitializer : DropCreateDatabaseAlways<ApplicationDbContext>
    {
        protected override void Seed(ApplicationDbContext context)
        {
            var userManager = new ApplicationUserManager(new UserStore<ApplicationUser>(context));

            var roleManager = new RoleManager<IdentityRole>(new RoleStore<IdentityRole>(context));

            //створюємо дві ролі
            var role1 = new IdentityRole { Name = "admin" };
            var role2 = new IdentityRole { Name = "user" };

            // додаємо ролі в БД
            roleManager.Create(role1);
            roleManager.Create(role2);

            // створюємо користувачів
            var admin = new ApplicationUser { Email = "somemail@mail.ru", UserName =
"somemail@mail.ru" };
            string password = "ad46D_ewr3";
            var result = userManager.Create(admin, password);

            // якщо створення користувачів відбулося успішно
            if(result.Succeeded)
            {
                // додаємо для користувача роль
                userManager.AddToRole(admin.Id, role1.Name);
                userManager.AddToRole(admin.Id, role2.Name);
            }

            base.Seed(context);
        }
    }
}
```

```

    }
}

```

Так як у нас за замовчуванням вже визначено в проекті контекст даних для управління користувачами і ролями `ApplicationDbContext`, то ми його вказуємо при створенні об'єктів `UserManager` і `RoleManager`. Метод `roleManager.Create` дозволяє додати роль в БД в таблицю `AspNetRoles`, а метод `userManager.AddToRole` встановлює певну роль для користувача з переданим в метод `id`. Зверніть увагу, що ми можемо додати для одного користувача безліч ролей, як і в даному випадку. Для того, щоб ініціалізація бази даних запрацювала, додамо її виклик в файл `Global.asax` в метод `App_Start`:

```

protected void Application_Start()
{
    Database.SetInitializer<ApplicationDbContext>(new AppDbInitializer());

    AreaRegistration.RegisterAllAreas();
    FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
    RouteConfig.RegisterRoutes(RouteTable.Routes);
    BundleConfig.RegisterBundles(BundleTable.Bundles);
}

```

Тепер при запуску програми в базі даних виявиться один користувач і дві ролі. І тоді ми зможемо встановлювати розмежування доступу за ролями на рівні контролерів і методів :

```

public class HomeController : Controller
{
    [Authorize(Roles="admin")]
    public ActionResult About()
    {
        ViewBag.Message = "Your application description page.";

        return View();
    }
}

```

Тепер до методу `About` можуть звернутися тільки адміністратори. Але підемо далі. У нас є ще одна роль, і ми, для прикладу, хочемо, щоб при реєстрації всім користувачам привласнювалася роль `"user"`. Для цього змінимо метод `Register` наступним чином:

```

[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Register(RegisterViewModel model)
{
    if (ModelState.IsValid)
    {

```



```

var user = new ApplicationUser { UserName = model.Email, Email = model.Email };
var result = await UserManager.CreateAsync(user, model.Password);
if (result.Succeeded)
{
    // якщо створення відбулось успішно, додаємо роль користувача
    await UserManager.AddToRoleAsync(user.Id, "user");
    await SignInManager.SignInAsync(user, isPersistent:false, rememberBrowser:false);

    return RedirectToAction("Index", "Home");
}
AddErrors(result);
}
return View(model);
}

```

Тепер виведемо на веб-сторінку все ролі поточного користувача:

```

[Authorize]
public ActionResult Index()
{
    IList<string> roles= new List<string>{"Роль не определена"};
    ApplicationUser userManager = HttpContext.GetOwinContext()
        .GetUserManager<ApplicationUserManager>();
    ApplicationUser user = userManager.FindByEmail(User.Identity.Name);
    if (user != null)
        roles = userManager.GetRoles(user.Id);
    return View(roles);
}

```

За допомогою методу Get Roles () класу User Manager ми можемо отримати набір ролей певного користувача. Для видалення ролі у користувача використовується метод RemoveFromRole(), який також визначено у класу UserManager. Наприклад, видалимо у поточного користувача роль "user":

```
userManager.RemoveFromRole(user.Id, "user");
```

3.12.6 Аутентифікація форм

Хоча рекомендованим способом аутентифікації в додатках ASP.NET MVC5 є аутентифікація на основі AspNet Identity, але все ж це не єдиний доступний механізм. У попередніх версіях MVC механізм аутентифікації ґрунтувався на аутентифікації форм. І ми також можемо використовувати даний механізм в додатках ASP.NET MVC 5. Розглянемо це на прикладі. Створимо новий додаток для MVC 5, вибравши при створенні як тип аутентифікації No Authentication:

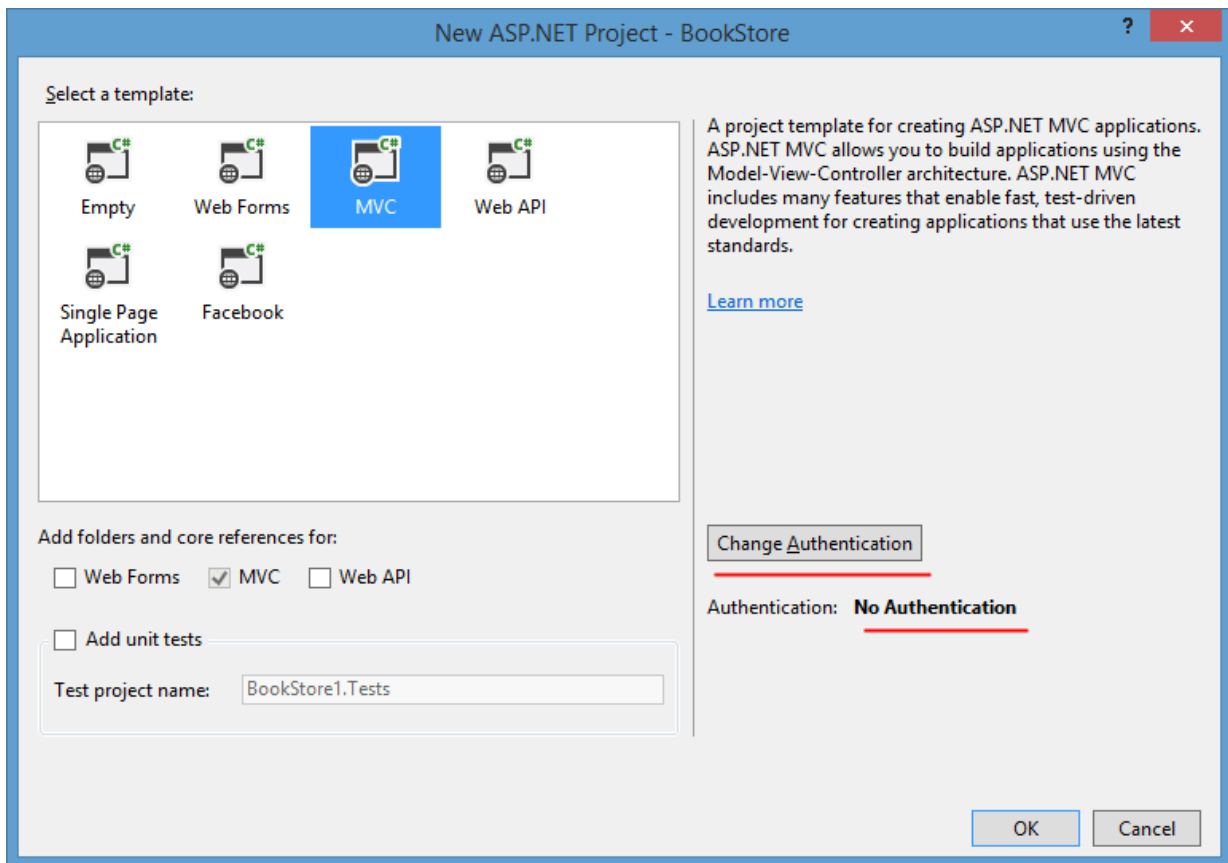


Рисунок 3.66 – Створення нового проекту

Оскільки всі дані про користувачів у нас будуть знаходитися в базі даних MS SQL Server, то для взаємодії з нею додамо в проект через NuGet бібліотеки EntityFramework. Після підключення Entity Framework в проект нам потрібно включити аутентифікацію форм. Це робиться в файлі web.config. Знайдемо в ньому секцію <system.web> і всередині цієї секції помістимо наступну підсекцію:

```
<authentication mode="Forms">
  <forms name="cookies" timeout="2880" loginUrl="~/Account/Login" />
</authentication>
```

Встановивши mode = "Forms", ми тим самим підключити аутентифікацію форм. Далі ми задаємо ряд параметрів. Нам доступний в цілому наступний список параметрів:

- **cookieless**: визначає, чи застосовуються куки-набори та як вони використовуються. Може приймати наступні значення: **UseCookies** (визначає, що куки-набори будуть використовуватися завжди незалежно від пристрою), **UseUri** (куки-набори ніколи не використовуються), **AutoDetect** (якщо пристрій підтримує куки-набори, то вони використовуються, в іншому випадку вони не застосовуються, при цьому проводиться тестування, яке визначає, чи дозволені), **UseDeviceProfile** (використовується за умовчанням) (якщо пристрій підтримує куки-набори, то вони використовуються, в

іншому випадку вони не застосовуються, на відміну від попереднього випадку тестування не проводиться)

- **defaultUrl**: визначає шлях, по якому здійснюється перехід після авторизації
- **domain**: визначає куки-набори для всього домену. Завдяки цьому ми можемо використовувати одні і ті ж куки-набори для головного домену та його субдоменів. За замовчуванням має значення як порожнього рядка
- **loginUrl**: адреса для аутентифікації користувача. Значення за замовчуванням - "~/ Account / Login"
- **name**: задає ім'я для куки-набору. Значення за замовчуванням - ".ASPXAUTH"
- **path**: задає шлях для куки-наборів. Значення за замовчуванням - "/"
- **requireSSL**: визначає, чи треба SSL-з'єднання для передачі куки-наборів. Значення за замовчуванням false
- **timeout**: визначає термін дії куків в хвилинах

У нашому випадку ми задаємо тільки три параметри. Також додамо в файл web.config рядок підключення для EntityFramework:

```
<connectionStrings>
  <add name="DefaultConnection" connectionString="Data Source=(LocalDb)\v11.0;
    AttachDbFilename=|DataDirectory|\FormsAuth.mdf;
    Initial Catalog=FormsAuth;Integrated Security=True"
    providerName="System.Data.SqlClient" />
</connectionStrings>
```

Тепер треба створити всю необхідну інфраструктуру для аутентифікації - моделі, контролери та представлення. Створимо модель User, яка буде описувати користувача:

```
public class User
{
    public int Id { get; set; }
    public string Email { get; set; }
    public string Password { get;set;}
    public int Age { get; set; }
}
```

Також додамо контекст даних User Context Для взаємодії з БД:

```
public class UserContext : DbContext
{
    public UserContext() :
        base("DefaultConnection")
    {}
    public DbSet<User> Users { get; set; }
}
```

При реєстрації та логуванні ми не будемо безпосередньо взаємодіяти з моделлю User. Замість цього ми будемо використовувати спеціальні моделі, які також додамо в проект:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.ComponentModel.DataAnnotations;

namespace FormsAuthApp.Models
{
    public class LoginModel
    {
        [Required]
        public string Name { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }

    public class RegisterModel
    {
        [Required]
        public string Name { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }

        [Required]
        [DataType(DataType.Password)]
        [Compare("Password", ErrorMessage="Паролі не співпадають")]
        public string ConfirmPassword { get; set; }

        [Required]
        public int Age { get; set; }
    }
}
```

Ці моделі будуть використовуватися відповідно для представлень логіна і реєстрації. Нехай представлення для реєстрації буде виглядати наступним чином:

```
@model FormsAuthApp.Models.RegisterModel

@{
    ViewBag.Title = "Реєстрація";
}
```

<h2> Реєстрація </h2>

```
@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()

    <div class="form-horizontal">
        @Html.ValidationSummary(true)

        <div class="form-group">
            @Html.LabelFor(model => model.Name, new { @class = "control-label col-md-2" })
            <div class="col-md-10">
                @Html.EditorFor(model => model.Name)
                @Html.ValidationMessageFor(model => model.Name)
            </div>
        </div>

        <div class="form-group">
            @Html.LabelFor(model => model.Password, new { @class = "control-label col-md-2" })
            <div class="col-md-10">
                @Html.EditorFor(model => model.Password)
                @Html.ValidationMessageFor(model => model.Password)
            </div>
        </div>

        <div class="form-group">
            @Html.LabelFor(model => model.ConfirmPassword, new { @class = "control-label col-md-2" })
        </div>
        <div class="col-md-10">
            @Html.EditorFor(model => model.ConfirmPassword)
            @Html.ValidationMessageFor(model => model.ConfirmPassword)
        </div>
    </div>

    <div class="form-group">
        @Html.LabelFor(model => model.Age, new { @class = "control-label col-md-2" })
        <div class="col-md-10">
            @Html.EditorFor(model => model.Age)
            @Html.ValidationMessageFor(model => model.Age)
        </div>
    </div>

    <div class="form-group">
        <div class="col-md-offset-2 col-md-10">
            <input type="submit" value="Зареєструвати" class="btn btn-default" />
        </div>
    </div>
</div>
}
```

```
@section Scripts {
    @Scripts.Render("~/bundles/jqueryval")
}
```

Представлення для логіна буде виглядати аналогічно. Тепер додамо новий контролер AccountController з наступним вмістом:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Security;
using System.Web.Mvc;
using FormsAuthApp.Models;

namespace FormsAuthApp.Controllers
{
    public class AccountController : Controller
    {
        public ActionResult Login()
        {
            return View();
        }

        [HttpPost]
        [ValidateAntiForgeryToken]
        public ActionResult Login(LoginModel model)
        {
            if (ModelState.IsValid)
            {
                // пошук користувача в БД
                User user = null;
                using (UserContext db = new UserContext())
                {
                    user = db.Users.FirstOrDefault(u => u.Email == model.Name && u.Password ==
model.Password);
                }
                if (user != null)
                {
                    FormsAuthentication.SetAuthCookie(model.Name, true);
                    return RedirectToAction("Index", "Home");
                }
                else
                {
                    ModelState.AddModelError("", "Користувача з таким логіном і паролем немає");
                }
            }

            return View(model);
        }
    }
}
```

```

public ActionResult Register()
{
    return View();
}
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Register(RegisterModel model)
{
    if(ModelState.IsValid)
    {
        User user = null;
        using (UserContext db = new UserContext())
        {
            user = db.Users.FirstOrDefault(u => u.Email == model.Name);
        }
        if (user == null)
        {
            // створюємо нового користувача
            using (UserContext db = new UserContext())
            {
                db.Users.Add(new User { Email = model.Name, Password = model.Password,
Age=model.Age });
                db.SaveChanges();

                user = db.Users.Where(u => u.Email == model.Name && u.Password ==
model.Password).FirstOrDefault();
            }
            // якщо користувач успішно було додано в БД
            if (user != null)
            {
                FormsAuthentication.SetAuthCookie(model.Name, true);
                return RedirectToAction("Index", "Home");
            }
        }
        else
        {
            ModelState.AddModelError("", "Користувач з таким логіном вже існує");
        }
    }

    return View(model);
}
public ActionResult Logoff()
{
    FormsAuthentication.SignOut();
    return RedirectToAction("Index", "Home");
}
}
}

```

При реєстрації ми додаємо нового користувача в БД, а при логіні дивимося, чи є такий користувач. І якщо користувач знайдений, то за допомогою аутентифікації форм встановлюємо куки:

```
FormsAuthentication.SetAuthCookie (model.Name, true);
```

Тепер, якщо користувач аутентифікований в системі, і його куки встановлені, то ми можемо ідентифікувати його в будь-якому місці програми. Наприклад, за замовчуванням у нас вже є контролер Home, а в ньому метод Index. Змінимо цей метод так, щоб він повертав логін аутентифікованого користувача:

```
public string Index()
{
    string result = "Ви не авторизовані";
    if(User.Identity.IsAuthenticated)
    {
        result = "Ваш логін: " + User.Identity.Name;
    }
    return result;
}
```

Вся інформація про користувача в Asp.Net MVC зберігається у властивості HttpContext.User, яке представляє реалізацію інтерфейсу IPrincipal, який визначений в просторі імен System.Security.Principal. Інтерфейс IPrincipal визначає властивість Identity, яке зберігає об'єкт інтерфейсу IIdentity, який описує поточного користувача.

Інтерфейс IIdentity містить наступні властивості:

- AuthenticationType: тип аутентифікації
- IsAuthenticated: якщо користувач аутентифікований, то повертає true
- Name: ім'я користувача в системі

Для визначення чи аутентифікований користувач, ASP.NET MVC приймає від браузера куки, і якщо користувач аутентифікований, у властивості IIdentity.IsAuthenticated встановлюється значення true, а властивість Name отримує в якості значення ім'я користувача.

3.12.7 Використання об'єктів Claim

Об'єкти Claim дозволяють нам зберігати додаткову інформацію, пов'язану з поточним користувачем. Наприклад, візьмемо новий стандартний проект, який використовує аутентифікацію Individual User Accounts і змінимо в ньому клас ApplicationUser, який знаходиться в файлі IdentityModels.cs:

```
public class ApplicationUser : IdentityUser
{
    public async Task<ClaimsIdentity> GenerateUserIdentityAsync(UserManager<ApplicationUser>
manager)
    {
```



```

        var userIdentity = await manager.CreateIdentityAsync(this,
DefaultAuthenticationTypes.ApplicationCookie);
        userIdentity.AddClaim(new Claim(ClaimTypes.Gender, this.Gender));
        userIdentity.AddClaim(new Claim("age", this.Age.ToString()));
        return userIdentity;
    }

    public int Age { get; set; }
    public string Gender { get; set; }
}

```

Ми визначили дві додаткових властивості в класі `ApplicationUser`, а також для кожного додаткового властивості додали відповідний об'єкт `claim`. Виклик методу `GenerateUserIdentityAsync`, який вже є в класі за замовчуванням, повертає нам об'єкт `ClaimsIdentity`, для якого власне і проводиться додавання `claims`.

Додавання дуже просте: використовуємо метод `AddClaim`, в який передаємо новий `Claim`. Об'єкт `Claim` приймає два значення: тип клейма у вигляді рядка і його значення рядка. За замовчуванням фреймворк містить ряд константних значень для типів `claim` в перерахуванні `ClaimTypes`, однак цей набір обмежений, тому для збереження віку використовується рядок `"age"`, так як `ClaimTypes` не містить потрібного типу.

Щоб користувач зміг поставити ці дані при реєстрації, змінимо модель реєстрації `RegisterViewModel` (що знаходиться в файлі `AccountViewModels.cs`), яка буде містити всі ці властивості:

```

public class RegisterViewModel
{
    [Required]
    [Display(Name = "Email")]
    public string Email { get; set; }

    [Required]
    [Display(Name = "Age")]
    public int Age { get; set; }

    [Required]
    [Display(Name = "Gender")]
    public string Gender { get; set; }

    [Required]
    [DataType(DataType.Password)]
    [Display(Name = "Password")]
    public string Password { get; set; }

    [DataType(DataType.Password)]
    [Display(Name = "Confirm password")]
    [Compare("Password", ErrorMessage = "The password and confirmation password do not match.")]
    public string ConfirmPassword { get; set; }
}

```

```
}
```

Відповідно нам треба змінити і представлення реєстрації, наприклад, наступним чином:

```
@model ClaimsApp.Models.RegisterViewModel
@{
    ViewBag.Title = "Register";
    Layout = null;
}

<h2>@ViewBag.Title.</h2>

@using (Html.BeginForm("Register", "Account", FormMethod.Post))
{
    @Html.AntiForgeryToken()
    @Html.ValidationSummary()
    <div>
        @Html.LabelFor(m => m.Email)
        <div>
            @Html.TextBoxFor(m => m.Email)
        </div>
    </div>
    <div>
        @Html.LabelFor(m => m.Password)
        <div>
            @Html.PasswordFor(m => m.Password)
        </div>
    </div>
    <div>
        @Html.LabelFor(m => m.ConfirmPassword)
        <div>
            @Html.PasswordFor(m => m.ConfirmPassword)
        </div>
    </div>

    <div>
        @Html.LabelFor(m => m.Age)
        <div>
            @Html.TextBoxFor(m => m.Age)
        </div>
    </div>
    <div>
        @Html.LabelFor(m => m.Gender)
        <div>
            @Html.RadioButtonFor(m => m.Gender, "жіночий")
            <span>жіночий</span>
        </div>
        <div>
            @Html.RadioButtonFor(m => m.Gender, "чоловічий")
            <span>чоловічий</span>
        </div>
    </div>
}
```

```

</div>
<br />
<div>
    <div>
        <input type="submit" value="Register" />
    </div>
</div>
}

```

```

@section Scripts {
    @Scripts.Render("~/bundles/jqueryval")
}

```

І в методі Register контролера AccountController змінимо створення нового користувача:

```

[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Register(RegisterViewModel model)
{
    if (ModelState.IsValid)
    {
        var user = new ApplicationUser { UserName = model.Email, Email = model.Email,
Age=model.Age, Gender=model.Gender };
        //.....
    }
}

```

І щоб перевірити в дії цей функціонал, визначимо в контролері HomeController нову дію:

```

public string GetInfo()
{
    var identity = (ClaimsPrincipal)Thread.CurrentPrincipal;
    var email = HttpContext.User.Identity.Name;
    var gender = identity.Claims.Where(c => c.Type == ClaimTypes.Gender).Select(c =>
c.Value).SingleOrDefault();
    var age = identity.Claims.Where(c => c.Type == "age").Select(c => c.Value).SingleOrDefault();
    return "<p>Ел. адреса: " + email + "</p><p>Стать:" + gender + "</p><p>Вік:" + age + "</p>";
}

```

Таким чином, завдяки використанню claims ми можемо зберігати ряд важливої інформації, пов'язаної з поточним користувачем, і зменшити кількість звернень до БД для отримання цієї інформації.

3.12.8 Налаштування валідації пароля в ASP.NET Identity

При використанні системи аутентифікації та авторизації ASP.NET Identity ми можемо зіткнутися з такою проблемою: при введенні пароля спрацьовує вбудована логіка валідації - довжина пароля не може бути менше 6 символів, і ми цю логіку ніяк не можемо змінити, використовуючи стандартний механізм

атрибутів валідації. Але що, якщо нам треба встановити іншу мінімальну довжину пароля? Або якщо ми хочемо, щоб в паролі могли б використовуватися тільки цифрові або тільки алфавітні символи?

І незважаючи на те, що вбудована логіка валідації прихована від наших очей, ми її можемо перевизначити. За механізм валідації відповідає властивість `PasswordValidator` класу `userManager`. Воно зберігає об'єкт валідації, який ми можемо змінити. Якщо ми перейдемо в файл `IdentityConfig.cs` в папці `App_Start`, то у визначенні класу `ApplicationUserManager` ми можемо побачити застосування валідатора пароля:

```
var manager = new ApplicationUserManager
    (new UserStore<ApplicationUser>
        (context.Get<ApplicationDbContext>()));

//.....

manager.PasswordValidator = new PasswordValidator
{
    RequiredLength = 6,
    RequireNonLetterOrDigit = true,
    RequireDigit = true,
    RequireLowercase = true,
    RequireUppercase = true,
};
```

Для визначення валідації пароля тут застосовується вбудований клас `Password Validator`, який має такі властивості:

- **RequiredLength**: мінімальна довжина пароля
- **RequireNonLetterOrDigit**: якщо одно `true`, то пароль повинен буде мати як мінімум один символ, який не є алфавітно-цифровим
- **RequireDigit**: якщо одно `true`, то пароль повинен буде мати як мінімум одну цифру
- **RequireLowercase**: якщо одно `true`, то пароль повинен буде мати як мінімум один символ в нижньому регістрі
- **RequireUppercase**: якщо одно `true`, то пароль повинен буде мати як мінімум один символ у верхньому регістрі

Для більшості випадків даного функціоналу цілком вистачить, ми можемо налаштувати мінімальну довжину пароля, введення алфавітних, цифрових або неалфавітний-цифрових символів. Однак, якщо нам буде потрібно більш складний логіка валідації, то доведеться визначати свій клас валідатора. Об'єкт валідатора представляє собою об'єкт інтерфейсу `IIdentityValidator`. В даному випадку в якості такого об'єкта використовується екземпляр класу `PasswordValidator`, який встановлює мінімальну довжину пароля і ряд інших правил. Однак більше нічого з цим валідатором зробити не можливо. Тому для створення більш витонченої логіки валідації нам треба створити окремий клас. Отже, створимо наступний клас:

```

public class CustomPasswordValidator : IIdentityValidator<string>
{
    public int RequiredLength { get; set; } // мінімальна довжина
    public CustomPasswordValidator(int length)
    {
        RequiredLength = length;
    }
    public Task<IdentityResult> ValidateAsync(string item)
    {
        if (String.IsNullOrEmpty(item) || item.Length < RequiredLength)
        {
            return Task.FromResult(IdentityResult.Failed(
                String.Format("Мінімальна довжина пароля рівна {0}", RequiredLength)));
        }
        string pattern = "[0-9]+$";

        if (!Regex.IsMatch(item, pattern))
        {
            return Task.FromResult(IdentityResult.Failed("Пароль повинен складатися тільки з
цифр"));
        }

        return Task.FromResult(IdentityResult.Success);
    }
}

```

Клас валідатора реалізує інтерфейс `IIdentity Validator`, який має один метод `Validate Async`. Цей метод приймає рядок, на місце якого потім в програмі буде передаватися рядок валідованого пароля.

Крім перевірки кількості символів пароль тут також перевіряється на відповідність регулярному виразу, яке говорить про те, що всі символи повинні представляти цифри (`string pattern = "[0-9] + $"`). Таким чином, ми можемо визначити в методі різні способи перевірок. І потім також у визначенні класу `ApplicationUserManager` ми можемо його використовувати:

```

var manager = new ApplicationUserManager
    (new UserStore<ApplicationUser>
        (context.Get<ApplicationDbContext>()));
//.....
manager.PasswordValidator = new CustomPasswordValidator(5);

```

3.12.9 Валідація користувача в ASP.NET Identity

Крім валідації пароля в ASP.NET Identity також використовується валідація користувача. Якщо ми перейдемо в файл `IdentityConfig.cs` в папці `App_Start`, то у визначенні класу `ApplicationUserManager` ми знайдемо валідацію користувача:

```

var manager = new ApplicationUserManager
    (new UserStore<ApplicationUser>
        (context.Get<ApplicationDbContext>()));

//.....

manager.UserValidator = new UserValidator<ApplicationUser>(manager)
{
    AllowOnlyAlphanumericUserNames = false,
    RequireUniqueEmail = true
};

```

Для валідації користувача використовується вбудований клас `UserValidator`, який має такі властивості:

- **AllowOnlyAlphanumericUserNames**: якщо одно `true`, то юзернейм повинен містити тільки алфавітно-цифрові символи
- **RequireUniqueEmail**: якщо одно `true`, то email користувача повинен бути унікальним

Можливості вбудованого валідатора досить обмежені, але ми можемо перевизначити його поведінку. Для цього нам треба успадкувати клас від `UserValidator <ApplicationUser>`. Тому, створимо наступний клас:

```

public class CustomUserValidator : UserValidator<ApplicationUser>
{
    public CustomUserValidator(ApplicationUserManager mgr)
        : base(mgr)
    {
        AllowOnlyAlphanumericUserNames = false;
    }
    public override async Task<IdentityResult> ValidateAsync(ApplicationUser user)
    {
        IdentityResult result = await base.ValidateAsync(user);
        if (user.Email.ToLower().EndsWith("@spam.com"))
        {
            var errors = result.Errors.ToList();
            errors.Add("Даний домен знаходиться в спам-базі. Виберіть інший поштовий сервіс");
            result = new IdentityResult(errors);
        }
        if (user.UserName.Contains("admin"))
        {
            var errors = result.Errors.ToList();
            errors.Add("Нік користувача не повинен містити слово 'admin'");
            result = new IdentityResult(errors);
        }
        return result;
    }
}

```

У валідаторі ми можемо використовувати ті ж властивості `AllowOnlyAlphanumericUserNames` і `RequireUniqueEmail`. Зокрема, в конструкторі встановлюється `AllowOnlyAlphanumericUserNames = false`

Щоб написати свою логіку валідації, треба перевизначити метод `ValidateAsync`. У нашому випадку він перевіряє значення полів `UserName` і `Email` і встановлює повідомлення про помилки, якщо ці значення не задовольняють деяким умовам. Результатом роботи методу є об'єкт `IdentityResult`, який містить всі виниклі помилки. В класі `ApplicationUserManager` змінити використовуваний валідатор користувача:

```
var manager = new ApplicationUserManager  
    (new UserStore<ApplicationUser>  
        (context.Get<ApplicationDbContext>()));  
manager.UserValidator = new CustomUserValidator(manager);
```

Для цілей тестування ми визначили додаткову властивість `Name` в моделі реєстрації і додали поле для його введення в представлення реєстрації. В контролері `AccountController` при створенні нового користувача застосовується ця властивість для `UserName`:

```
public async Task<ActionResult> Register(RegisterViewModel model)  
{  
    if (ModelState.IsValid)  
    {  
        var user = new ApplicationUser { UserName = model.Name, Email = model.Email };  
        //.....
```

І тепер, якщо ми ввели некоректні значення, додаток відобразить нам помилки.

3.12.10 Підтвердження реєстрації по Email

Як правило, більшість великих сервісів при реєстрації вимагають її підтвердження по email. Подивимося, як зробити подібний механізм в ASP.NET MVC. У ASP.NET MVC є кілька моделей авторизації і аутентифікації. В даному випадку використовуємо механізм авторизації `AspNet Identity`, так як він більш складний в плані розуміння. При використанні інших методів аутентифікації / авторизації загальна схема підтвердження реєстрації по email буде аналогічна. Отже, створимо новий проект ASP.NET MVC 5 з типом аутентифікації `Individual User Accounts`. Він за замовчуванням містить весь інструментарій для авторизації: моделі, контролер і представлення. Перейдемо в папку моделей і відкриємо файл `IdentityModels.cs`, який містить модель `ApplicationUser`. Змінимо цю модель на наступну:

```
public class ApplicationUser : IdentityUser  
{  
    public string Email { get; set; }  
    public bool ConfirmedEmail { get; set; }
```

```
}
```

До стандартного визначення моделі ми додали властивість для email і властивість для зберігання значення, підтверджена чи реєстрація. Відповідно нам треба змінити модель реєстрації RegisterViewModel в файлі AccountViewModels.cs, щоб вона приймала додаткове поле для email:

```
public class RegisterViewModel
{
    [Required]
    [Display(Name = "User name")]
    public string UserName { get; set; }

    [Required]
    [Display(Name = "Email")]
    public string Email { get; set; }

    [Required]
    [StringLength(100, ErrorMessage = "The {0} must be at least {2} characters long.",
    MinimumLength = 6)]
    [DataType(DataType.Password)]
    [Display(Name = "Password")]
    public string Password { get; set; }

    [DataType(DataType.Password)]
    [Display(Name = "Confirm password")]
    [Compare("Password", ErrorMessage = "The password and confirmation password do not
    match.")]
    public string ConfirmPassword { get; set; }
}
```

І також додамо поле для email в представлення для реєстрації Register.html:

```
<div class="form-group">
    @Html.LabelFor(m => m.Email, new { @class = "col-md-2 control-label" })
    <div class="col-md-10">
        @Html.TextBoxFor(m => m.Email, new { @class = "form-control" })
    </div>
</div>
```

Тепер контролер AccountController, який і здійснює основну роботу. Ми повинні відправляти листи, тому підключимо в нього необхідний простір імен:

```
using System.Net.Mail;
```

Тепер додамо в AccountController дві нових дії і змінимо наступним чином дію Register, яка обробляє post-запити:

```
[HttpPost]
```



```

[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Register(RegisterViewModel model)
{
    if (ModelState.IsValid)
    {
        var user = new ApplicationUser() { UserName = model.UserName };
        user.Email = model.Email;
        user.ConfirmedEmail = false;
        var result = await UserManager.CreateAsync(user, model.Password);
        if (result.Succeeded)
        {
            // наш email с заголовком листа
            MailAddress from = new MailAddress("somemail@gmail.com", "Web Registration");
            // кому відправляємо
            MailAddress to = new MailAddress(user.Email);
            // створюємо об'єкт повідомлення
            MailMessage m = new MailMessage(from, to);
            // тема листа
            m.Subject = "Email confirmation";
            // текст листа - включаємо в нього лінк
            m.Body = string.Format("Для завершення реєстрації перейдіть по лінку:" +
                "<a href=\"{0}\" title=\"Підтвердити реєстрацію\">{0}</a>",
                Url.Action("ConfirmEmail", "Account", new { Token = user.Id, Email = user.Email },
                Request.Url.Scheme));
            m.IsBodyHtml = true;
            // адрес smtp-сервера, з якого ми будемо відправляти листа
            SmtpClient smtp = new System.Net.Mail.SmtpClient("smtp.yandex.ru", 25);
            // логін и пароль
            smtp.Credentials = new System.Net.NetworkCredential("somemail@gmail.com ",
                "password");
            smtp.Send(m);
            return RedirectToAction("Confirm", "Account", new { Email = user.Email });
        }
        else
        {
            AddErrors(result);
        }
    }
    return View(model);
}

```

```

[AllowAnonymous]
public string Confirm(string Email)
{
    return "На поштову адресу " + Email + " Вам надіслані подальші" +
        "інструкції для завершення реєстрації";
}

```

```

[AllowAnonymous]
public async Task<ActionResult> ConfirmEmail(string Token, string Email)

```

```

{
    ApplicationUser user = this.UserManager.FindById(Token);
    if (user != null)
    {
        if (user.Email == Email)
        {
            user.ConfirmedEmail = true;
            await UserManager.UpdateAsync(user);
            await SignInAsync(user, isPersistent: false);
            return RedirectToAction("Index", "Home", new { ConfirmedEmail = user.Email });
        }
        else
        {
            return RedirectToAction("Confirm", "Account", new { Email = user.Email });
        }
    }
    else
    {
        return RedirectToAction("Confirm", "Account", new { Email = "" });
    }
}

```

У порівнянні зі стандартним визначенням методу Register тут додана логіка відправлення листа стандартними механізмами .NET, які знаходяться в просторі імен System.Net.Mail. Після відправки відвідувач отримує на свою адресу лист з посиланням, по якому йому треба пройти. Це посилання вказує на метод ConfirmEmail.

Після переходу в методі ConfirmEmail оновлюється дані користувача в базі даних: для поля ConfirmedEmail встановлюється значення true, що означає, що користувач підтвердив реєстрацію.

Тепер треба також змінити метод Login, що обробляє post-запит:

```

[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Login(LoginViewModel model, string returnUrl)
{
    if (ModelState.IsValid)
    {
        var user = await UserManager.FindAsync(model.UserName, model.Password);
        if (user != null)
        {
            if (user.ConfirmedEmail == true)
            {
                await SignInAsync(user, model.RememberMe);
                return RedirectToLocal(returnUrl);
            }
            else
            {
                ModelState.AddModelError("", "Не підтверджений email.");
            }
        }
    }
}

```

```

    }
    else
    {
        ModelState.AddModelError("", "Невірний логін или пароль");
    }
}
return View(model);
}

```

Тепер при логін буде враховуватися, чи підтвердив раніше користувач свій email.

3.12.11 Підтвердження Email в ASP.NET Identity 2.0

У попередньому розділі розглядався загальний механізм підтвердження реєстрації електронною поштою, головним чином, стосовно ASP.NET Identity 1.0. У Identity 2.0 цей процес буде трохи відрізнятися, хоча суть буде та ж.

Отже, створимо новий проект ASP.NET MVC 5 з типом аутентифікації Individual User Accounts. Він за замовчуванням містить весь інструментарій для авторизації: моделі, контролер і представлення.

Тепер якщо ми використовуємо в проекті ASP.NET Identity 2, то модель, що представляє користувача в системі, вже має необхідні властивості: властивість Email для зберігання адреси електронної пошти та булева властивість EmailConfirmed, яке показує, підтверджена чи ні електронна адреса.

У ASP.NET Identity 2 є вже вся необхідна інфраструктура для створення реєстрації через електронну пошту. Так, модель і представлення реєстрації налаштовані на прийняття електронної адреси в якості логіна. Тому перейдемо до контролера AccountController.

У контролері AccountController змінимо метод Register, що обробляє post-запити, таким чином, щоб при реєстрації він міг посилати повідомлення на електронну пошту:

```

[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Register(RegisterViewModel model)
{
    if (ModelState.IsValid)
    {
        var user = new ApplicationUser { UserName = model.Email, Email = model.Email };
        var result = await UserManager.CreateAsync(user, model.Password);
        if (result.Succeeded)
        {
            // генеруємо токен для підтвердження реєстрації
            var code = await UserManager.GenerateEmailConfirmationTokenAsync(user.Id);
            // створюємо лінк для підтвердження
            var callbackUrl = Url.Action("ConfirmEmail", "Account", new { userId = user.Id, code = code },
                protocol: Request.Url.Scheme);
            // відправка листа

```



```

        new UserStore<ApplicationUser>(context.Get<ApplicationDbContext>());
//.....
manager.EmailService = new EmailService();
manager.SmsService = new SmsService();
//.....
}

```

Таким чином, для настройки надсилання повідомлень електронної пошти нам треба налаштувати клас EmailService. Для цього в початок файлу IdentityConfig.cs підключимо простір імен System.Net.Mail, яке надає функціональність для відправки пошти.

Далі змінимо клас EmailService наступним чином:

```

public class EmailService : IIdentityMessageService
{
    public Task SendAsync(IdentityMessage message)
    {
        // настройка логіна, пароля відправника
        var from = "somemail@yandex.ru";
        var pass = "password12";

        // адреса і порт smtp-сервера, з якого ми і будем відправляти листа
        SmtpClient client = new SmtpClient("smtp.yandex.ru", 25);

        client.DeliveryMethod = SmtpDeliveryMethod.Network;
        client.UseDefaultCredentials = false;
        client.Credentials = new System.Net.NetworkCredential(from, pass);
        client.EnableSsl = true;

        // створюємо лист: message.Destination - адреса отримувача
        var mail = new MailMessage(from, message.Destination);
        mail.Subject = message.Subject;
        mail.Body = message.Body;
        mail.IsBodyHtml = true;

        return client.SendMailAsync(mail);
    }
}

```

Тут проводиться стандартна для .NET відправка листа в асинхронному варіанті. В результаті користувач отримає лист з посиланням. Тепер треба також змінити метод Login, що обробляє post-запит:

```

[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Login(LoginViewModel model, string returnUrl)
{
    if (ModelState.IsValid)
    {
        var user = await UserManager.FindAsync(model.Email, model.Password);
    }
}

```

```

if (user != null)
{
    if (user.EmailConfirmed == true)
    {
        await SignInManager.SignInAsync(user, isPersistent: false, rememberBrowser: false);
        return RedirectToLocal(returnUrl);
    }
    else
    {
        ModelState.AddModelError("", "Не підтверджений email.");
    }
}
else
{
    ModelState.AddModelError("", "Невірний логін або пароль");
}
}
return View(model);
}

```

Тепер при логіні буде враховуватися, чи підтвердив раніше користувач свій email.

3.12.12 Підтвердження телефону по SMS в ASP.NET Identity 2.0

ASP.NET MVC 5 підтримує також підтвердження реєстрації по sms. Але щоб включити цю функціональність в додатку, нам треба зв'язати наш додаток з сервісом, який буде відправляти sms через спеціальний API. Одним з таких сервісів є **Twilio**.

Отже, створимо проект і насамперед підключимо через NuGet пакет, який дозволить взаємодіяти з цим сервісом:

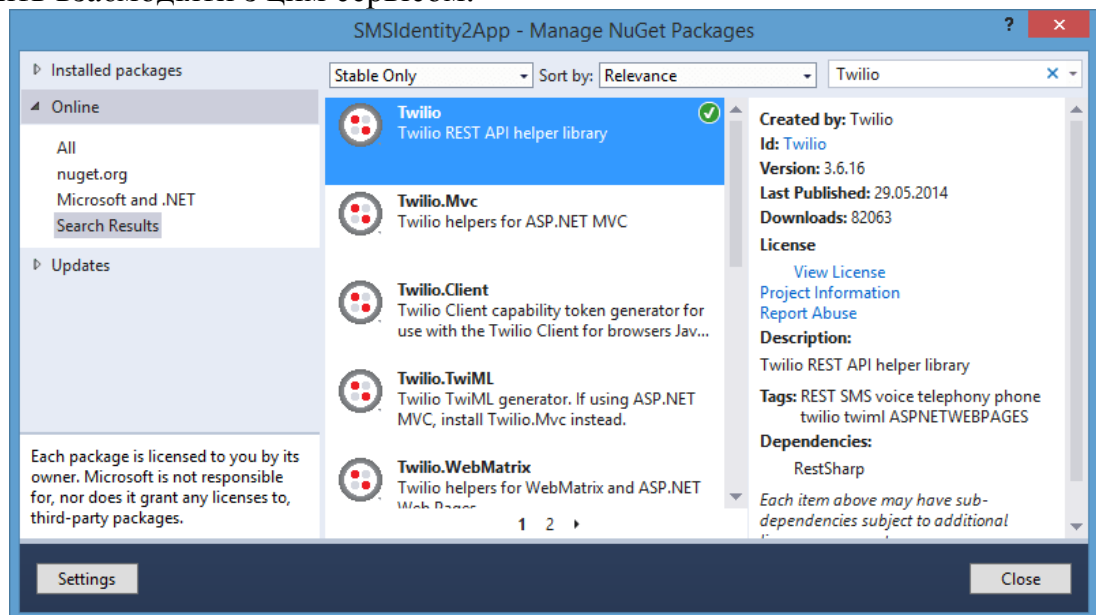


Рисунок 3.67 - Підключення сервісу Twilio

Тепер треба зареєструватися на самому сервісі і отримати ідентифікатор облікового запису ACCOUNT SID, токен AUTH TOKEN і прив'язаний до додатка номер телефону, з якого будуть відправлятися sms.

Значення ACCOUNT SID і AUTH TOKEN можна знайти в особистому кабінеті:

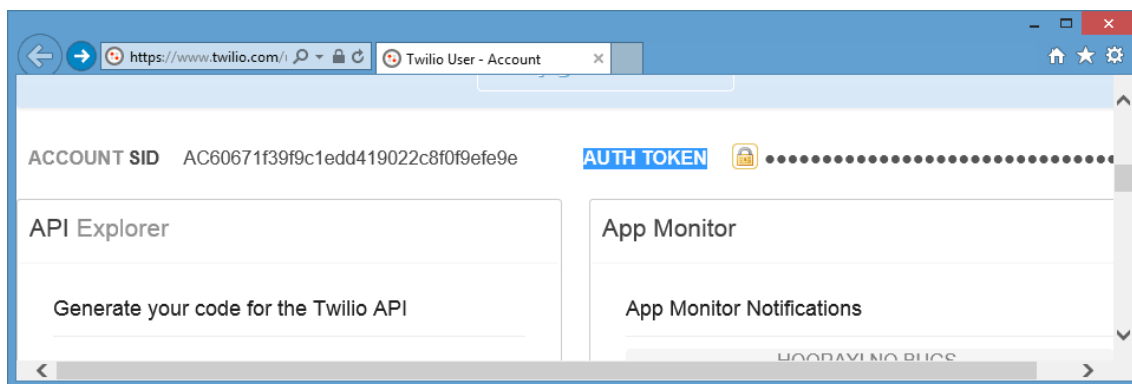


Рисунок 3.68 - Одержання ACCOUNT SID і AUTH TOKEN

Через пункт меню Numbers можна побачити номер телефону, прив'язаний до аккаунту :

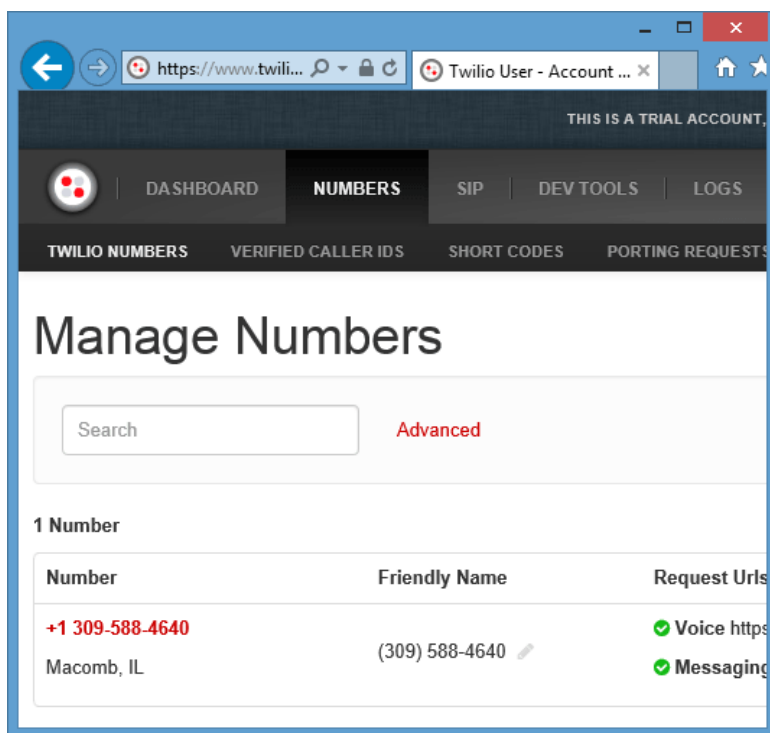


Рисунок 3.69 – Керування номерами

Також нам потрібно підключити всі ті країни, з телефонними кодами яких ми будемо працювати. Це можна зробити в особистому кабінеті в меню Geographic Permissions на вкладці Messaging. Тут нам треба зазначити всі необхідні країни:

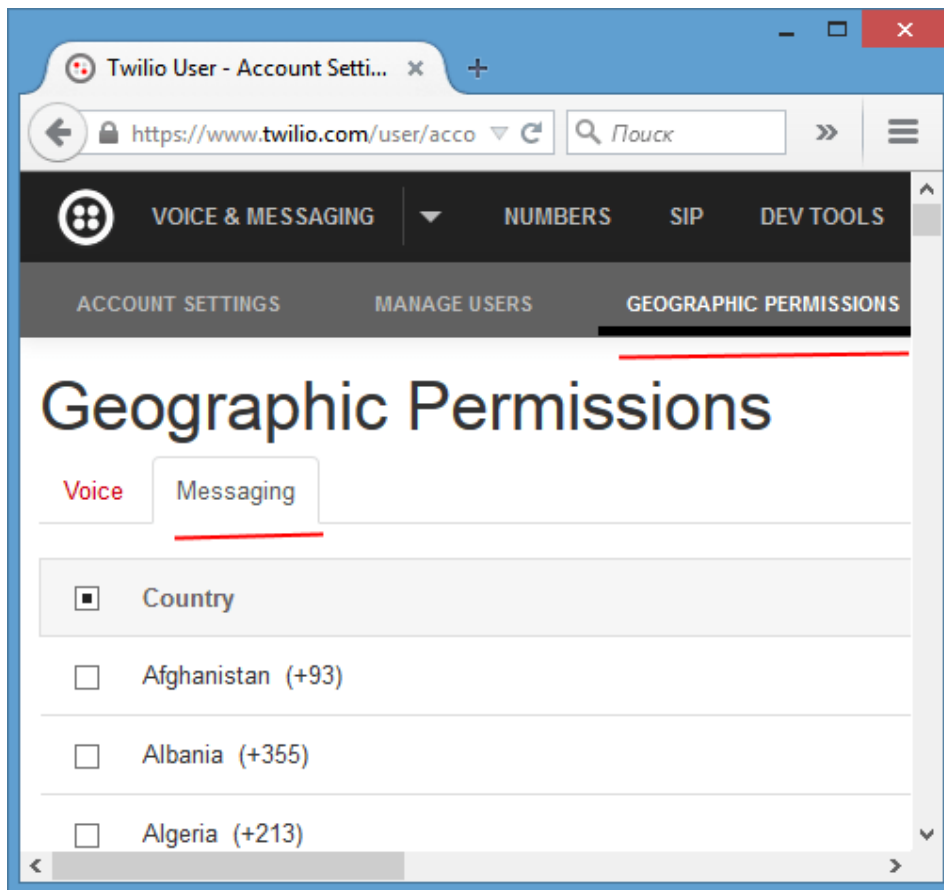


Рисунок 3.70 – Визначення дозволених для роботи країн

Після створення облікового запису в Twilio насамперед підключимо цей сервіс до себе в додаток. Для цього перейдемо в файл `IdentityConfig.cs`, який розташовується в папці `App_Start`. У ньому вже приготований спеціальний клас для відправки sms, але поки він по суті нічого не робить:

```
public class SmsService : IIdentityMessageService
{
    public Task SendAsync(IdentityMessage message)
    {
        // Plug in your sms service here to send a text message.
        return Task.FromResult(0);
    }
}
```

Там же в файлі в методі `Create` відбувається установка цього сервісу для менеджера облікових записів: `manager.SmsService = new SmsService ();`

І тепер підключимо в початок файлу функціональність twilio: `using Twilio;` і змінимо код сервісу наступним чином:

```
public class SmsService : IIdentityMessageService
{
    public Task SendAsync(IdentityMessage message)
    {
        string AccountSid = "AC60671f39f9c1edd419022c8f0f9efe9e";
```



```

string AuthToken = "f62e1374032b4b2838d07f085b678d15";

string twilioPhoneNumber = "+13095884640";

var twilio = new TwilioRestClient(AccountSid, AuthToken);
twilio.SendMessage(twilioPhoneNumber, message.Destination, message.Body);

return Task.FromResult(0);
}

```

Змінна AccountSid буде зберігати ідентифікатор вашого облікового запису ACCOUNT SID, змінна AuthToken містить токен AUTH TOKEN, а twilioPhoneNumber - номер телефону, прив'язаного до вашого аккаунту. В даному випадку всі ці значення з мого аккаунта, для вашого облікового запису Twilio вони будуть відрізнятися.

Тепер треба створити інфраструктуру для відправки sms і верифікації отриманих через sms кодів. Модель користувача ApplicationUser вже має властивість PhoneNumber, яке представляє телефон користувача, а також властивість PhoneNumberConfirmed, яке вказує, чи був підтверджений номер телефону. Однак безпосередньо з цими властивостями при підтвердженні ми не будемо взаємодіяти, а будемо звертатися через спеціальні методи. Для цього перейдемо до контролера ManageController. Він уже має дві пари методів, які нам знадобляться для роботи. По-перше, метод AddPhoneNumber, який приймає номер телефону і відправляє на нього код:

```

public ActionResult AddPhoneNumber()
{
    return View();
}

[HttpPost]
[ValidateAntiForgeryToken]
public async Task<ActionResult> AddPhoneNumber(AddPhoneNumberViewModel model)
{
    if (!ModelState.IsValid)
    {
        return View(model);
    }
    var code = await
        UserManager.GenerateChangePhoneNumberTokenAsync(User.Identity.GetUserId(),
            model.Number);
    if (UserManager.SmsService != null)
    {
        var message = new IdentityMessage
        {
            Destination = model.Number,
            Body = "Your security code is: " + code
        };
        await UserManager.SmsService.SendAsync(message);
    }
}

```

```

    }
    return RedirectToAction("VerifyPhoneNumber", new { PhoneNumber = model.Number });
}

```

По-друге, це метод Verify Phone Number, який верифікує код:

```

public async Task<ActionResult> VerifyPhoneNumber(string phoneNumber)
{
    var code = await
    UserManager.GenerateChangePhoneNumberTokenAsync(User.Identity.GetUserId(),
    phoneNumber);
    return phoneNumber == null ? View("Error") : View(new VerifyPhoneNumberViewModel {
    PhoneNumber = phoneNumber });
}
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<ActionResult> VerifyPhoneNumber(VerifyPhoneNumberViewModel model)
{
    if (!ModelState.IsValid)
    {
        return View(model);
    }
    var result = await UserManager.ChangePhoneNumberAsync(User.Identity.GetUserId(),
    model.PhoneNumber, model.Code);
    if (result.Succeeded)
    {
        var user = await UserManager.FindByIdAsync(User.Identity.GetUserId());
        if (user != null)
        {
            await SignInManager.SignInAsync(user, isPersistent: false, rememberBrowser: false);
        }
        return RedirectToAction("Index", new { Message = ManageMessageId.AddPhoneSuccess });
    }
    ModelState.AddModelError("", "Failed to verify phone");
    return View(model);
}

```

Метод AddPhoneNumber в GET-версії повертає представлення, яке призначене для введення телефону. А POST-версія методу приймає введений номер у вигляді моделі AddPhoneNumberViewModel і на основі цього номера генерує код:

```

var code = await UserManager.GenerateChangePhoneNumberTokenAsync (User.Identity.GetUserId
(), model.Number);

```

Потім формується повідомлення: `var message = new IdentityMessage {Destination = model.Number, Body = "Your security code is:" + code} ;`. Це повідомлення відправляється `await UserManager.SmsService.SendAsync (message);` а фактично передається вищевизначеному класу `SmsService` для безпосередньої відправки. Після цього користувач перенаправляється на сторінку введення коду, а на телефон користувача приходять код. Після введення коду дані обробляються `post`-методом `VerifyPhoneNumber`, в якому за

допомогою виклику `var result = await UserManager.ChangePhoneNumberAsync (User.Identity.GetUserId(), model.PhoneNumber, model.Code);` введений код порівнюється з токеном з БД. Якщо порівняння пройшло успішно, то `result.Succeeded` поверне значення `true`. Для даних методів в файлі `ManageViewModels.cs` вже визначені всі необхідні моделі:

```
public class AddPhoneNumberViewModel
{
    [Required]
    [Phone]
    [Display(Name = "Phone Number")]
    public string Number { get; set; }
}

public class VerifyPhoneNumberViewModel
{
    [Required]
    [Display(Name = "Code")]
    public string Code { get; set; }

    [Required]
    [Phone]
    [Display(Name = "Phone Number")]
    public string PhoneNumber { get; set; }
}
```

А в папці `Views / Manage` вже є представлення `Add Phone Number.cshtml` і `VerufyPhoneNumber.cs`. Запустимо додаток, реєструватимемося і перейдемо до додавання номера за посиланням `Manage / AddPhoneNumber`:

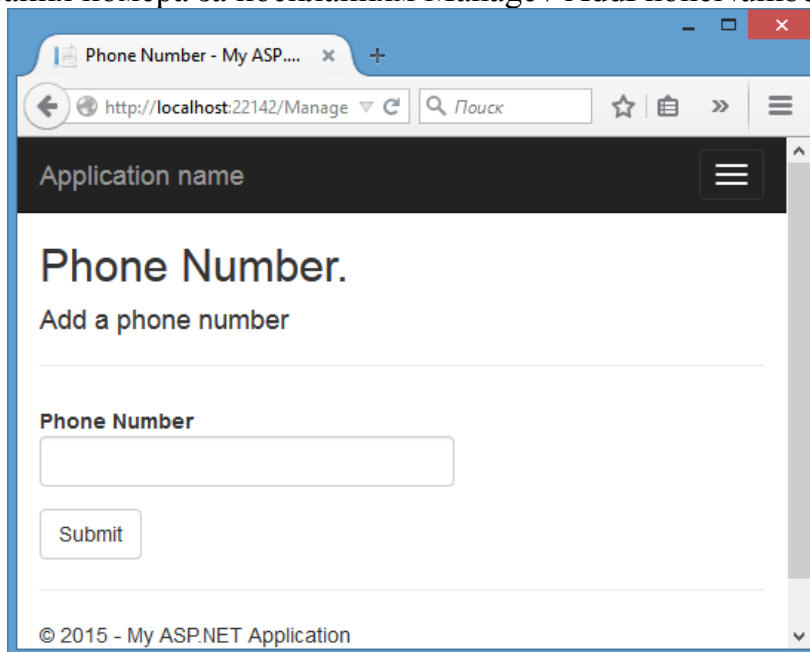


Рисунок 3.71 - Додавання номера

Номер вводиться з плюсом, наприклад, `+38062221133`. Після введення номера ми отримаємо SMS на телефон наступного виду :

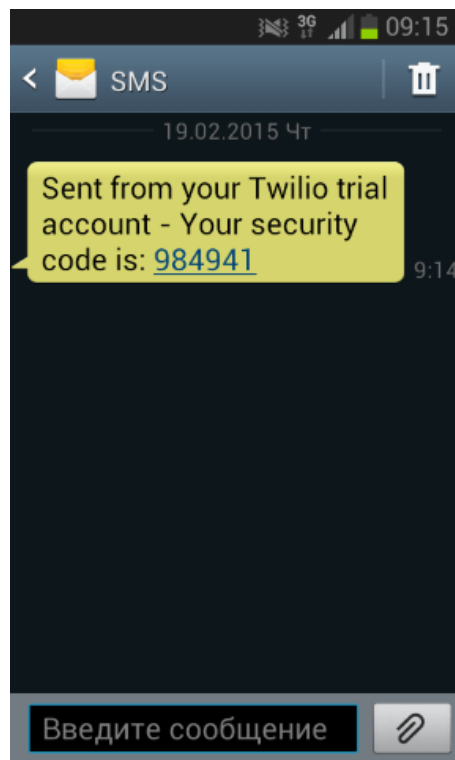


Рисунок 3.72 – Одержання СМС підтвердження

Введемо отриманий код:

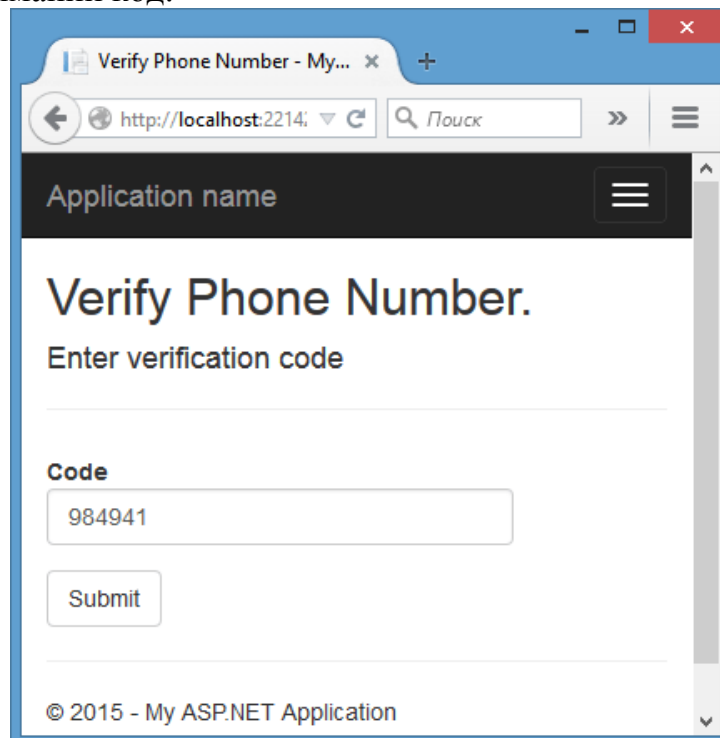


Рисунок 3.73 – Підтвердження одержаного коду

І після успішної перевірки нам відобразиться відповідна сторінка:

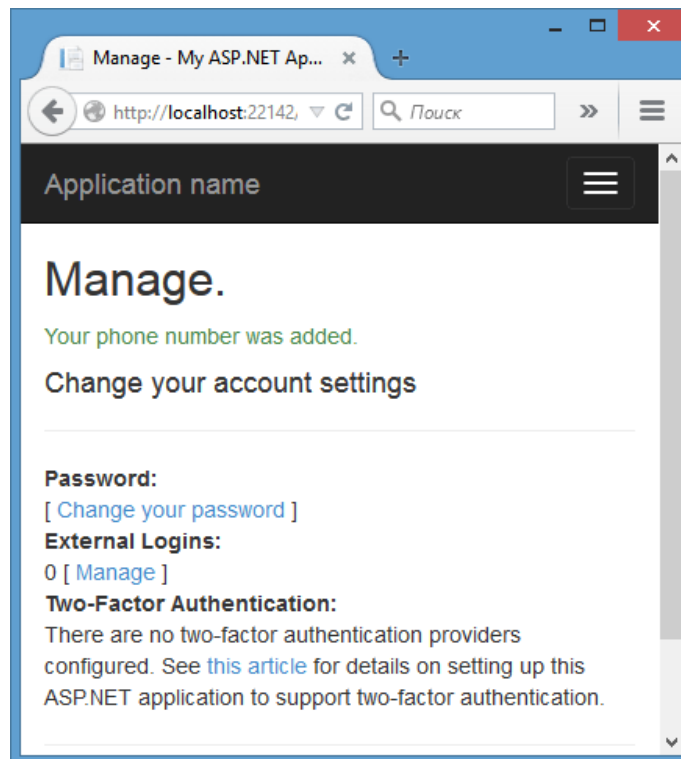


Рисунок 3.74 - Управління реєстраційними даними

Все, ми пройшли перевірку. Тепер в базі даних для даного користувача в поле `PhoneNumber` буде значитися номер телефону, і в програмі ми його зможемо отримати і використовувати: `user.PhoneNumber`

3.12.13 Скидання пароля в ASP.NET Identity 2.0

Розглянемо механізм скидання пароля в ASP.NET Identity. Для Identity 2.0 вже є необхідна інфраструктура для скидання пароля: це представлення і методи контролера. У файлі `AccountViewModels.cs` вже описані моделі, які використовуються для скидання пароля: `ForgotPasswordViewModel` і `ResetPasswordViewModel`, а в контролері `AccountController` визначені методи, необхідні для скидання пароля, зокрема, методи `ResetPassword` і `ForgotPassword`. Для скидання пароля ми будемо звертатися до методу `ForgotPassword`, який буде викликати представлення `ForgotPassword.cshtml`. Зазначене представлення містить форму для введення електронної адреси, який буде використовуватися для скидання пароля.

Дані форми обробляються `post`-версією методу `ForgotPassword`, який за замовчуванням не містить функціонал для скидання пароля. Тому змінимо його наступним чином:

```
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<ActionResult> ForgotPassword(ForgotPasswordViewModel model)
{
    if (ModelState.IsValid)
    {
```

```

var user = await UserManager.FindByNameAsync(model.Email);
if (user == null || !(await UserManager.IsEmailConfirmedAsync(user.Id)))
{
    return View("ForgotPasswordConfirmation");
}
string code = await UserManager.GeneratePasswordResetTokenAsync(user.Id);
var callbackUrl = Url.Action("ResetPassword", "Account",
    new { userId = user.Id, code = code }, protocol: Request.Url.Scheme);
await UserManager.SendEmailAsync(user.Id, "Сброс пароля",
    "Для скидання пароля, перейдіть за посиланням <a href=\"" + callbackUrl +
    "\">сбросить</a>");
return RedirectToAction("ForgotPasswordConfirmation", "Account");
}
return View(model);
}

```

Після відправлення листа користувач перенаправляється до методу `Forgot Password Confirmation`. Всі інші дії щодо скидання пароля виконує метод `ResetPassword`, стандартна реалізація якого в принципі не вимагає зміни. Але щоб була можливою відправка електронних повідомлень, повинен бути налаштований відповідним чином сервіс відправки пошти. Для цього перейдемо до файлу `IdentityConfig.cs` і знайдемо в ньому клас `EmailService`, який і буде відправляти електронні повідомлення. Змінимо його наступним чином:

```

public class EmailService : IdentityMessageService
{
    public Task SendAsync(IdentityMessage message)
    {
        // настройка логіна, пароля відправника
        var from = "somemail@yandex.ru";
        var pass = "password";

        // адреса і порт smtp-сервера, з якого ми і будемо відправляти листа
        SmtpClient client = new SmtpClient("smtp.yandex.ru", 25);

        client.DeliveryMethod = SmtpDeliveryMethod.Network;
        client.UseDefaultCredentials = false;
        client.Credentials = new System.Net.NetworkCredential(from, pass);
        client.EnableSsl = true;

        // створюємо лист: message.Destination - адреса отримувача
        var mail = new MailMessage(from, message.Destination);
        mail.Subject = message.Subject;
        mail.Body = message.Body;
        mail.IsBodyHtml = true;
        return client.SendMailAsync(mail);
    }
}

```

Отже, припустимо, у нас в системі вже є зареєстрований з поштовою адресою користувач. Спробуємо скинути пароль і звернемося до методу ForgotPassword:

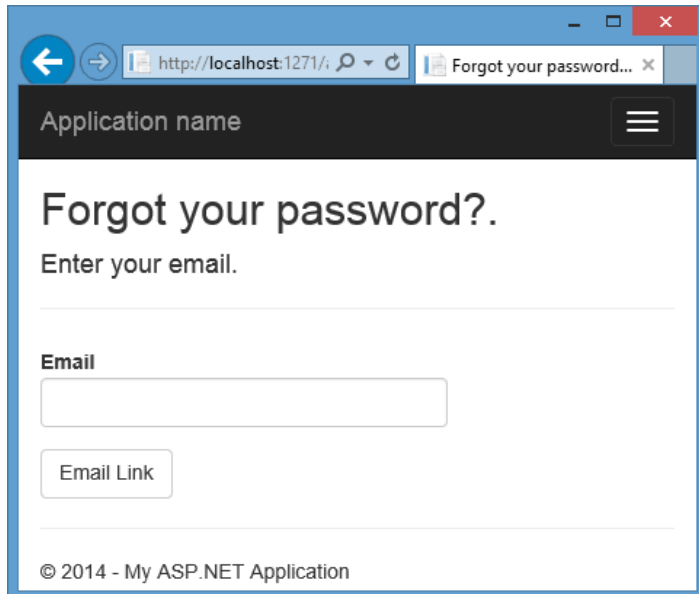


Рисунок 3.75 - Форма відновлення пароля

І після цього ми будемо перенаправлені за методом Forgot Password Confirmation:

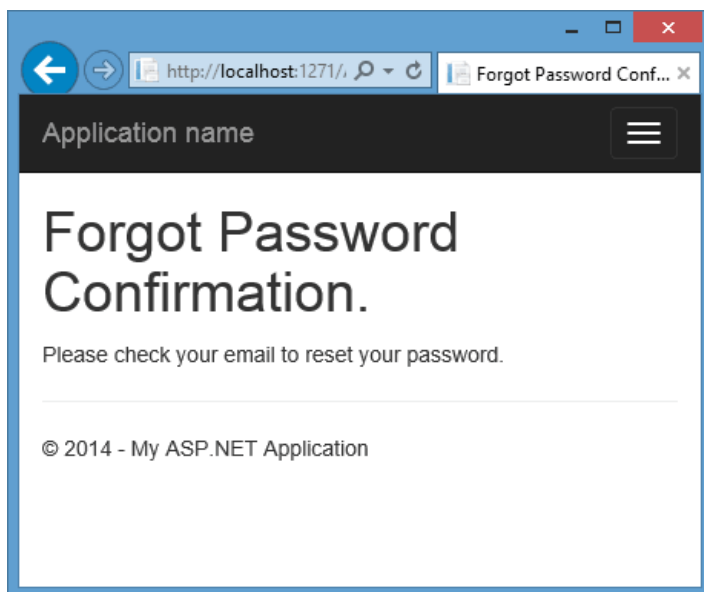


Рисунок 3.76 - Підтвердження відправлення інструкцій на пошту

Тим часом на вказану електронну пошту приходить лист з посиланням. Після переходу за цим посиланням спрацьовує метод ResetPassword, і ми опиняємося на сторінці скидання пароля:

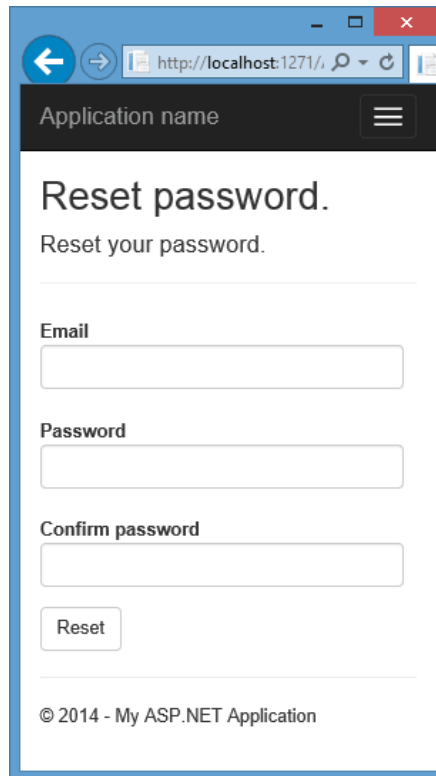


Рисунок 3.77 – Сторінка скидання пароля

Вказуємо ту ж пошту і новий пароль і відправляємо форму на сервер. Після цього post-метод `ResetPassword` обробляє дані і перенаправляє нас на метод `ResetPasswordConfirmation`:

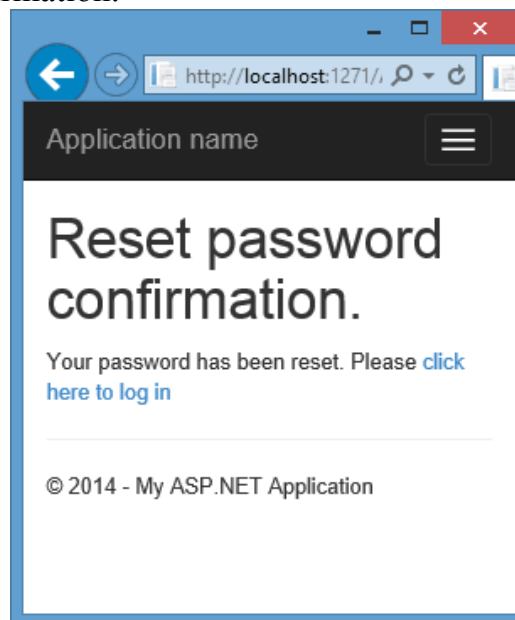


Рисунок 3.78 - Підтвердження скидання пароля

І після переустановлення пароля ми можемо заходити на сайт під тією ж електронною адресою і новим паролем.

3.12.14 Універсальні провайдери в MVC 5

Крім стандартної системи ASP.NET Identity в MVC 5 також можна використовувати і більш традиційну для попередніх версій систему

аутентифікації і авторизації - універсальні провайдери. Універсальні провайдери є провайдерами членства, ролей, профілів, сесії, які мають певну реалізацію за замовчуванням. Але при бажанні ми можемо їх перевизначити.

У проектах MVC 5 універсальні провайдери за замовчуванням відсутні, тому їх треба додавати через NuGet менеджер. Отже, створимо проект MVC 5 без аутентифікації і додамо через NuGet пакет Microsoft.AspNet.Providers:

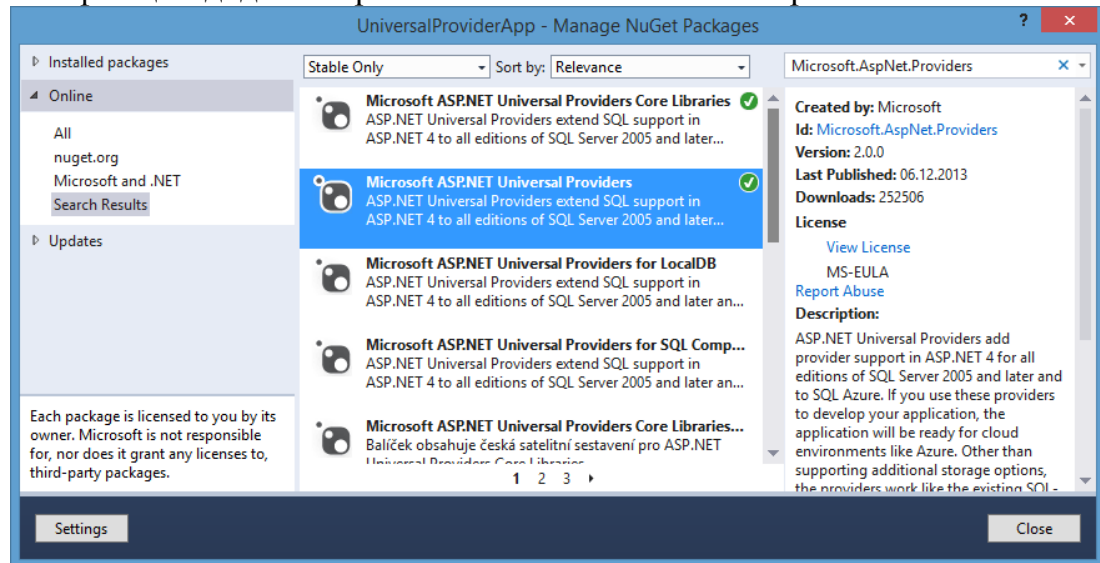


Рисунок 3.79 - Додавання універсальних провайдерів

Після установки в проекті в файлі web.config відбудеться кілька змін. По-перше, будуть додані визначення провайдерів в вузлі system.web:

```
<profile defaultProvider="DefaultProfileProvider">
  <providers>
    <add name="DefaultProfileProvider" type="System.Web.Providers.DefaultProfileProvider,
System.Web.Providers, Version=2.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35"
connectionStringName="DefaultConnection" applicationName="/" />
  </providers>
</profile>
<membership defaultProvider="DefaultMembershipProvider">
  <providers>
    <add name="DefaultMembershipProvider"
type="System.Web.Providers.DefaultMembershipProvider, System.Web.Providers, Version=2.0.0.0,
Culture=neutral, PublicKeyToken=31bf3856ad364e35" connectionStringName="DefaultConnection"
enablePasswordRetrieval="false" enablePasswordReset="true"
requiresQuestionAndAnswer="false" requiresUniqueEmail="false"
maxInvalidPasswordAttempts="5" minRequiredPasswordLength="6"
minRequiredNonalphanumericCharacters="0" passwordAttemptWindow="10"
applicationName="/" />
  </providers>
</membership>
<roleManager defaultProvider="DefaultRoleProvider">
  <providers>
    <add name="DefaultRoleProvider" type="System.Web.Providers.DefaultRoleProvider,
System.Web.Providers, Version=2.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35"
connectionStringName="DefaultConnection" applicationName="/" />
  </providers>
</roleManager>
```

```

</providers>
</roleManager>
<sessionState mode="InProc" customProvider="DefaultSessionProvider">
  <providers>
    <add name="DefaultSessionProvider"
type="System.Web.Providers.DefaultSessionStateProvider, System.Web.Providers, Version=2.0.0.0,
Culture=neutral, PublicKeyToken=31bf3856ad364e35" connectionStringName="DefaultConnection"
/>
  </providers>
</sessionState>

```

Також буде додано рядок підключення. І крім того, щоб задіяти універсальні провайдери, нам треба додати в вузол `system.web` підтримку аутентифікації форм:

```

<authentication mode="Forms">
  <forms name="cookies" loginUrl="~/Account/Login" />
</authentication>

```

І тепер за аутентифікацію і авторизацію відповідатимуть універсальні провайдери. Тепер додамо інфраструктуру для нашого проекту. На початку додамо моделі логіна і реєстрації в папку `Mod`:

```

public class LogOnModel
{
  [Required]
  [Display(Name = "Логін")]
  public string UserName { get; set; }

  [Required]
  [DataType(DataType.Password)]
  [Display(Name = "Пароль")]
  public string Password { get; set; }

  [Display(Name = "Запам'ятати")]
  public bool RememberMe { get; set; }
}

public class RegisterModel
{
  [Required]
  [Display(Name = "Логін")]
  public string UserName { get; set; }

  [Required]
  [DataType(DataType.EmailAddress)]
  [Display(Name = "Електронна пошта")]
  public string Email { get; set; }

  [Required]

```

```

[DataType(DataType.Password)]
[Display(Name = "Пароль")]
public string Password { get; set; }

[DataType(DataType.Password)]
[Display(Name = "Підтвердіть пароль")]
[Compare("Password", ErrorMessage = "Паролі не співпадають.")]
public string ConfirmPassword { get; set; }
}

```

Також додамо в папку Controllers контролер AccountController наступного вмісту:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Security;
using UniversalProviderApp.Models;

namespace UniversalProviderApp.Controllers
{
    [AllowAnonymous]
    public class AccountController : Controller
    {
        public ActionResult Login()
        {
            return View();
        }

        [HttpPost]
        public ActionResult Login(LogOnModel model, string returnUrl)
        {
            if (ModelState.IsValid)
            {
                if (Membership.ValidateUser(model.UserName, model.Password))
                {
                    FormsAuthentication.SetAuthCookie(model.UserName, model.RememberMe);
                    if (Url.IsLocalUrl(returnUrl))
                    {
                        return Redirect(returnUrl);
                    }
                    else
                    {
                        return RedirectToAction("Index", "Home");
                    }
                }
            }
            else
            {

```

```

        ModelState.AddModelError("", "Неправильний пароль або логін");
    }
}
return View(model);
}

public ActionResult LogOff()
{
    FormsAuthentication.SignOut();
    return RedirectToAction("Login", "Account");
}

public ActionResult Register()
{
    return View();
}

[HttpPost]
public ActionResult Register(RegisterModel model)
{
    if (ModelState.IsValid)
    {
        MembershipCreateStatus createStatus;
        Membership.CreateUser(model.UserName, model.Password, model.Email,
            passwordQuestion: null, passwordAnswer: null, isApproved: true,
            providerUserKey: null, status: out createStatus);

        if (createStatus == MembershipCreateStatus.Success)
        {
            FormsAuthentication.SetAuthCookie(model.UserName, false);
            return RedirectToAction("Index", "Home");
        }
        else
        {
            ModelState.AddModelError("", "Помилка при реєстрації");
        }
    }
    return View(model);
}
}
}

```

При реєстрації користувача за допомогою методу `Membership.Create User` новий користувач додається в БД, а потім здійснюємо вхід за допомогою методу `FormsAuthentication.SetAuthCookie()`, який встановлює куки. При логіні метод `Membership.ValidateUser` перевіряє, чи є в базі даних користувач з введеними логіном і паролем в системі. Тепер створимо необхідні представлення. Представлення реєстрації `Register.cshtml`:

@model UniversalProviderApp.Models.RegisterModel

```

@{
    ViewBag.Title = "Реєстрація";
}

<h2> Реєстрація</h2>

@using (Html.BeginForm()) {
    @Html.ValidationSummary(true)

    <fieldset>
        <legend>Введіть дані</legend>

        <div class="editor-label">
            @Html.LabelFor(model => model.UserName)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.UserName)
            @Html.ValidationMessageFor(model => model.UserName)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.Email)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Email)
            @Html.ValidationMessageFor(model => model.Email)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.Password)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Password)
            @Html.ValidationMessageFor(model => model.Password)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.ConfirmPassword)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.ConfirmPassword)
            @Html.ValidationMessageFor(model => model.ConfirmPassword)
        </div>

        <p>
            <input type="submit" value="Зареєструвати" />
        </p>
    </fieldset>
}

```

І представлення логіна Login.cshtml:

```
@model UniversalProviderApp.Models.LogOnModel
@{
    ViewBag.Title = "Вхід в систему";
}
<h2>Вхід в систему</h2>

@using (Html.BeginForm()) {
    @Html.ValidationSummary(true)

    <fieldset>
        <legend>Введіть логін и пароль</legend>

        <div class="editor-label">
            @Html.LabelFor(model => model.UserName)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.UserName)
            @Html.ValidationMessageFor(model => model.UserName)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.Password)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Password)
            @Html.ValidationMessageFor(model => model.Password)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.RememberMe)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.RememberMe)
            @Html.ValidationMessageFor(model => model.RememberMe)
        </div>

        <p>
            <input type="submit" value="Ввійти" />
        </p>
    </fieldset>
}
```

Після запуску фреймворк створить в БД ряд таблиць, призначених для зберігання облікових записів, ролей та іншої інформації, пов'язаної з членством:

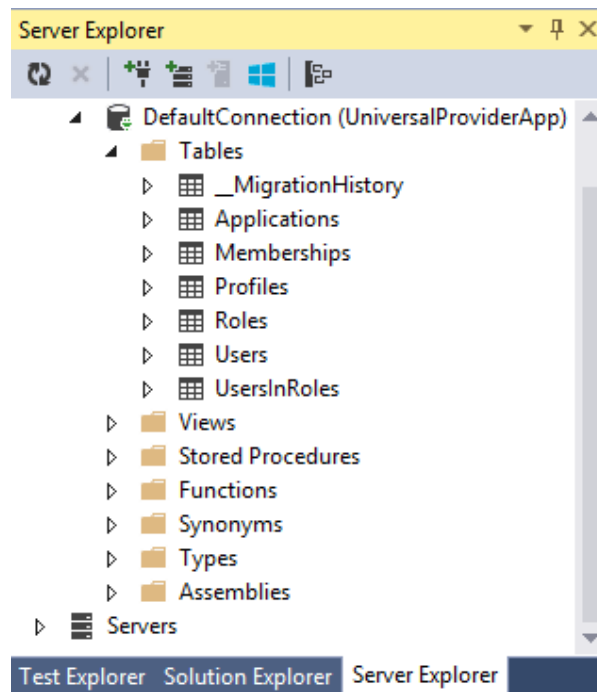


Рисунок 3.80 – Структура бази даних

Зареєструємо нового користувача в онлайн-формі, після чого інформація про користувача потрапить в БД.

3.12.15 Перевизначення універсальних провайдерів. Провайдер ролей

Універсальні провайдери надають уже готовий функціонал авторизації. Але в той же час ці провайдери володіють достатньою гнучкістю - зокрема ми можемо їх перевизначити на свій розсуд. При цьому нам необов'язково перевизначати і використовувати всі чотири провайдера, що досить зручно, особливо в ситуації, коли нам не потрібні всі можливості ASP.NET Identity, а потрібно побудувати дуже просту систему авторизації.

Отже, перевизначимо провайдер ролей. Для цього спочатку створимо новий проект і додамо в нього через NuGet пакет Microsoft.AspNet.Providers і Entity Framework.

Насамперед визначимо моделі, які будуть описувати облікові записи. Це будуть моделі користувачів і ролей, а також контекст даних для доступу до них. Тому додамо в папку Models наступні класи:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.ComponentModel.DataAnnotations;
using System.Data.Entity;

namespace CustomRoleProviderApp.Models
{
    public class UserContext : DbContext
    {

```

```

    public UserContext() : base("DefaultConnection")
    {
    }
    public DbSet<User> Users { get; set; }
    public DbSet<Role> Roles { get; set; }
}

public class User
{
    public int Id { get; set; }
    public string Email { get; set; }
    public string Password { get; set; }
    public int RoleId { get; set; }
    public Role Role { get; set; }
}

public class Role
{
    public int Id { get; set; }
    public string Name { get; set; }
}

public class UserDbInitializer : DropCreateDatabaseAlways<UserContext>
{
    protected override void Seed(UserContext db)
    {
        db.Roles.Add(new Role { Id = 1, Name = "admin" });
        db.Roles.Add(new Role { Id = 2, Name = "user" });
        db.Users.Add(new User
        {
            Id=1,
            Email="somemail@gmail.com",
            Password="123456",
            RoleId=1
        });
        base.Seed(db);
    }
}
}

```

Тут визначено клас користувача і клас ролей. Клас користувача містить посилання на виконувачу роль в системі. І також визначено контекст UserContext, через який ми будемо отримувати дані з БД. Контекст даних зв'язується з підключення до БД, яке автоматично було додано в файл web.config при установці пакету Microsoft.AspNet.Providers. При бажанні можна визначити будь-яке інше підключення.

Крім того, щоб ініціалізувати базу даних початковими даними, через клас ініціалізатор UserDbInitializer ми додаємо пару ролей і одного користувача в базу даних. І щоб ініціалізатор спрацював, додамо його виклик в файл global.asax:

```

public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()

```



```

{
    Database.SetInitializer(new UserDbInitializer());
    //.....
}
}

```

Тепер визначимо сам провайдер ролей. Для цього спочатку додамо в проект папку Providers і потім в нього додамо новий клас CustomRoleProvider:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Security;
using CustomRoleProviderApp.Models;

namespace CustomRoleProviderApp.Providers
{
    public class CustomRoleProvider : RoleProvider
    {
        public override string[] GetRolesForUser(string username)
        {
            string[] role = new string[] { };
            using (UserContext db = new UserContext())
            {
                // Одержуємо користувача
                User user = db.Users.FirstOrDefault(u=>u.Email == username);
                if (user != null)
                {
                    // одержуємо роль
                    Role userRole = db.Roles.Find(user.RoleId);
                    if (userRole != null)
                        role = new string[] { userRole.Name };
                }
            }
            return role;
        }
        public override void CreateRole(string roleName)
        {
            Role newRole = new Role() { Name = roleName };
            UserContext db = new UserContext();
            db.Roles.Add(newRole);
            db.SaveChanges();
        }
        public override bool IsUserInRole(string username, string roleName)
        {
            bool outputResult = false;
            // Находимо користувача
            using (UserContext db = new UserContext())
            {
                // Одержуємо користувача

```

```

        User user = db.Users.FirstOrDefault(u=>u.Email == username);
        if (user != null)
        {
            // одержуємо роль
            Role userRole = db.Roles.Find(user.RoleId);
            //порівнюємо
            if (userRole != null && userRole.Name == roleName)
                outputResult = true;
        }
    }
    return outputResult;
}
public override void AddUsersToRoles(string[] usernames, string[] roleNames)
{
    throw new NotImplementedException();
}

public override string ApplicationName
{
    get { throw new NotImplementedException(); }
    set { throw new NotImplementedException(); }
}

public override bool DeleteRole(string roleName, bool throwOnPopulatedRole)
{
    throw new NotImplementedException();
}

public override string[] FindUsersInRole(string roleName, string usernameToMatch)
{
    throw new NotImplementedException();
}

public override string[] GetAllRoles()
{
    throw new NotImplementedException();
}

public override string[] GetUsersInRole(string roleName)
{
    throw new NotImplementedException();
}

public override void RemoveUsersFromRoles(string[] usernames, string[] roleNames)
{
    throw new NotImplementedException();
}

public override bool RoleExists(string roleName)
{
    throw new NotImplementedException();
}

```

```

    }
  }
}

```

З метою демонстрації перевизначимо три методи. Перший з них - `GetRolesForUser` дозволяє отримувати набір ролей для певного користувача. Другий метод - `CreateRole` - дозволяє створення ролі. І третій метод - `IsUserInRole` - визначає, чи виконує користувач певну роль в системі.

Щоб використовувати провайдер ролей в додатку, треба додати його визначення в файл конфігурації. Відкриємо файл `web.config` і видалимо з нього визначення провайдерів, які були додані автоматично при додаванні пакета `Microsoft.AspNet.Providers`:

```

<profile defaultProvider="DefaultProfileProvider">
  <providers>
    <add name="DefaultProfileProvider" type="System.Web.Providers.DefaultProfileProvider,
System.Web.Providers, Version=2.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35"
connectionStringName="DefaultConnection" applicationName="/" />
  </providers>
</profile>
<membership defaultProvider="DefaultMembershipProvider">
  <providers>
    <add name="DefaultMembershipProvider"
type="System.Web.Providers.DefaultMembershipProvider, System.Web.Providers, Version=2.0.0.0,
Culture=neutral, PublicKeyToken=31bf3856ad364e35" connectionStringName="DefaultConnection"
enablePasswordRetrieval="false" enablePasswordReset="true"
requiresQuestionAndAnswer="false" requiresUniqueEmail="false"
maxInvalidPasswordAttempts="5" minRequiredPasswordLength="6"
minRequiredNonalphanumericCharacters="0" passwordAttemptWindow="10"
applicationName="/" />
  </providers>
</membership>
<roleManager defaultProvider="DefaultRoleProvider">
  <providers>
    <add name="DefaultRoleProvider" type="System.Web.Providers.DefaultRoleProvider,
System.Web.Providers, Version=2.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35"
connectionStringName="DefaultConnection" applicationName="/" />
  </providers>
</roleManager>
<sessionState mode="InProc" customProvider="DefaultSessionProvider">
  <providers>
    <add name="DefaultSessionProvider" type="System.Web.Providers.DefaultSessionStateProvider,
System.Web.Providers, Version=2.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35"
connectionStringName="DefaultConnection" />
  </providers>
</sessionState>

```

І замість цього в межах вузла `system.web` додамо наш провайдер:

```

<authentication mode="Forms">

```

```

    <forms name="cookies" loginUrl="~/Account/Login" />
</authentication>
<roleManager enabled="true" defaultProvider="MyRoleProvider">
    <providers>
        <add name="MyRoleProvider" type="CustomRoleProviderApp.Providers.CustomRoleProvider" />
    </providers>
</roleManager>

```

І в кінці додаємо контролер AccountController для логіна і реєстрації:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Security;
using CustomRoleProviderApp.Models;

namespace CustomRoleProviderApp.Controllers
{
    public class AccountController : Controller
    {
        public ActionResult Login()
        {
            return View();
        }
        [HttpPost]
        [ValidateAntiForgeryToken]
        public ActionResult Login(LoginModel model)
        {
            if (ModelState.IsValid)
            {
                // пошук користувача в БД
                User user = null;
                using (UserContext db = new UserContext())
                {
                    user = db.Users.FirstOrDefault(u => u.Email == model.Name && u.Password ==
model.Password);
                }
                if (user != null)
                {
                    FormsAuthentication.SetAuthCookie(model.Name, true);
                    return RedirectToAction("Index", "Home");
                }
                else
                {
                    ModelState.AddModelError("", "Користувача з таким логіном и паролем нема");
                }
            }
            return View(model);
        }
    }
}

```

```

    }

    public ActionResult Register()
    {
        return View();
    }
    [HttpPost]
    [ValidateAntiForgeryToken]
    public ActionResult Register(RegisterModel model)
    {
        if (ModelState.IsValid)
        {
            User user = null;
            using (UserContext db = new UserContext())
            {
                user = db.Users.FirstOrDefault(u => u.Email == model.Name && u.Password ==
model.Password);
            }
            if (user == null)
            {
                using (UserContext db = new UserContext())
                {
                    db.Users.Add(new User { Email = model.Name, Password = model.Password, RoleId=2
});
                    db.SaveChanges();

                    user = db.Users.Where(u => u.Email == model.Name && u.Password ==
model.Password).FirstOrDefault();
                }
                if (user != null)
                {
                    FormsAuthentication.SetAuthCookie(model.Name, true);
                    return RedirectToAction("Index", "Home");
                }
            }
            else
            {
                ModelState.AddModelError("", "Користувач з таким логіном вже існує");
            }
        }
        return View(model);
    }
    public ActionResult Logoff()
    {
        FormsAuthentication.SignOut();
        return RedirectToAction("Index", "Home");
    }
}
}

```

У методі Register при додаванні нового користувача ми йому будемо присвоювати в роль user. Моделі, які використовуються методами Login і Register, виглядають так:

```
public class RegisterModel
{
    [Required]
    public string Name { get; set; }

    [Required]
    [DataType(DataType.Password)]
    public string Password { get; set; }

    [Required]
    [DataType(DataType.Password)]
    [Compare("Password", ErrorMessage = "Паролі не співпадають")]
    public string ConfirmPassword { get; set; }
}

public class LoginModel
{
    [Required]
    public string Name { get; set; }

    [Required]
    [DataType(DataType.Password)]
    public string Password { get; set; }
}
```

І тепер ми можемо розмежувати доступ до методів різних контролерів за допомогою атрибута Authorize:

```
public class HomeController : Controller
{
    [Authorize(Roles = "admin")]
    public ActionResult Index()
    {
        return View();
    }
}
```

3.12.16 Створення програми з ASP.NET Identity «з нуля»

При створенні проекту з типом аутентифікації Individual User Accounts в нього за замовчуванням додаються всі необхідні файли для роботи з AspNet Identity. Однак, як правило, рідко затребуваний весь стандартний функціонал. Якщо нам потрібна лише реєстрація та логін, то інші файли і невикористаний код будуть просто висіти в проекті, або їх доведеться видаляти. Однак ми можемо вибрати будь-який інший тип проекту і в нього вже додати вручну функціонал AspNet Identity, причому тільки той, який нам буде потрібен. Це дасть нам більший контроль над тим кодом, який розміщується в проекті.

Наприклад, створимо звичайний проект MVC з типом аутентифікації No Authentication. Щоб додати в проект ASP.NET Identity, нам треба додати такі NuGet-пакети:

- Microsoft.AspNet.Identity.EntityFramework
- Microsoft.AspNet.Identity.OWIN
- Microsoft.Owin.Host.SystemWeb

Після додавання пакетів треба оновити файл web.config: додамо рядок підключення:

```
<connectionStrings>
  <add name="IdentityDb" providerName="System.Data.SqlClient"
    connectionString="Data
Source=(localdb)\v11.0;AttachDbFilename=|DataDirectory|\IdentityDb.mdf;Integrated
Security=True;" />
</connectionStrings>
```

Тепер додамо класи користувачів і контексту даних в папку Models. Клас контексту буде успадковуватися від IdentityDbContext:

```
using System.Data.Entity;
using Microsoft.AspNet.Identity.EntityFramework;

public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
{
    public ApplicationDbContext() : base("IdentityDb") { }

    public static ApplicationDbContext Create()
    {
        return new ApplicationDbContext();
    }
}
```

Клас користувачів:

```
using Microsoft.AspNet.Identity.EntityFramework;

public class ApplicationUser : IdentityUser
{
    public int Year { get; set; }
    public ApplicationUser()
    {
    }
}
```

У класі користувача крім успадкованих від IdentityUser властивостей також визначається і властивість Year для зберігання року народження користувача.

Оскільки вся робота з користувачами йде не безпосередньо, а через менеджер користувачів, то також додамо в папку Models відповідний клас:

```
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.EntityFramework;
```

```

using Microsoft.AspNet.Identity.Owin;
using Microsoft.Owin;

public class ApplicationUserManager : UserManager<ApplicationUser>
{
    public ApplicationUserManager(IUserStore<ApplicationUser> store)
        : base(store)
    {
    }

    public static ApplicationUserManager Create (IdentityFactoryOptions<ApplicationUserManager>
options, IOwinContext context)
    {
        ApplicationDbContext db = context.Get<ApplicationDbContext>();
        ApplicationUserManager manager = new ApplicationUserManager(new
UserStore<ApplicationUser>(db));
        return manager;
    }
}

```

Клас менеджера користувачів успадковується від UserManager. У конструкторі він приймає об'єкт сховища користувачів IUserStore. А статичний метод Create() створює екземпляр класу ApplicationUserManager за допомогою об'єкта контексту IOwinContext

Останній крок у початковому налаштуванні проекту для AspNet Identity полягає в додаванні в проект файлу запуску програми OWIN. Отже, додамо в папку App_Start файл Startup.cs з наступним вмістом:

```

using Microsoft.Owin;
using Owin;
using AspNetIdentityApp.Models;
using Microsoft.Owin.Security.Cookies;
using Microsoft.AspNet.Identity;
[assembly: OwinStartup(typeof(AspNetIdentityApp.Startup))]
namespace AspNetIdentityApp
{
    public class Startup
    {
        public void Configuration(IAppBuilder app)
        {
            // настраюємо контекст і менеджер
            app.CreatePerOwinContext<ApplicationDbContext>(ApplicationDbContext.Create);
            app.CreatePerOwinContext<ApplicationUserManager>(ApplicationUserManager.Create);
            app.UseCookieAuthentication(new CookieAuthenticationOptions
            {
                AuthenticationType = DefaultAuthenticationTypes.ApplicationCookie,
                LoginPath = new PathString("/Account/Login"),
            });
        }
    }
}

```


Інтерфейс `IAppBuilder` визначає множину методів, однак в даному випадку нам достатньо трьох методів. Метод `CreatePerOwinContext` реєструє в OWIN менеджер користувачів `ApplicationUserManager` і клас контексту `ApplicationContext`. Метод `UseCookieAuthentication` за допомогою об'єкта `CookieAuthenticationOptions` встановлює аутентифікацію на основі куки в якості типу аутентифікації в додатку. А властивість `LoginPath` дозволяє встановити адресу URL, за яким будуть перенаправлятися неавторизовані користувачі.

Як правило, це адреса `/ Account / Login`.

Це мінімально необхідна настройка проекту для використання `AspNet Identity`, на основі якої ми вже зможемо створювати всю решту системи авторизації та аутентифікації.

3.12.17 Реєстрація та створення користувачів в ASP.NET Identity

Продовжимо роботу з проектом і додамо в нього функціонал реєстрації користувачів. Клас `IdentityUser` визначає багато властивостей, проте нам не обов'язково всіх їх встановлювати. І в даному випадку для створення користувачів нам краще скористатися сполучною моделлю, яка встановить всі необхідні властивості. Отже, додамо в папку `Models` клас, який буде представляти користувача:

```
using System;
using System.ComponentModel.DataAnnotations;

namespace AspNetIdentityApp.Models
{
    public class RegisterModel
    {
        [Required]
        public string Email { get; set; }

        [Required]
        public int Year { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }

        [Required]
        [Compare("Password", ErrorMessage = "Паролі не співпадають")]
        [DataType(DataType.Password)]
        public string PasswordConfirm { get; set; }
    }
}
```

Тепер додамо в папку `Controllers` новий контролер `AccountController`:

```
using System;
using System.Collections.Generic;
```

```

using System.Linq;
using System.Threading.Tasks;
using System.Web;
using System.Web.Mvc;
using AspNetIdentityApp.Models;
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.Owin;

namespace AspNetIdentityApp.Controllers
{
    public class AccountController : Controller
    {
        private ApplicationUserManager UserManager
        {
            get
            {
                return HttpContext.GetOwinContext().GetUserManager<ApplicationUserManager>();
            }
        }

        public ActionResult Register()
        {
            return View();
        }

        [HttpPost]
        public async Task<ActionResult> Register(RegisterModel model)
        {
            if (ModelState.IsValid)
            {
                ApplicationUser user = new ApplicationUser { UserName = model.Email, Email =
model.Email, Year = model.Year };
                IdentityResult result = await UserManager.CreateAsync(user, model.Password);
                if (result.Succeeded)
                {
                    return RedirectToAction("Login", "Account");
                }
                else
                {
                    foreach (string error in result.Errors)
                    {
                        ModelState.AddModelError("", error);
                    }
                }
            }
            return View(model);
        }
    }
}

```

Насамперед в контролері створюється властивість User Manager, яка повертає об'єкт ApplicationUserManager. Через нього ми будемо взаємодіяти зі сховищем користувачів. Для отримання сховища застосовується вираз HttpContext.GetOwinContext(). GetUserManager <ApplicationUserManager>().

Для власне реєстрації визначено метод Register в Get- і Post-версіях. Post-версія являє асинхронний метод, оскільки для створення користувача тут використовується асинхронний виклик UserManager.CreateAsync(). Цей метод повертає об'єкт IdentityResult.

Якщо створення користувача пройшло успішно, то його властивість Succeeded дорівнюватиме true.

І в кінці створимо подання для реєстрації Register.cshtml:

```
@model AspNetIdentityApp.Models.RegisterModel
```

```
@{  
    ViewBag.Title = "Register";  
}
```

```
<h2>Регистрация</h2>
```

```
@using (Html.BeginForm())
```

```
{  
    @Html.AntiForgeryToken()
```

```
<div>  
    <h4>Реєстрація користувача</h4>  
    <hr />  
    @Html.ValidationSummary(false)
```

```
<div>  
    Електронна адреса  
    <div>  
        @Html.EditorFor(model => model.Email)  
    </div>  
</div>
```

```
<div>  
    Рік народження  
    <div>  
        @Html.EditorFor(model => model.Year)  
    </div>  
</div>
```

```
<div>  
    Пароль  
    <div>  
        @Html.EditorFor(model => model.Password)  
    </div>
```

```

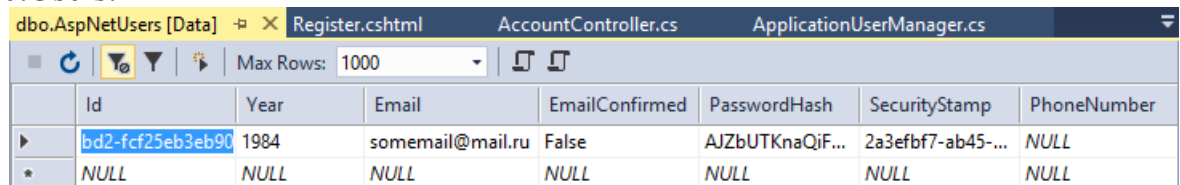
</div>

<div>
    Підтвердіть пароль
    <div>
        @Html.EditorFor(model => model.PasswordConfirm)
    </div>
</div>

<div>
    <div>
        <input type="submit" value="Зареєструвати" />
    </div>
</div>
</div>
}

```

І після введення даних дані про користувача потрапляють в таблицю `AspNetUsers`:



	Id	Year	Email	EmailConfirmed	PasswordHash	SecurityStamp	PhoneNumber
▶	bd2-fcf25eb3eb90	1984	somemail@mail.ru	False	AJZbUTKnaQiF...	2a3efbf7-ab45-...	NULL
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Рисунок 3.81 - Структура таблиці бази даних `AspNetUsers`

3.12.18 Авторизація користувачів в ASP.NET Identity

Для створення інфраструктури для входу користувачів на сайт спочатку додамо в проект в папку `Models` спеціальну модель `LoginModel`:

```

public class LoginModel
{
    [Required]
    public string Email { get; set; }
    [Required]
    [DataType(DataType.Password)]
    public string Password { get; set; }
}

```

У минулій темі у нас вже був доданий контролер `AccountController`. Тепер же додамо в нього наступні рядки:

```

private IAuthenticationManager AuthenticationManager
{
    get
    {
        return HttpContext.GetOwinContext().Authentication;
    }
}

```

```

public ActionResult Login(string returnUrl)
{
    ViewBag.returnUrl = returnUrl;
    return View();
}

[HttpPost]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Login(LoginModel model, string returnUrl)
{
    if (ModelState.IsValid)
    {
        ApplicationUser user = await UserManager.FindAsync(model.Email, model.Password);
        if (user == null)
        {
            ModelState.AddModelError("", "Невірний логін або пароль.");
        }
        else
        {
            ClaimsIdentity claim = await UserManager.CreateIdentityAsync(user,
                DefaultAuthenticationTypes.ApplicationCookie);
            AuthenticationManager.SignOut();
            AuthenticationManager.SignIn(new AuthenticationProperties
            {
                IsPersistent = true
            }, claim);
            if (String.IsNullOrEmpty(returnUrl))
                return RedirectToAction("Index", "Home");
            return Redirect(returnUrl);
        }
    }
    ViewBag.returnUrl = returnUrl;
    return View(model);
}

public ActionResult Logout()
{
    AuthenticationManager.SignOut();
    return RedirectToAction("Login");
}

```

Спочатку визначається нова властивість, що представляє об'єкт `AuthenticationManager`, за допомогою якої ми будемо управляти входом на сайт. Для цього інтерфейс `IAuthenticationManager` визначає два методи:

- `SignIn()`: створює аутентифікаційні куки
- `SignOut()`: видаляє аутентифікаційні куки

У `Get`-версії методу `Login` ми отримуємо адресу для повернення і передаємо його в представлення. В `Post`-версії методу `Login` отримуємо дані з представлення у вигляді моделі `LoginModel` і по ньому намагаємося отримати користувача з БД за допомогою методу `UserManager.FindAsync(model.Email,`

model.Password). Цей метод повертає об'єкт ApplicationUser в разі успіху пошуку.

AspNet Identity використовує аутентифікацію на основі об'єктів **claim**. Тому нам треба спочатку створити об'єкт ClaimsIdentity, який представляє реалізацію інтерфейсу IIdentity в AspNet Identity. Для створення ClaimsIdentity застосовується метод CreateIdentityAsync ()

І на фінальному етапі викликається метод AuthenticationManager.SignIn(), в який передається об'єкт конфігурації аутентифікації AuthenticationProperties, а також раніше створений об'єкт ClaimsIdentity. Властивість IsPersistent дозволяє зберігати аутентифікаційні дані в браузері навіть після закриття користувачем браузера. Метод Logout видаляє аутентифікаційні куки в браузері, ніби роблячи вихід із системи. Представлення логіна може виглядати так:

```
@model AspNetIdentityApp.Models.LoginModel

@{
    ViewBag.Title = "Login";
}
<h2>Вхід на сайт</h2>
@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()

    <div>
        @Html.ValidationSummary()
        <input type="hidden" name="returnUrl" value="@ViewBag.returnUrl" />
        <p>
            <label>Email</label><br />
            @Html.EditorFor(x => x.Email)
        </p>
        <p>
            <label>Пароль</label><br />
            @Html.EditorFor(x => x.Password)
        </p>
        <p><button type="submit">Ввійти</button></p>
    </div>
}
```

3.12.19 Редагування і видалення користувачів

Редагування і видалення користувачів в ASP.NET Identity представляється досить простим завданням завдяки класу UserManager. Так, візьмемо проект з попередньої теми, де у нас вже є механізм реєстрації і логіна користувачів. Додамо в контролер AccountController наступні методи:

```
[HttpGet]
public ActionResult Delete()
{
    return View();
}
```

```
[HttpPost]
[ActionName("Delete")]
```

```

public async Task<ActionResult> DeleteConfirmed()
{
    ApplicationUser user = await UserManager.FindByEmailAsync(User.Identity.Name);
    if (user != null)
    {
        IdentityResult result = await UserManager.DeleteAsync(user);
        if (result.Succeeded)
        {
            return RedirectToAction("Logout", "Account");
        }
    }
    return RedirectToAction("Index", "Home");
}

public async Task<ActionResult> Edit()
{
    ApplicationUser user = await UserManager.FindByEmailAsync(User.Identity.Name);
    if (user != null)
    {
        EditModel model = new EditModel { Year = user.Year };
        return View(model);
    }
    return RedirectToAction("Login", "Account");
}

[HttpPost]
public async Task<ActionResult> Edit(EditModel model)
{
    ApplicationUser user = await UserManager.FindByEmailAsync(User.Identity.Name);
    if (user != null)
    {
        user.Year = model.Year;
        IdentityResult result = await UserManager.UpdateAsync(user);
        if (result.Succeeded)
        {
            return RedirectToAction("Index", "Home");
        }
        else
        {
            ModelState.AddModelError("", "Щось пішло не так...");
        }
    }
    else
    {
        ModelState.AddModelError("", "Користувача не знайдено");
    }
    return View(model);
}

```

Методи Delete і Delete Confirmed відображають користувачеві представлення для видалення і приймають вибір користувача про видалення. Для видалення використовується метод UserManager.DeleteAsync(). Він повертає об'єкт IdentityResult, який дозволяє відстежити успіх операції.

Метод Edit також відображає представлення для редагування, передаючи в нього модель EditModel, яку ми далі створюємо. В даному випадку ми редагуємо тільки значення властивості Year. POST-версія методу приймає дані моделі і встановлює значення її властивостей для користувача. Редагування також проводиться одним методом - методом UserManager.UpdateAsync(). Далі створимо модель EditModel:

```
public class EditModel
{
    public int Year { get; set; }
}
```

Нехай редагування буде проводитися тільки для року народження користувача. Представлення для видалення могло б виглядати так:

```
@{
    ViewBag.Title = "Delete";
}
<form method="post">
    <h2>Вы впевнені, що хочете видалити свій акаунт?</h2>
    <button type="submit" >Да</button>
</form>
```

І представлення для редагування:

```
@model AspNetIdentityApp.Models.EditModel

@{
    ViewBag.Title = "Edit";
}

<h2>Редагувати дані профіля</h2>

@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()

    <div>
        @Html.ValidationSummary(true, "")
        <div>
            <p>Year: @Html.EditorFor(model => model.Year)</p>
        </div>
        <div>
            <div>
                <input type="submit" value="Зберегти" />
            </div>
        </div>
    </div>
}
```



```

        </div>
    </div>
</div>
}

```

3.12.20 Додавання ролей в ASP.NET Identity

Сутність ролей в ASP.NET Identity представлена класом `IdentityRole`, який реалізує інтерфейс `IRole`. І ми можемо продовжувати використовувати `IdentityRole`, але при необхідності також можемо перевизначити, додавши в нього нові властивості. Отже, візьмемо проект з минулого теми і додамо в нього функціональність управління ролями.

Для початку додамо в папку `Models` новий клас `ApplicationRole`:

```

using Microsoft.AspNet.Identity.EntityFramework;
public class ApplicationRole: IdentityRole
{
    public ApplicationRole(){}

    public string Description { get; set; }
}

```

Новий клас успадковує весь функціонал від `IdentityRole` плюс додає нову властивість `Description`, яке буде містити опис ролі. Для управління ролями використовується менеджер ролей `RoleManager`. Тому додамо в папку `Models` новий клас `ApplicationRoleManager`:

```

class ApplicationRoleManager : RoleManager<ApplicationRole>
{
    public ApplicationRoleManager(RoleStore<ApplicationRole> store)
        : base(store)
    {}
    public static ApplicationRoleManager Create(IdentityFactoryOptions<ApplicationRoleManager>
options,
        IOwinContext context)
    {
        return new ApplicationRoleManager(new
            RoleStore<ApplicationRole>(context.Get<ApplicationContext>()));
    }
}

```

Знову успадковуємо функціонал від уже наявного класу `RoleManager`. Метод `Create` дозволить класу додатка OWIN створювати екземпляри менеджера ролей для обробки кожного запиту, де йде звернення до сховища ролей `RoleStore`. Після цього нам треба зареєструвати менеджера ролей в класі програми OWIN. Тому відкриємо файл `Startup.cs`, який по попередніх темах у нас знаходився в папці `App_Start`, і змінимо його вміст в такий спосіб:

```

public class Startup
{

```

```

public void Configuration(IAppBuilder app)
{
    app.CreatePerOwinContext<ApplicationContext>(ApplicationContext.Create);
    app.CreatePerOwinContext<ApplicationUserManager>(ApplicationUserManager.Create);

    // реєстрація менеджера ролей
    app.CreatePerOwinContext<ApplicationRoleManager>(ApplicationRoleManager.Create);

    app.UseCookieAuthentication(new CookieAuthenticationOptions
    {
        AuthenticationType = DefaultAuthenticationTypes.ApplicationCookie,
        LoginPath = new PathString("/Account/Login"),
    });
}
}

```

Завдяки реєстрації менеджер ролей буде використовувати той же контекст даних, що і менеджер користувачів. Якщо ми створюємо систему ролей в додатку після першого запуску програми, то нам треба буде зробити міграцію, так як контекст даних у нас змінився в силу зміни системи ролей. Тому в Visual Studio в вікні Package Manager Console введемо команду:

enable-migrations і натиснемо Enter.

Тепер нам треба створити саму міграцію. Там же в консолі Package Manager Console введемо команду:

```
PM> Add-Migration "DataMigration"
```

Visual Studio автоматично генерує клас міграції:

```

public partial class DataMigration : DbMigration
{
    public override void Up()
    {
        AddColumn("dbo.AspNetRoles", "Description", c => c.String());
        AddColumn("dbo.AspNetRoles", "Discriminator", c => c.String(nullable: false, maxLength: 128));
    }

    public override void Down()
    {
        DropColumn("dbo.AspNetRoles", "Discriminator");
        DropColumn("dbo.AspNetRoles", "Description");
    }
}

```

І щоб виконати міграцію, застосуємо цей клас, набравши в тій же консолі команду:

```
PM> Update-Database
```

Тепер створимо контролер, який буде виконувати стандартні дії з ролями:

```
using AspNetIdentityApp.Models;
```

```

using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.Owin;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using System.Web;
using System.Web.Mvc;

public class RolesController : Controller
{
    private ApplicationRoleManager RoleManager
    {
        get
        {
            return HttpContext.GetOwinContext().GetUserManager<ApplicationRoleManager>();
        }
    }

    public ActionResult Index()
    {
        return View(RoleManager.Roles);
    }

    public ActionResult Create()
    {
        return View();
    }

    [HttpPost]
    public async Task<ActionResult> Create(CreateRoleModel model)
    {
        if (ModelState.IsValid)
        {
            IdentityResult result = await RoleManager.CreateAsync(new ApplicationRole
            {
                Name = model.Name,
                Description = model.Description
            });
            if (result.Succeeded)
            {
                return RedirectToAction("Index");
            }
            else
            {
                ModelState.AddModelError("", "Щось пішло не так");
            }
        }
        return View(model);
    }

    public async Task<ActionResult> Edit(string id)

```

```

{
    ApplicationRole role = await RoleManager.FindByIdAsync(id);
    if (role != null)
    {
        return View(new EditRoleModel { Id = role.Id, Name = role.Name, Description =
role.Description });
    }
    return RedirectToAction("Index");
}

[HttpPost]
public async Task<ActionResult> Edit(EditRoleModel model)
{
    if (ModelState.IsValid)
    {
        ApplicationRole role = await RoleManager.FindByIdAsync(model.Id);
        if (role != null)
        {
            role.Description = model.Description;
            role.Name = model.Name;
            IdentityResult result = await RoleManager.UpdateAsync(role);
            if (result.Succeeded)
            {
                return RedirectToAction("Index");
            }
        }
        else
        {
            ModelState.AddModelError("", " Щось пішло не так ");
        }
    }
    return View(model);
}

public async Task<ActionResult> Delete(string id)
{
    ApplicationRole role = await RoleManager.FindByIdAsync(id);
    if (role != null)
    {
        IdentityResult result = await RoleManager.DeleteAsync(role);
    }
    return RedirectToAction("Index");
}
}

```

Це стандартний CRUD-контролер, який виконує читання, видалення, редагування і додавання ролей. Спочатку для взаємодії з менеджером ролей ми отримуємо його об'єкт з контексту OWIN:

```
HttpContext.GetOwinContext().GetUserManager<ApplicationRoleManager>()
```

Потім методи менеджера ролей використовуються для управління ролями. Зверніть увагу, що з представлень в методи Create і Edit ми отримуємо не об'єкт `ApplicationRole`, а спеціальні моделі `EditRoleModel` і `CreateRoleModel`, який буде виглядати так:

```
public class EditRoleModel
{
    public string Id { get; set; }
    public string Name { get; set; }
    public string Description { get; set; }
}
public class CreateRoleModel
{
    public string Name { get; set; }
    public string Description { get; set; }
}
```

Використання моделей дозволить уникнути різних проблем з контекстом даних і управлінням об'єктами, які могли виникнути, якби ми безпосередньо використовували `ApplicationRole`. Представлення будуть виглядати стандартно. Представлення `Index.cshtml`:

```
@model IEnumerable<AspNetIdentityApp.Models.ApplicationRole>
```

```
@{
    ViewBag.Title = "Ролі";
}
```

```
<h2>Ролі</h2>
<table class="table">
    <tr>
        <th>Назва</th>
        <th>Опис</th>
        <th></th>
    </tr>
```

```
@foreach (var item in Model) {
    <tr>
        <td>@item.Name</td>
        <td>@item.Description</td>
        <td>
            @Html.ActionLink("Edit", "Edit", new { id=item.Id }) |
            @Html.ActionLink("Delete", "Delete", new { id=item.Id })
        </td>
    </tr>
}
</table>
```

Представлення `Create.cshtml`:

```

@model AspNetIdentityApp.Models.CreateRoleModel
<h2>Додавання ролі</h2>
@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()
    <div>
        @Html.ValidationSummary(true, "")
        <p>Назва: @Html.EditorFor(model => model.Name)</p>
        <p>Опис: @Html.EditorFor(model => model.Description)</p>
        <p><input type="submit" value="Додати" /></p>
    </div>
}

```

І представлення для редагування Edit.cshtml:

```

@model AspNetIdentityApp.Models.EditRoleModel
<h2>Редагування ролі</h2>
@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()

    <div>
        @Html.ValidationSummary(true, "")
        @Html.HiddenFor(model => model.Id)
        <p>Название: @Html.EditorFor(model => model.Name)</p>
        <p>Описание: @Html.EditorFor(model => model.Description)</p>
        <p><input type="submit" value="Змінити" /></p>
    </div>
}

```

3.12.21 Аутентифікація OWIN і ClaimsIdentity

У цій статті розглянемо, як ми можемо аутентифікувати користувача і розмежувати доступ за ролями без всяких провайдерів і ASP.NET Identity засобами інфраструктури OWIN. І для цього спочатку створимо новий проект без аутентифікації. Перш за все, нам необхідні моделі користувача і ролі, а також контекст даних. Тому додамо в папку Models наступні класи:

```

public class User
{
    public int Id { get; set; }
    public string Email { get; set; }
    public string Password { get; set; }
    public Role Role { get; set; }
    public int RoleId { get; set; }
}
public class Role
{
    public int Id { get; set; }
    public string Name { get; set; }
}

```

```
}
```

І також додамо клас контексту даних і ініціалізатор:

```
public class UserContext : DbContext
{
    public UserContext():base("DefaultConnection")
    {}
    public DbSet<User> Users { get; set; }
    public DbSet<Role> Roles { get; set; }
}

public class UserDbInitializer : DropCreateDatabaseAlways<UserContext>
{
    protected override void Seed(UserContext db)
    {
        db.Roles.Add(new Role { Id = 1, Name = "admin" });
        db.Roles.Add(new Role { Id = 2, Name = "user" });
        db.Users.Add(new User
        {
            Email = "alice@gmail.com",
            Password = "123456",
            RoleId = 1
        });
        db.Users.Add(new User
        {
            Email = "tom@gmail.com",
            Password = "123456",
            RoleId = 2
        });
        base.Seed(db);
    }
}
```

Нехай у файлі web.config у нас буде визначений рядок підключення DefaultConnection, а в файлі Global.asax викликається вищевизначений ініціалізатор БД. Тепер нам треба підключити OWIN в проект. Для цього додамо через Nuget наступні пакети:

```
Microsoft.Owin.Host.SystemWeb
Microsoft.Owin.Security.Cookies
```

Вони мають ряд залежностей, і разом з ними будуть додані пакети OWIN, Microsoft.Owin і Microsoft.Owin.Security. Потім додамо в проект в папку App_Start клас OWIN, який назовемо Startup:

```
using Microsoft.Owin;
using Owin;
using Microsoft.Owin.Security.Cookies;
```

```
[assembly: OwinStartup(typeof(IdentityUoWApp.Startup))]
namespace IdentityUoWApp
{
    public class Startup
    {
        public void Configuration(IApplicationBuilder app)
        {
            app.UseCookieAuthentication(new CookieAuthenticationOptions
            {
                AuthenticationType = "ApplicationCookie",
                LoginPath = new PathString("/Account/Login"),
            });
        }
    }
}
```

За допомогою методу розширення `UseCookieAuthentication()` тут встановлюється, що при аутентифікації користувачів будуть використовуватися куки: `AuthenticationType = "ApplicationCookie"`. Тепер зробимо функціонал для входу на сайт. Спочатку додамо в проект папку `ViewModels` і в ній визначимо клас `LoginViewModel` - модель входу на сайт:

```
using System.ComponentModel.DataAnnotations;
public class LoginModel
{
    [Required]
    public string Email { get; set; }
    [Required]
    [DataType(DataType.Password)]
    public string Password { get; set; }
}
```

Потім додамо в папку `Controllers` новий контролер `AccountController`:

```
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Data.Entity;
using System.Security.Claims;
using Microsoft.Owin.Security;
using IdentityUoWApp.ViewModels; // простір імен LoginViewModel
using IdentityUoWApp.Models; // простір імен моделей
using System.Threading.Tasks;

namespace IdentityUoWApp.Controllers
{
    public class AccountController : Controller
    {
        UserContext db = new UserContext();
    }
}
```



```

private IAuthenticationManager AuthenticationManager
{
    get
    {
        return HttpContext.GetOwinContext().Authentication;
    }
}

public ActionResult Login()
{
    return View();
}

[HttpPost]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Login(LoginModel model)
{
    if (ModelState.IsValid)
    {
        User user = await db.Users.Include(u=>u.Role).FirstOrDefaultAsync(u => u.Email ==
model.Email && u.Password == model.Password);

        if (user == null)
        {
            ModelState.AddModelError("", "Невірний логін або пароль.");
        }
        else
        {
            ClaimsIdentity claim = new ClaimsIdentity("ApplicationCookie",
ClaimsIdentity.DefaultNameClaimType, ClaimsIdentity.DefaultRoleClaimType);
            claim.AddClaim(new Claim(ClaimTypes.NameIdentifier, user.Id.ToString(),
ClaimValueTypes.String));
            claim.AddClaim(new Claim(ClaimsIdentity.DefaultNameClaimType, user.Email,
ClaimValueTypes.String));
            claim.AddClaim(new
Claim("http://schemas.microsoft.com/accesscontrolservice/2010/07/claims/identityprovider",
"OWIN Provider", ClaimValueTypes.String));
            if (user.Role!=null)
                claim.AddClaim(new Claim(ClaimsIdentity.DefaultRoleClaimType, user.Role.Name,
ClaimValueTypes.String));

            AuthenticationManager.SignOut();
            AuthenticationManager.SignIn(new AuthenticationProperties
            {
                IsPersistent = true
            }, claim);
            return RedirectToAction("Index", "Home");
        }
    }
    return View(model);
}

```

```

public ActionResult Logout()
{
    AuthenticationManager.SignOut();
    return RedirectToAction("Index", "Home");
}
}
}

```

Насамперед за допомогою контексту OWIN ми отримуємо об'єкт Authentication Manager. Що це за об'єкт? Вихідний код класу AuthenticationManager можна знайти тут. Даний об'єкт управляє аутентифікаційних куками і виконує ряд інших дій. У POST-методі Login ми отримуємо з представлення об'єкт LoginModel і по ньому визначаємо користувача. Якщо користувач був знайдений, то створюється об'єкт ClaimsIdentity:

```

ClaimsIdentity claim = new ClaimsIdentity("ApplicationCookie",
ClaimsIdentity.DefaultNameClaimType, ClaimsIdentity.DefaultRoleClaimType);

```

Перший параметр - "ApplicationCookie" вказує на тип аутентифікації, тобто куки. І в конструктор також передаються константи ClaimsIdentity.DefaultNameClaimType і ClaimsIdentity.DefaultRoleClaimType, які представляють назву клейма логіна користувача та назву клейма ролі. Далі в об'єкт ClaimsIdentity послідовно додаються клейма, які містять логін користувача, його id, роль:

```

claim.AddClaim(new Claim(ClaimTypes.NameIdentifier, user.Id.ToString(), ClaimValueTypes.String));
claim.AddClaim(new Claim(ClaimsIdentity.DefaultNameClaimType, user.Email,
ClaimValueTypes.String));
if (user.Role!=null)
    claim.AddClaim(new Claim(ClaimsIdentity.DefaultRoleClaimType, user.Role.Name,
ClaimValueTypes.String));

```

І також нам треба додати спеціальне клеймо провайдера:

```

claim.AddClaim(new
Claim("http://schemas.microsoft.com/accesscontrolservice/2010/07/claims/identityprovider",
"OWIN Provider", ClaimValueTypes.String));

```

Причому другий параметр представляє в даному випадку довільну назву провайдера - "OWIN Provider". За схожою схемою відбувається створення аутентифікаційного тікета в ASP.NET Identity, правда, там провайдер чітко визначено - "Asp Net Identity". Далі за допомогою методів AuthenticationManager.SignOut() і AuthenticationManager.SignIn() куки видаляються і знову встановлюються. Таким чином, аутентифікаційний білет буде містити дані про логін користувача, його ролі і id. Ми вже за замовчуванням можемо отримувати логін користувача: User.Identity.Name. Але

на даний момент в проєкті немає ніякого способу визначати ід користувача і роль. І для цього додамо в папку Models новий клас IdentityExtensions:

```
using System;
using System.Globalization;
using System.Security.Claims;
using System.Security.Principal;

namespace IdentityUoWApp.Models
{
    public static class IdentityExtensions
    {
        public static T GetUserId<T>(this IIdentity identity) where T : IConvertible
        {
            if (identity == null)
            {
                throw new ArgumentNullException("identity");
            }
            var ci = identity as ClaimsIdentity;
            if (ci != null)
            {
                var id = ci.FindFirst(ClaimTypes.NameIdentifier);
                if (id != null)
                {
                    return (T)Convert.ChangeType(id.Value, typeof(T), CultureInfo.InvariantCulture);
                }
            }
            return default(T);
        }
        public static string GetUserRole(this IIdentity identity)
        {
            if (identity == null)
            {
                throw new ArgumentNullException("identity");
            }
            var ci = identity as ClaimsIdentity;
            string role = "";
            if (ci != null)
            {
                var id = ci.FindFirst(ClaimsIdentity.DefaultRoleClaimType);
                if (id != null)
                {
                    role = id.Value;
                }
            }
            return role;
        }
    }
}
```

Даний клас створює два методи розширення Get User Id і GetUserRoles для отримання з аутентифікаційного білета ід і ролі користувача.

Тому ми можемо, наприклад, використовувати розмежування входу і ці методи в наявному у нас HomeController:

```
using System.Web.Mvc;
using IdentityUoWApp.Models;

namespace IdentityUoWApp.Controllers
{
    [Authorize]
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            return View();
        }
        [Authorize(Roles ="admin")]
        public ActionResult About()
        {
            ViewBag.Message = User.Identity.GetUserRole();

            return View();
        }
        public ActionResult Contact()
        {
            int id = User.Identity.GetUserId<int>();
            ViewBag.Message = "Ваш id: " + id.ToString();

            return View();
        }
    }
}
```

3.12.22 Авторизація на основі Claims

Інструмент Claims в рамках ASP.NET Identity представляє нам спосіб для створення системи авторизації, яка заснована на самих різних властивості об'єкта або так звана Claims-Based Authorization. Наприклад, якщо ми хочемо розмежувати доступ до ресурсів в залежності від віку користувача, то за допомогою Claims ми це легко можемо зробити.

Отже, створимо новий проект з типом авторизації Individual User Accounts. Нехай нам треба додати розмежування доступу за віком користувача. Для цього перш за все змінимо підсистему реєстрації. У модель RegisterModel з файлу Models / AccountViewModels.cs додамо властивість Year для зберігання року народження користувача:

```
public class RegisterViewModel
{
    [Required]
    [EmailAddress]
```

```

[Display(Name = "Email")]
public string Email { get; set; }

[Required]
[DataType(DataType.Password)]
[Display(Name = "Password")]
public string Password { get; set; }

[DataType(DataType.Password)]
[Display(Name = "Confirm password")]
[Compare("Password", ErrorMessage = "The password and confirmation password do not
match.")]
public string ConfirmPassword { get; set; }

[Required]
[Range(1910, 2015)]
[Display(Name = "Year")]
public int Year { get; set; }
}

```

І також додамо в представлення для реєстрації Register.xhtml код для створення поля введення року народження:

```

<div class="form-group">
    @Html.LabelFor(m => m.Year, new { @class = "col-md-2 control-label" })
    <div class="col-md-10">
        @Html.TextBoxFor(m => m.Year, new { @class = "form-control" })
    </div>
</div>

```

Отже, в моделі Register Model визначено властивість Year, а в поданні у нього є поле введення. Тепер змінимо post-версію методу Register, щоб використовувати цю властивість:

```

[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Register(RegisterViewModel model)
{
    if (ModelState.IsValid)
    {
        var user = new ApplicationUser { UserName = model.Email, Email = model.Email };

        var result = await UserManager.CreateAsync(user, model.Password);
        if (result.Succeeded)
        {
            // створюємо claim для зберігання року народження
            var identityClaim = new IdentityUserClaim { ClaimType = "Year", ClaimValue =
model.Year.ToString() };
            // додаємо claim користувачу
            user.Claims.Add(identityClaim);

```

```

        // зберігаємо зміни
        await UserManager.UpdateAsync(user);
        await SignInManager.SignInAsync(user, isPersistent:false, rememberBrowser:false);
        return RedirectToAction("Index", "Home");
    }
    AddErrors(result);
}
return View(model);
}

```

Після створення користувача спочатку формується об'єкт IdentityUser Claim:

```

var identityClaim = new IdentityUserClaim { ClaimType = "Year", ClaimValue = model.Year.ToString()
};

```

Щоб задіяти цей клас, треба підключити простір імен Microsoft.AspNet.Identity.EntityFramework. Властивість ClaimType відповідає за тип даних. Оскільки тут зберігається рік, то і тип називається Year. А властивість ClaimValue зберігає значення для цього типу у вигляді рядка. Далі claim додається користувачеві, і відбувається оновлення:

```

user.Claims.Add(identityClaim);
await UserManager.UpdateAsync(user);

```

В результаті claim буде збережений в базі даних. Для зберігання claim в ASP.NET Identity використовується таблиця AspNetUserClaims. А збережені дані будуть виглядати приблизно так:

	Id	UserId	ClaimType	ClaimValue
	1	6540f673-ba13-4a01-abc7-82682586517a	Year	1990
	2	a16426d7-9b1d-44db-87d9-e04f409c4fa2	Year	1991
	3	226adaef-1a3e-421d-bea1-b35c438ddf80	Year	1992
	4	8e07a7b2-816d-4c55-84a9-1baf733a5efe	Year	1995
*	NULL	NULL	NULL	NULL

Рисунок 3.82 - Вміст таблиці AspNetUserClaims

В даному випадку визначається claim "Year" для зберігання року народження, але також ми можемо зберігати і будь-яку іншу інформацію про користувача. Змінимо клас ApplicationUser, щоб він зберігав дані свого claim в куках:

```

public class ApplicationUser : IdentityUser
{
    public async Task<ClaimsIdentity> GenerateUserIdentityAsync(UserManager<ApplicationUser>
manager)
    {

```

```

        var userIdentity = await manager.CreateIdentityAsync(this,
DefaultAuthenticationTypes.ApplicationCookie);

        var yearClaim = Claims.FirstOrDefault(c => c.ClaimType == "Year");
        if (yearClaim != null)
            userIdentity.AddClaim(new Claim(yearClaim.ClaimType, yearClaim.ClaimValue));
        return userIdentity;
    }
}

```

І в кінці нам треба створити атрибут, який би дозволив встановлювати обмеження щодо доступу. Для цього додамо в проект нову папку Util, і Хатем в неї додамо новий клас ClaimsAuthorizeAttribute:

```

using System;
using System.Security.Claims;
using System.Web;
using System.Web.Mvc;

namespace AspClaimsApp.Util
{
    public class ClaimsAuthorizeAttribute : AuthorizeAttribute
    {
        public int Age { get; set; }

        protected override bool AuthorizeCore(HttpContextBase httpContext)
        {
            ClaimsIdentity claimsIdentity;
            if (!httpContext.User.Identity.IsAuthenticated)
                return false;

            claimsIdentity = httpContext.User.Identity as ClaimsIdentity;
            var yearClaims = claimsIdentity.FindFirst("Year");
            if (yearClaims == null)
                return false;

            int year; // одержуємо рік
            if (!Int32.TryParse(yearClaims.Value, out year))
                return false;

            // перевіряємо вік відносно поточної дати
            if ((DateTime.Now.Year - year) < this.Age)
                return false;

            // звертаємось до методу базового класу
            return base.AuthorizeCore(httpContext);
        }
    }
}

```

Даний клас успадковується від атрибута авторизації `AuthorizeAttribute` і додає властивість `Age`, яке дозволить встановлювати обмеження за віком.

У самому класі за допомогою одержуваного контексту запиту `HttpContext` витягаємо `claim` і встановлюємо, чи має поточний користувач доступ до ресурсу. Якщо користувач проходить авторизацію, то повертається `true`, якщо немає, то `false`. Після цього ми зможемо використовувати цей атрибут в контролері. наприклад:

```
public class HomeController : Controller
{
    [ClaimsAuthorize(Age=18)]
    public ActionResult Index()
    {
        return View();
    }

    [ClaimsAuthorize(Age = 22)]
    public ActionResult About()
    {
        return View();
    }
}
```

Контрольні питання.

1. Особливості ідентифікації за допомогою ASP.NET Identify
2. Особливості ідентифікації за допомогою об'єктів Claim
3. Валідація. Способи підтвердження валідації.

3.13 Бандли і мініфікація

3.13.1 Вступ до бандлів і мініфікацій

У ASP.NET MVC4 була введена концепція бандлів, яка допомагає організувати файли скриптів і стилів більш ефективним шляхом для зниження витрат при передачі клієнту. Подивимося, що представляють собою бандли.

У попередніх версіях MVC ми могли стандартним чином підключати скрипти і стилі, наприклад:

```
<head>
  <meta name="viewport" content="width=device-width" />
  <link type="text/css" rel="stylesheet" href="~/Content/Site.css" />
  <script src="~/Scripts/jquery-1.10.2.js" type="text/javascript"></script>
  <title>Index</title>
</head>
```


Фреймворк MVC для настройки шляхів пропонує нам використовувати URL-хелпери:

```
<head>
  <meta name="viewport" content="width=device-width" />
  <link type="text/css" rel="stylesheet" href="@Url.Content("~/Content/Site.css")" />
  <script src="@Url.Content("~/Scripts/jquery-1.10.2.js")" type="text/javascript"></script>
  <title>Index</title>
</head>
```

Використання бандлів представляє зовсім інший підхід до використання скриптів і стилів. При створенні нового проекту MVC 5 функціональність бандлів вже за замовчуванням включається в додаток. Якщо ви подивіться на стандартний код майстер-сторінки `_Layout.cshtml`, то побачите зовсім інший спосіб підключення скриптів і стилів:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>@ViewBag.Title - My ASP.NET Application</title>
  @Styles.Render("~/Content/css")
  @Scripts.Render("~/bundles/modernizr")

</head>
<body>
  <!-- Основний вміст майстер-сторінки -->

  @Scripts.Render("~/bundles/jquery")
  @Scripts.Render("~/bundles/bootstrap")
  @RenderSection("scripts", required: false)
</body>
</html>
```

В даному випадку підключаються не конкретні скрипти і стилі, а саме бандли. За допомогою хелпера `Styles.Render` підключаються бандли стилів, а за допомогою хелпера `Scripts.Render` ці хелпери приймають в якості параметра ім'я бандла. За замовчуванням проекти MVC 5 вже містять реєстрацію бандлів - в файлі `BundleConfig.cs`, який знаходиться в папці `App_Start`:

```
using System.Web;
using System.Web.Optimization;

namespace BundlesApp
{
    public class BundleConfig
    {
        public static void RegisterBundles(BundleCollection bundles)
```

```

{
    bundles.Add(new ScriptBundle("~/bundles/jquery").Include(
        "~/Scripts/jquery-{version}.js"));

    bundles.Add(new ScriptBundle("~/bundles/jqueryval").Include(
        "~/Scripts/jquery.validate*"));

    bundles.Add(new ScriptBundle("~/bundles/modernizr").Include(
        "~/Scripts/modernizr-*"));

    bundles.Add(new ScriptBundle("~/bundles/bootstrap").Include(
        "~/Scripts/bootstrap.js",
        "~/Scripts/respond.js"));

    bundles.Add(new StyleBundle("~/Content/css").Include(
        "~/Content/bootstrap.css",
        "~/Content/site.css"));
}
}
}

```

Тут метод RegisterBundles додає всі створювані бандли в колекцію bundles. Оголошення бандла виглядає наступним чином:

```
new ScriptBundle("~/bundles/jquery").Include("~/Scripts/jquery-{version}.js").
```

У конструктор ScriptBundle передається віртуальний шлях бандла. А за допомогою методу Include вже включаються до цього бандла конкретні файли скриптів. У виразі "~/Scripts/jquery-{version}.js" параметр {version} є замінником для будь-якого символічного позначення версії скрипта. Це дуже зручно, оскільки через деякий час ми можемо поміняти версію бібліотеки, але при цьому в коді нам нічого не доведеться змінювати, так як система вже автоматично прийме нову версію.

Вираз "~/Scripts/jquery.validate*" за допомогою знака зірочки замінює іншу частину рядка. Наприклад, цей вираз підключить в бандл відразу два файли: jquery.validate.js і jquery.validate.unobtrusive.js (і їх мінімізовані версії), так як їх назви починаються з jquery.validate*.

Те ж саме стосується і створення бандлів стилів, тільки в цьому випадку використовується клас StyleBundle. Однак саме оголошення бандлів в файлі BundleConfig.cs ще не підключає автоматично їх в проект. Для цього в файлі Global.asax прописується відповідна директива:

```

public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        //.....
        BundleConfig.RegisterBundles(BundleTable.Bundles);
    }
}

```

Таким чином, ми і реєструємо бандли для вашого додатку.

Мініфікація

Ключовим моментом концепції бандлів є мініфікація. Її суть полягає в тому, що при розгортанні додатку клієнту віддається не повна, а мінімізована версія скриптів або стилів. За рахунок чого економляться ресурси сервера, так як йдуть віддачі файлів з меншим розміром. У той же час в процесі налагодження додаток віддає звичайну версію, оскільки завдяки їй, нам простіше розібратися в можливих помилках. Подивимося на прикладі. Нехай є деяка майстер-сторінка, яка підключає значну кількість бандлів:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>@ViewBag.Title</title>
  @Styles.Render("~/Content/css")
  @Scripts.Render("~/bundles/modernizr")
</head>
<body>
  @RenderBody()

  @Scripts.Render("~/bundles/jquery")
  @Scripts.Render("~/bundles/jqueryval")
  @Scripts.Render("~/bundles/bootstrap")
  @RenderSection("scripts", required: false)
</body>
</html>
```

В даному випадку ми підключили п'ять бандлів. Якщо запустити тепер додаток, то через засоби розробника в InternetExplorer 11 ми можемо побачити скрипти, які несуть бандли:

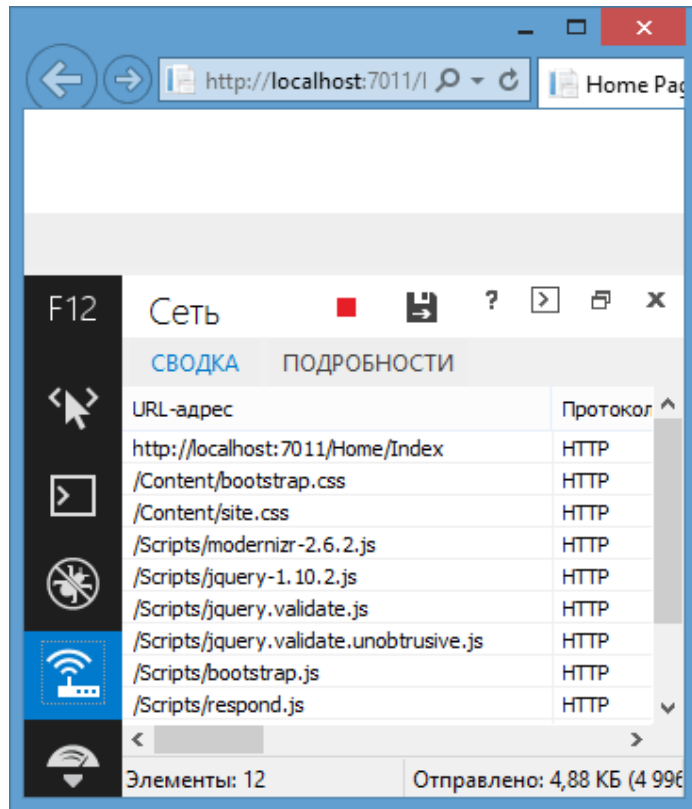
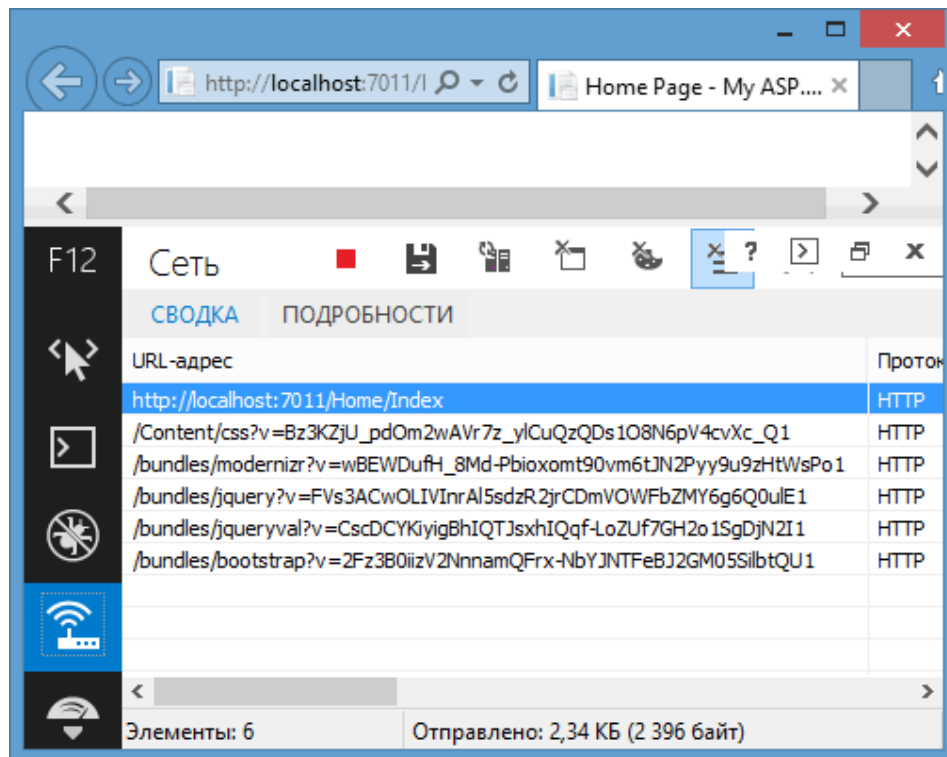


Рисунок 3.83 - Скрипти, які містять бандли

В даному випадку у нас за замовчуванням додаток знаходиться в режимі налагодження. У файлі `web.config` можна знайти наступну секцію:

```
<configuration>
<!------->
<system.web>
  <authentication mode="None" />
  <compilation debug="true" targetFramework="4.5" />
  <httpRuntime targetFramework="4.5" />
<!------->
```

Параметр `compilation debug = "true"` вказує, що запуск програми здійснюється в режимі налагодження. Тепер змінимо його значення на `false`: `debug = "false"` і перезапустимо проект. Тепер ми побачимо дещо інший результат:



І якщо ми подивимося, наприклад, в браузері в засобах розробника заголовки відповіді, то ви побачимо, що в цей раз в режимі розгортання сервер віддає нам мінімізований файл:

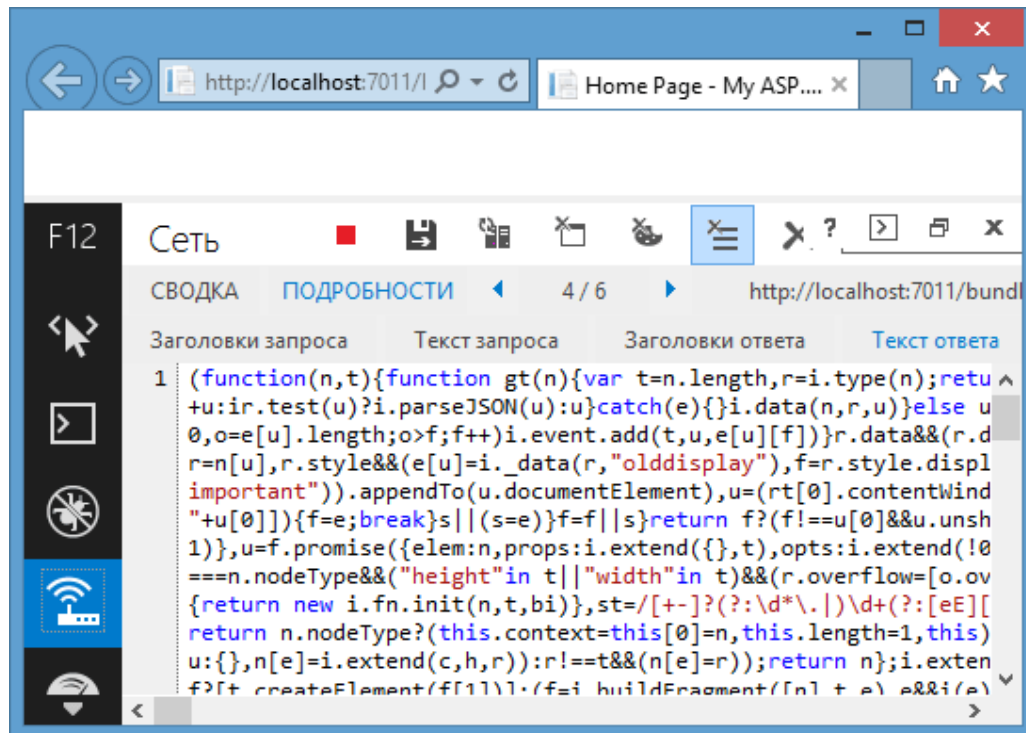


Рисунок 3.84 - Мінімізація файлу

Таким чином, використання бандлів в додатках на ASP.NET MVC підвищує ефективність і продуктивність.

Крім того, для мінімізації скриптів і стилів існує велика кількість утиліт, які можна додатково використовувати. Наприклад, непогана утиліта по мінімізації скриптів Microsoft Ajax Minifier, що дозволяє стиснути скрипти в середньому до 50%, а іноді і більше.

3.13.2 Створення бандлів. Бібліотеки з CDN

Створимо свій бандл. Для прикладу, визначимо в папці зі скриптами підкаталог MyScripts для своїх скриптів, які нам потрібно підключати в своєму бандлі:

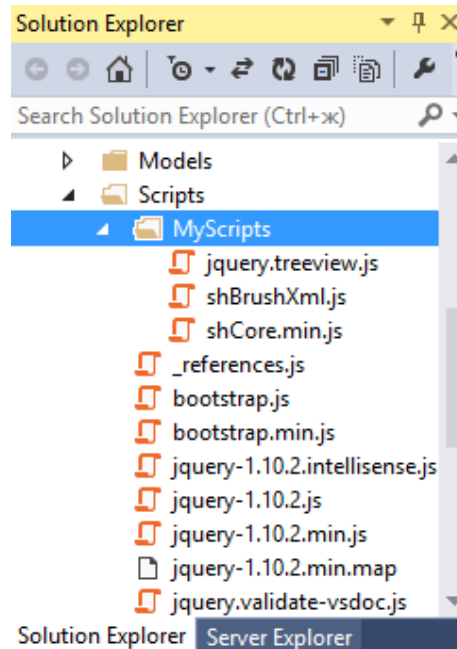


Рисунок 3.85 - Визначення каталогу скриптів

У файлі BundleConfig.cs ми додаємо свій бандл з моїми файлами:

```
using System.Web;
using System.Web.Optimization;

namespace Guestbook
{
    public class BundleConfig
    {
        public static void RegisterBundles(BundleCollection bundles)
        {
            bundles.Add(new ScriptBundle("~/bundles/myscripts").Include(
                "~/Scripts/MyScripts/*.js"));
            bundles.Add(new ScriptBundle("~/bundles/jquery").Include(
                "~/Scripts/jquery-{version}.js"));
            // Далі йде підключення інших бандлів
        }
    }
}
```

Вираз "~ / Scripts / MyScripts / *. Js" підключить в бандл всі файли js, які є в каталозі Scripts / MyScripts. Далі переходимо до розмітки і підключаємо там новий бандл:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width" />
  <title>@ViewBag.Title</title>
  @Styles.Render("~/Content/css")
  @Scripts.Render("~/bundles/modernizr")
</head>
<body>
  @RenderBody()

  @Scripts.Render("~/bundles/jquery")
  @Scripts.Render("~/bundles/myscripts")
  @RenderSection("scripts", required: false)
</body>
</html>
```

І після цього бандл буде підключатися до додатка. При підключенні бандлів не варто забувати про черговість: якщо у вас є бандли, файли яких залежать від інших файлів в інших бандлах, наприклад, від jquery, то, природно, залежні бандли треба підключати після основних.

Підключення бібліотек з мереж CDN

Звичайною справою на сьогоднішній день є підключення різних популярних бібліотек javascript з мереж CDN. Якщо ми хочемо підключати, наприклад, бібліотеку jquery не зі своєї програми, а з якої-небудь мережі CDN, то ми повинні включити підтримку CDN в бандлів в файлі BundleConfig.cs:

```
using System.Web;
using System.Web.Optimization;

namespace Guestbook
{
    public class BundleConfig
    {
        public static void RegisterBundles(BundleCollection bundles)
        {
            // Оригінальне підключення за коментуємо, оскільки будемо підключатися з мережі CDN
            //bundles.Add(new ScriptBundle("~/bundles/jquery").Include(
            //    "~/Scripts/jquery-{version}.js"));

            bundles.UseCdn = true; //включаєм підтримку CDN
        }
    }
}
```

```
// додаємо адресу потрібної бібліотеки в деякій мережі CDN
var jqueryCdnPath = "http://ajax.aspnetcdn.com/ajax/jquery/jquery-1.7.1.min.js";

bundles.Add(new ScriptBundle("~/bundles/jquery",
    jqueryCdnPath).Include(
    "~/Scripts/jquery-{version}.js"));

// Далі решта бандлів....
}
}
}
```

Далі підключаємо бандл звичайним способом. Крім того, ми можемо поставити можливість підключення бібліотеки зі свого додатку, якщо бібліотеку не вдалося завантажити з цієї мережі:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width" />
  <title>@ViewBag.Title</title>
  @Styles.Render("~/Content/css")
  @Scripts.Render("~/bundles/modernizr")
</head>
<body>
  @RenderBody()

  @Scripts.Render("~/bundles/jquery")
  <script type="text/javascript">
    if (typeof jQuery == 'undefined') {
      var e = document.createElement('script');
      e.src = '@Url.Content("~/Scripts/jquery-1.7.1.js")';
      e.type = 'text/javascript';
      document.getElementsByTagName("head")[0].appendChild(e);
    }
  </script>
  @RenderSection("scripts", required: false)
</body>
</html>
```

Контрольні питання.

1. Призначення бандлів.
2. Переваги при використанні мініфікацій.

3.14 Bootstrap в ASP.NET MVC 5

3.14.1 Вступ до Bootstrap-у

Bootstrap це css-фреймворк для створення адаптивних веб-додатків. Проект Bootstrap спочатку був створений і розвивався розробниками з Twitter для власних потреб, однак потім вийшов за межі твіттера і в даний час розвивається як open source-проект і є одним з найпопулярніших фреймворків для створення веб-додатків. Що в принципі і не дивно: в даний час все більше зростає кількість користувачів мобільного інтернету, все більше людей для веб-серфінгу використовують смартфони, планшети. Що в підсумку призвело до збільшення числа сайтів з адаптивним дизайном і відповідно до збільшення популярності засобів для розробки подібних сайтів, в тому числі фреймворка Bootstrap. Проекти ASP.NET MVC5 за замовчуванням вже містять всі необхідні файли Bootstrap:

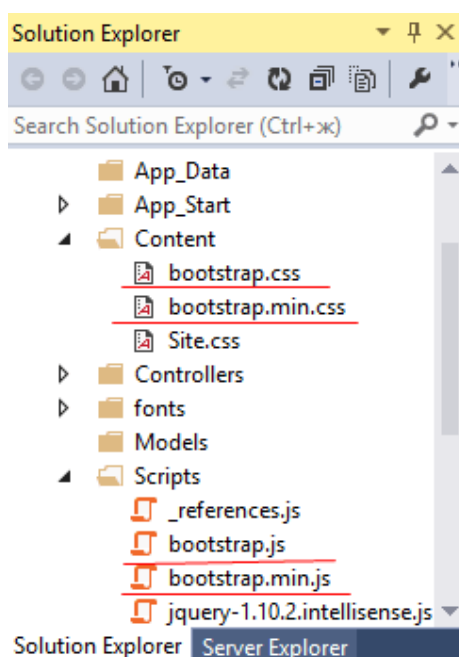


Рисунок 3.86 - Перевірка наявності Bootstrap в проекті

Функціональність Bootstrap укладена в файлі стилів bootstrap.css і скрипті bootstrap.js. Якщо ми відкриємо майстер-сторінку _Layout.cshtml, яка йде за замовчуванням в проекті, то ми побачимо підключення відповідних бібліотек:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>@ViewBag.Title - My ASP.NET Application</title>
  @Styles.Render("~/Content/css")
  @Scripts.Render("~/bundles/modernizr")
</head>
<body>
  <div class="navbar navbar-inverse navbar-fixed-top">
    <div class="container">
      <div class="navbar-header">
```

```

        <button type="button" class="navbar-toggle" data-toggle="collapse"
data-target=".navbar-collapse">
            <span class="icon-bar"></span>
            <span class="icon-bar"></span>
            <span class="icon-bar"></span>
        </button>
        @Html.ActionLink("Application name", "Index", "Home", null, new {
@class = "navbar-brand" })
    </div>
    <div class="navbar-collapse collapse">
        <ul class="nav navbar-nav">
            <li>@Html.ActionLink("Home", "Index", "Home")</li>
            <li>@Html.ActionLink("About", "About", "Home")</li>
            <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
        </ul>
    </div>
</div>
<div class="container body-content">
    @RenderBody()
    <hr />
    <footer>
        <p>© @DateTime.Now.Year - My ASP.NET Application</p>
    </footer>
</div>

@Scripts.Render("~/bundles/jquery")
@Scripts.Render("~/bundles/bootstrap")
@RenderSection("scripts", required: false)
</body>
</html>

```

Вираз `@Styles.Render("~/Content/css")` підключає необхідні стилі, в тому числі і bootstrap, а вираз `@Scripts.Render("~/bundles/bootstrap")` підключає скрипт bootstrap. Ми можемо це побачити, відкривши файлі `BundleConfig.cs`:

```

bundles.Add(new ScriptBundle("~/bundles/bootstrap").Include(
    "~/Scripts/bootstrap.js",
    "~/Scripts/respond.js"));

bundles.Add(new StyleBundle("~/Content/css").Include(
    "~/Content/bootstrap.css",
    "~/Content/site.css"));

```

Також на майстер-сторінці ми можемо побачити використання таких класів CSS, як **navbar**, **navbar-inverse**, **navbar-fixed-top** і т.д. - Все це класи bootstrap, які дозволяють адаптувати інтерфейс сторінки до різних пристроїв. Наприклад, запустимо сторінку в веб-браузері на ПК:

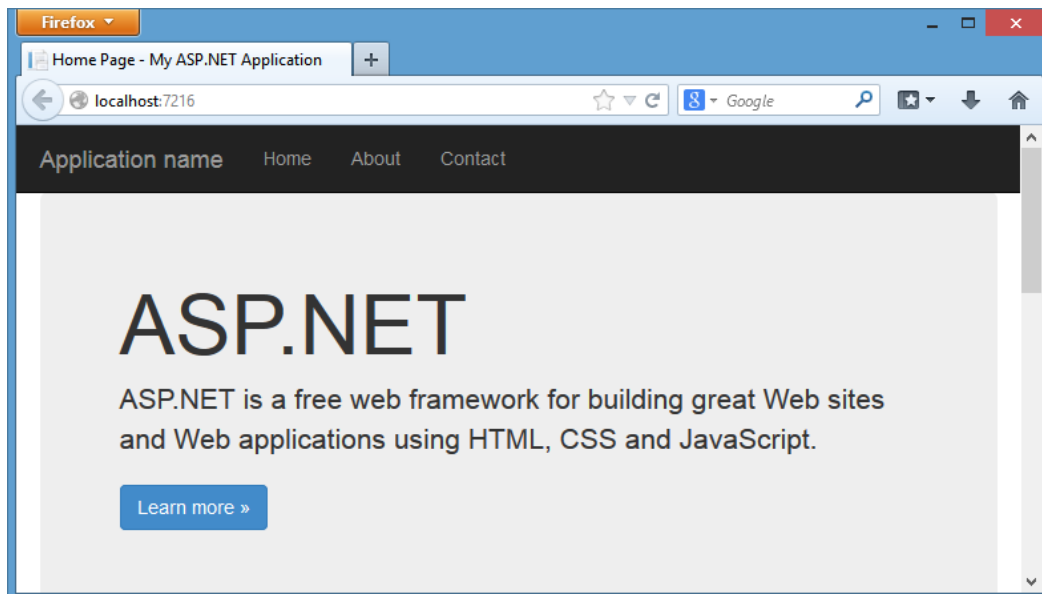


Рисунок 3.87 - Тестовий проект. Вид на ПК

Якщо ж ми запустимо цю програму на мобільному телефоні або емуляторі, то побачимо, що інтерфейс як і раніше виглядає досить непогано:

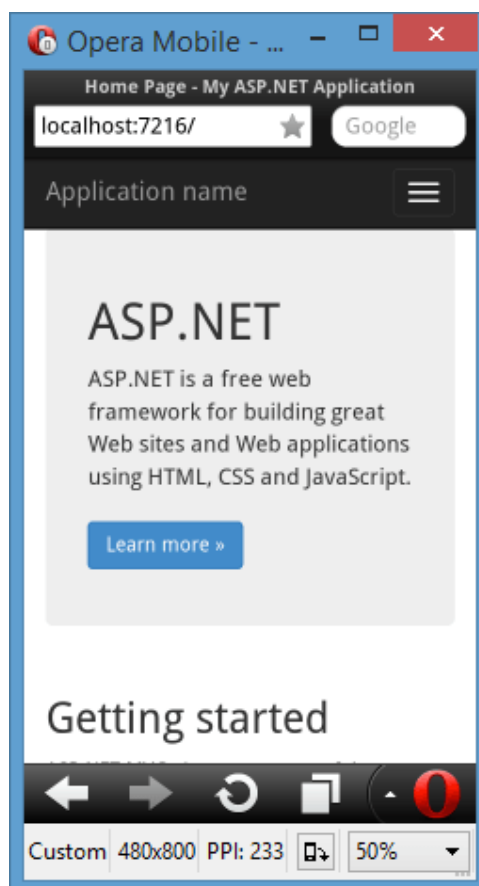


Рисунок 3.88 - Тестовий проект на екрані телефону

Власне в цьому і перевага використання Bootstrap. Основою для адаптивного дизайну стали правила *media queries*, які описують стилеві властивості елементів для різних дозволів екрану. З точки зору розробників Bootstrap всі пристрої можна поділити на такі групи:

- Дуже маленькі, ширина екрану яких менше 768 пікселів
- Невеликі - їх ширина від 768 пікселів і вище
- Середні з шириною від 992 пікселя і вище
- Великі з шириною від 1200 пікселів і вище

Цей поділ ми можемо побачити, якщо відкриємо файл bootstrap.css і знайдемо там правила media, наприклад:

```
@media (min-width: 768px) {
.....
}
```

А тепер розглянемо деякі класи даного фреймворка.

3.14.2 Позиціювання елементів

При використанні Bootstrap дуже зручно позиціонувати інтерфейс у вигляді таблиці або сітки, використовуючи рядки і стовпці. Так, якщо ми подивимося в браузері на стандартне представлення Index.cshtml, яке йде за замовчуванням, то ми побачимо подібне позиціонування: елементи під заголовком розташовані ніби в три стовпці і складають один рядок:

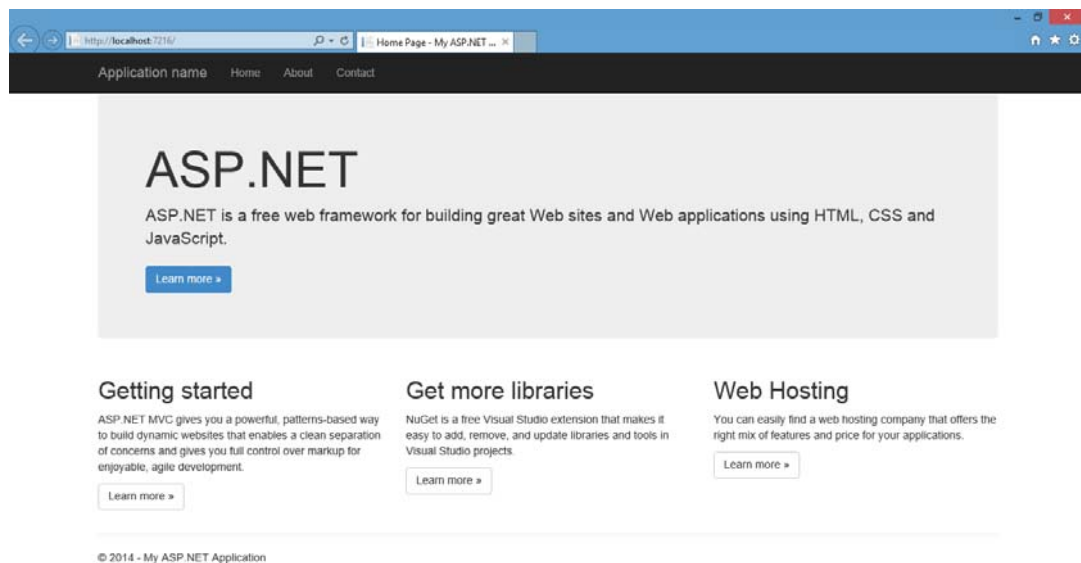


Рисунок 3.89 - Позиціювання елементів сторінки

Якщо ми відкриємо код представлення, то можемо побачити використання класів bootstrap:

```
<div class="row">
  <div class="col-md-4">
    <h2>Getting started</h2>
    <p>.....</p>
  </div>
  <div class="col-md-4">
    <h2>Get more libraries</h2>
```

```

        <p>.....</p>
    </div>
    <div class="col-md-4">
        <h2>Web Hosting</h2>
        <p>.....</p>
    </div>
</div>

```

Клас **row** задає розташування окремих блоків у вигляді одного рядка. Рядок в Bootstrap може мати до 12 стовпців. Рядків може бути скільки завгодно, але в даному випадку у нас тільки один рядок. Для створення окремого стовпця у рядку використовується клас **col-md-4**.

col, що як зрозуміло з назви, позначає стовпець.

Далі йде **md** - ідентифікатор пристрою. Як ми бачили в минулій темі, bootstrap ділить всі пристрої умовно на чотири групи в залежності від ширини екрану - **md**, зокрема, співвідноситься із середніми пристроями (тобто які мають ширину від 992 пікселя і вище). А число 4 вказує, скільки умовних одиниць в рядку буде займати цей блок. Таким чином, виходить, що клас **col-md-4** означає, що даний блок буде займати 4 умовних одиниць з 12 в рядку, тобто третину ширини екрану пристрою з екраном шириною від 992 пікселів. Оскільки у нас є чотири групи, то для кожної групи є свої класи. Наприклад, для дуже маленьких пристроїв з екраном менше 768 пікселів (тобто мобільних телефонів), подібний клас міг би бути таким **col-xs-4**.

Всі типи класів:

- col-xs- *: для пристроїв з шириною екрану менше 768 пікселів
- col-sm- *: для пристроїв з шириною екрану від 768 пікселів і вище
- col-md- *: для пристроїв з шириною екрану від 992 пікселя і вище
- col-lg- *: для пристроїв з шириною екрану від 1200 пікселів і вище

Хоча навіть на мобільних пристроях блок з класом col-md-4 буде виглядати цілком непогано, але ми можемо встановити відразу два класи, щоб ще більше деталізувати відображення на різних пристроях, наприклад:

```

<div class="col-xs-12 col-md-4">

```

І хоча в представленні Index.cshtml за замовчуванням всі три блоки мають відносну ширину в 4 одиниці, складаючи в цілому 12 одиниць, ми можемо поставити будь-яку іншу ширину, позиціонуючи їх на власний розсуд. наприклад:

.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1
.col-md-8								.col-md-4			
.col-md-4				.col-md-4				.col-md-4			
.col-md-6						.col-md-6					

Відступи

Спеціальні класи **col-md(sm|lg)-offset-*** дозволяють задати зсув щодо лівого блоку або початку рядка в умовних одиницях. Наприклад, у нас є такий рядок:

```
<div class="row">
  <div class="col-md-4">
    <h2>Лівий блок</h2>
    <p>.....</p>
  </div>
  <div class="col-md-4 col-md-offset-4">
    <h2>Правий блок</h2>
    <p>.....</p>
  </div>
</div>
```

Клас col-md-offset-4 буде зміщувати правий блок на 4 умовних одиниці вправо. Деякі приклади з використання зміщення:

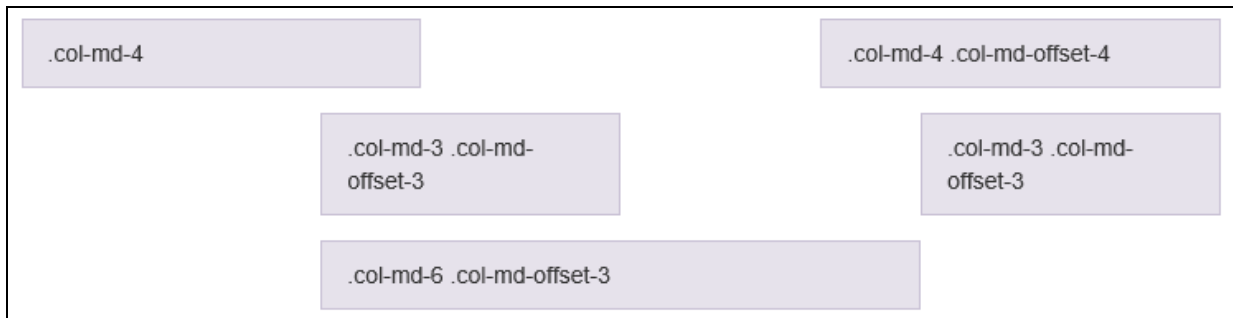


Рисунок 3.90 - Типи зміщень даних

Порядок стовпців

За допомогою класів **col-md(sm|lg)-push-*** і **col-md(sm|lg)-pull-*** ми можемо перевизначити порядок блоків в рядку. Наприклад, клас **col-md-push-4** зрушує блок на чотири одиниці вправо від поточного положення. І навпаки, клас **col-md-pull-4** зрушує блок вліво від поточної позиції. Так, попередній приклад зі зміщенням ми могли б переписати таким чином:

```
<div class="row">
  <div class="col-md-4">
    <h2>Лівий блок</h2>
    <p>.....</p>
```

```

</div>
<div class="col-md-4 col-md-push-4">
  <h2>Правий блок</h2>
  <p>.....</p>
</div>
</div>

```

В такому разі, у нас був би той же ефект. Але ми можемо також повністю перепорядкувати порядок проходження блоків:

```

<div class="row">
  <div class="col-md-4 col-md-push-8">
    <h2>Лівий блок</h2>
    <p>.....</p>
  </div>
  <div class="col-md-4 col-md-pull-4">
    <h2>Правий блок</h2>
    <p>.....</p>
  </div>
</div>

```

Тепер правий блок зміщується вліво на чотири одиниці, в лівий блок - вправо на 8 одиниць – таким чином вони поміняються місцями. Таким чином, використовуючи клас `row` і класи стовпців, ми можемо здійснити загальне розташування елементів, а адаптивність `bootstrap` гарантує, що на будь-яких пристроях подібна сітка елементів буде виглядати цілком нормально. Тепер розглянемо деякі компоненти, які пропонує нам `Bootstrap`.

3.14.3 Компоненти Bootstrap

`Bootstrap` має ряд компонентів, які не зводяться до стандартних кнопок або текстових полів, а представляють більш складні елементи. Повний список компонентів `Bootstrap` можна знайти на офіційній сторінці проекту <http://getbootstrap.com/components/>. Розглянемо коротко деякі з них:

Навігаційні панелі

Перший компонент `Bootstrap`, з яким ми стикаємося в проекті, це навігаційна панель. Даний компонент розташовується на майстер-сторінці `_Layout`:

```

<div class="navbar navbar-inverse navbar-fixed-top">
  <div class="container">
    <div class="navbar-header">
      <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-collapse">
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
      </button>

```

```

    @Html.ActionLink("Application name", "Index", "Home", null, new { @class = "navbar-
brand" })
  </div>
  <div class="navbar-collapse collapse">
    <ul class="nav navbar-nav">
      <li>@Html.ActionLink("Home", "Index", "Home")</li>
      <li>@Html.ActionLink("About", "About", "Home")</li>
      <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
    </ul>
  </div>
</div>
</div>

```

І навіть якщо ми звузимо границі веб-браузера або запустимо сайт на мобільному пристрої, то ми побачимо, що панель навігації залишається досить функціональною:

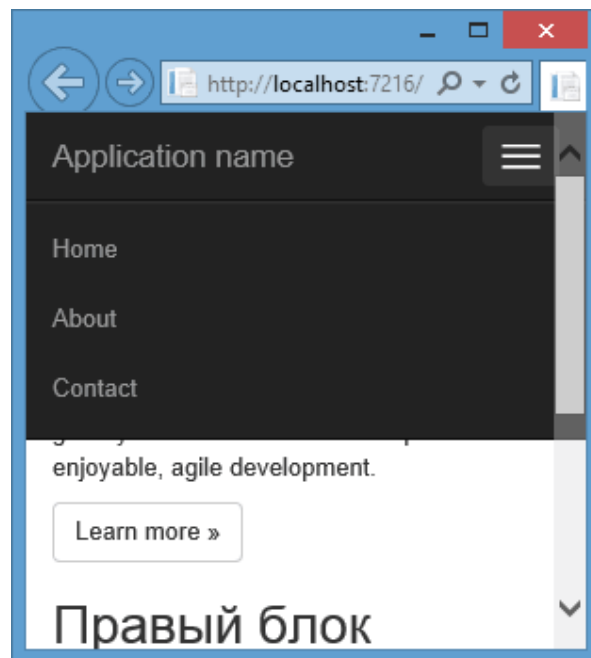


Рисунок 3.91 - Згорнута панель навігації

Компонент представляє клас **navbar**. Щоб панель була фіксована по верху сторінки, використовується також клас **navbar-fixed-top**. Якби ми, навпаки, захотіли фіксувати панель по низу, то в цьому випадку використовували б клас **navbar-fixed-bottom**. В оголошенні блоку навігації використовується клас **navbar-inverse**, який інвертує колір за замовчуванням. Замість цього класу ми могли б використовувати **navbar-default**, тоді в цьому випадку у нас би було меню стандартних світлих тонів.

Для створення посилань навігації застосовується клас **nav**. Bootstrap представляє кілька класів для оформлення посилань навігації. За замовчуванням використовується клас **navbar-nav**, але ми можемо використовувати і інші можливості. Для створення навігації по типу вкладок застосовується клас **nav-tabs**. Так, наприклад наступне меню:

```

<ul class="nav nav-tabs">

```



```

<li class="active">@Html.ActionLink("Home", "Index", "Home")</li>
<li>@Html.ActionLink("About", "About", "Home")</li>
<li>@Html.ActionLink("Contact", "Contact", "Home")</li>
</ul>

```

дасть такий ефект:

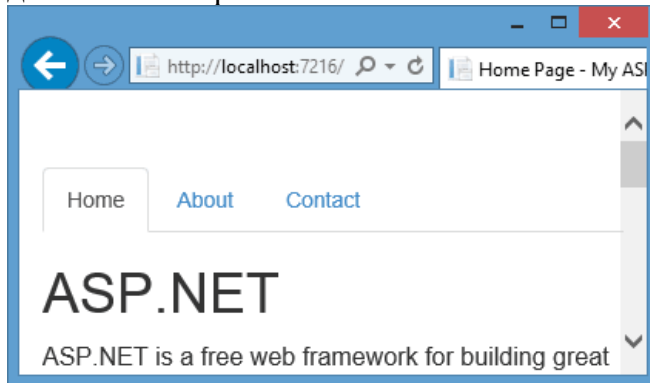


Рисунок 3.92 - Створення посилань навігації

Ще один варіант створення панелі посилань представляє клас `nav-pills`:

```

<ul class="nav nav-pills">
  <li class="active">@Html.ActionLink("Home", "Index", "Home")</li>
  <li>@Html.ActionLink("About", "About", "Home")</li>
  <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
</ul>

```

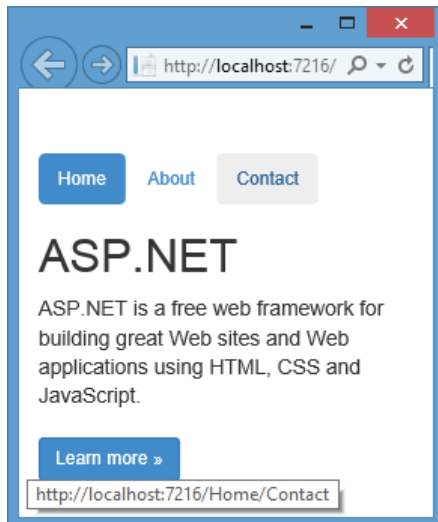


Рисунок 3.93 - Створення панелі посилань

Комбінуючи класи `nav-pills` і `nav-stacked`, ми можемо створити вертикальне меню:

```

<ul class="nav nav-pills nav-stacked">
  .....
</ul>

```

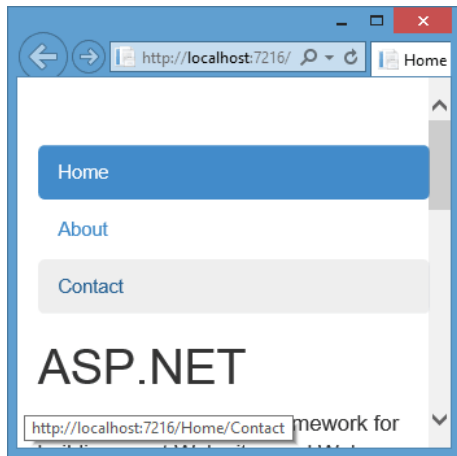


Рисунок 3.94 - Вертикальне меню

Посторінкова навігація

Клас **pagination** дозволяє створити панель посилань у вигляді посторінкової навігації:

```
<ul class="pagination">
  <li><a href="#"><</a></li>
  <li><a href="#">1</a></li>
  <li><a href="#">2</a></li>
  <li><a href="#">3</a></li>
  <li><a href="#">4</a></li>
  <li><a href="#">5</a></li>
  <li><a href="#">></a></li>
</ul>
```

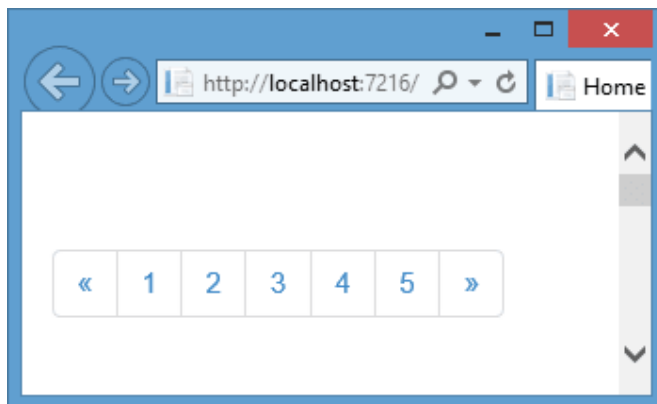


Рисунок 3.95 - Посторінкова навігація

Заголовки

Для створення посилань заголовків застосовується клас **breadcrumb**:

```
<ul class="breadcrumb">
  <li class="active">@Html.ActionLink("Home", "Index", "Home")</li>
  <li>@Html.ActionLink("About", "About", "Home")</li>
  <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
</ul>
```

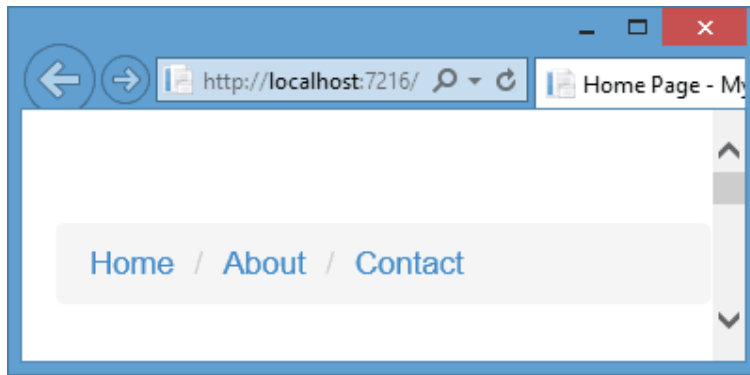


Рисунок 3.96 - Посилання заголовків

Кнопки

Для створення кнопок Bootstrap має клас btn. Як правило, кнопки оформляються в групу за допомогою класу btn-group:

```
<div class="btn-group" role="group">
  <button type="button" class="btn btn-default">Left</button>
  <button type="button" class="btn btn-default">Middle</button>
  <button type="button" class="btn btn-default">Right</button>
</div>
```

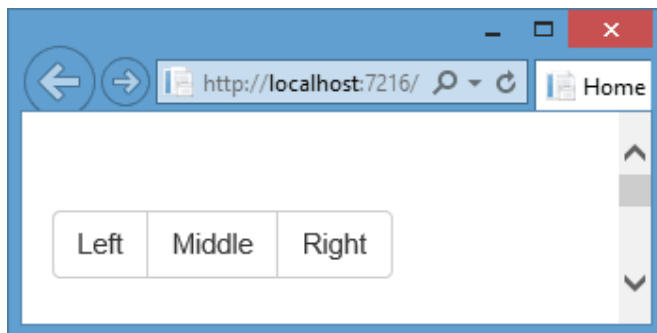


Рисунок 3.97- Кнопки Bootstrap

Кнопка зі списком, що випадає

Для створення списку за прикладом елемента нам треба використовувати кнопку разом зі списком, який повинен мати клас dropdown-menu:

```
<Div class = "btn-group">
  <Button type = "button" class = "btn btn-default dropdown-toggle" data-toggle = "dropdown">
    Мова програмування <span class = "caret"> </ span>
  </ Button>
  <Ul class = "dropdown-menu" role = "menu">
    <Li> <a href="#"> JavaScript </a> </ li>
    <Li> <a href="#"> C # </a> </ li>
    <Li> <a href="#"> VB.NET </a> </ li>
    <Li class = "divider"> </ li>
    <Li> <a href="#"> Java </a> </ li>
  </ Ul>
</ Div>
```

Мітки

Для оформлення фрагментів тексту в якості міток ми можемо використовувати клас `label`. Також ми можемо використовувати додаткові класи міток, щоб конкретизувати тип повідомлення:

```
<span class="label label-default">Default</span>  
<span class="label label-primary">Primary</span>  
<span class="label label-success">Success</span>  
<span class="label label-info">Info</span>  
<span class="label label-warning">Warning</span>  
<span class="label label-danger">Danger</span>
```

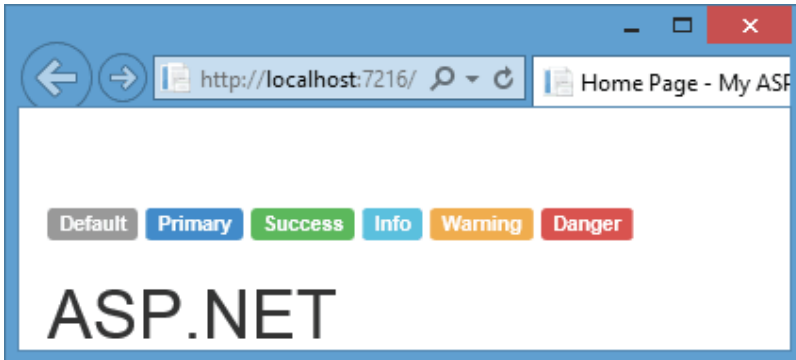


Рисунок 3.98 - Мітки

Схожу функціональність пропонує клас `alert`:

```
<Div class = "alert alert-success"> Завдання успішно завершена </ div>  
<Div class = "alert alert-info"> Додаткова інформація </ div>  
<Div class = "alert alert-warning"> Увага! </ Div>  
<Div class = "alert alert-danger"> Небезпечно! </ Div>
```

Це тільки деякі компоненти, які пропонує Bootstrap. Але вже по ним можна зрозуміти, що Bootstrap досить гнучкий, і навіть складні за структурою компоненти може легко адаптувати і пристосувати під конкретні пристрої.

3.14.4 Робота з формами в Bootstrap

Крім окремих компонентів Bootstrap підтримує також роботу з формами як з єдиним цілим. Якщо ми візьмемо стандартне представлення для реєстрації `Register.cshtml`, то ми побачимо, що до елементів форми вже застосовують класи Bootstrap:

```
@{  
    ViewBag.Title = "Register";  
}
```

```
<h2>@ViewBag.Title.</h2>
```

```
@using (Html.BeginForm("Register", "Account", FormMethod.Post, new { @class = "form-  
horizontal", role = "form" }))
```

```

{
    @Html.AntiForgeryToken()
    <h4>Create a new account.</h4>
    <hr />
    @Html.ValidationSummary()
    <div class="form-group">
        @Html.LabelFor(m => m.UserName, new { @class = "col-md-2 control-label" })
        <div class="col-md-10">
            @Html.TextBoxFor(m => m.UserName, new { @class = "form-control" })
        </div>
    </div>
    <div class="form-group">
        @Html.LabelFor(m => m.Password, new { @class = "col-md-2 control-label" })
        <div class="col-md-10">
            @Html.PasswordFor(m => m.Password, new { @class = "form-control" })
        </div>
    </div>
    <div class="form-group">
        @Html.LabelFor(m => m.ConfirmPassword, new { @class = "col-md-2 control-label" })
        <div class="col-md-10">
            @Html.PasswordFor(m => m.ConfirmPassword, new { @class = "form-control" })
        </div>
    </div>
    <div class="form-group">
        <div class="col-md-offset-2 col-md-10">
            <input type="submit" class="btn btn-default" value="Register" />
        </div>
    </div>
}

```

До форми за замовчуванням додається клас `form-horizontal`, який встановлює вирівнювання міток `label` по правій стороні пов'язаних елементів вводу:

Register.

Create a new account.

Рисунок 3.99 - Вирівнювання по правій стороні

Крім того, елементи візуально відрізняються від стандартних, так як до них застосовуються стилі з Bootstrap. Для стилізації елементів форм служить клас **form-control**. Тут також використовується клас **form-group**, який задає відстань між елементами форм.

Крім **form-horizontal** ми можемо застосувати до форми інший клас: **form-inline**, який вибудовує всі елементи в лінію. Наприклад:

```
@using (Html.BeginForm("Login", "Account", FormMethod.Post, new { @class = "form-inline", role = "form" }))
{
    @Html.AntiForgeryToken()
    <h4>Use a local account to log in.</h4>
    <hr />
    @Html.ValidationSummary(true)
    <div class="form-group">
        <div class="col-md-10">
            @Html.TextBoxFor(m => m.UserName, new { @class = "form-control", @placeholder = "Логин" })
            @Html.ValidationMessageFor(m => m.UserName)
        </div>
    </div>
    <div class="form-group">
        <div class="col-md-10">
            @Html.PasswordFor(m => m.Password, new { @class = "form-control", @placeholder="Пароль" })
            @Html.ValidationMessageFor(m => m.Password)
        </div>
    </div>
    <div class="form-group">
        <div class="col-md-offset-2 col-md-10">
            <div class="checkbox">
                @Html.CheckBoxFor(m => m.RememberMe)
                @Html.LabelFor(m => m.RememberMe)
            </div>
        </div>
    </div>
    <div class="form-group">
        <div class="col-md-offset-2 col-md-10">
            <input type="submit" value="Log in" class="btn btn-default" />
        </div>
    </div>
}
```

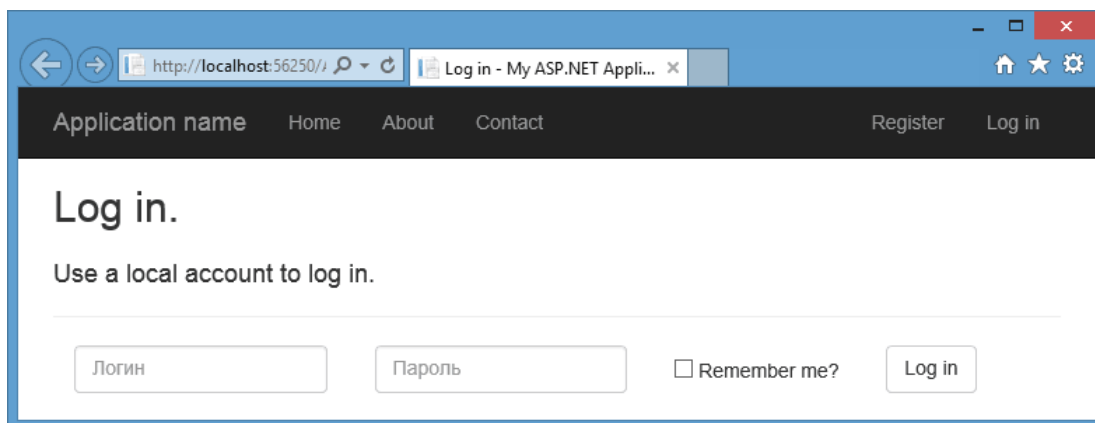


Рисунок 3.100 - Вирівнювання елементів в лінію

І на закінчення слід сказати, що при використанні шаблонів формування, коли Visual Studio автоматично генерує нам розмітку для представлень при створенні і редагування моделей, створювані автоматично форми вже застосовують класи Bootstrap, тому вам не доведеться вручну додавати до них вищеописані класи.

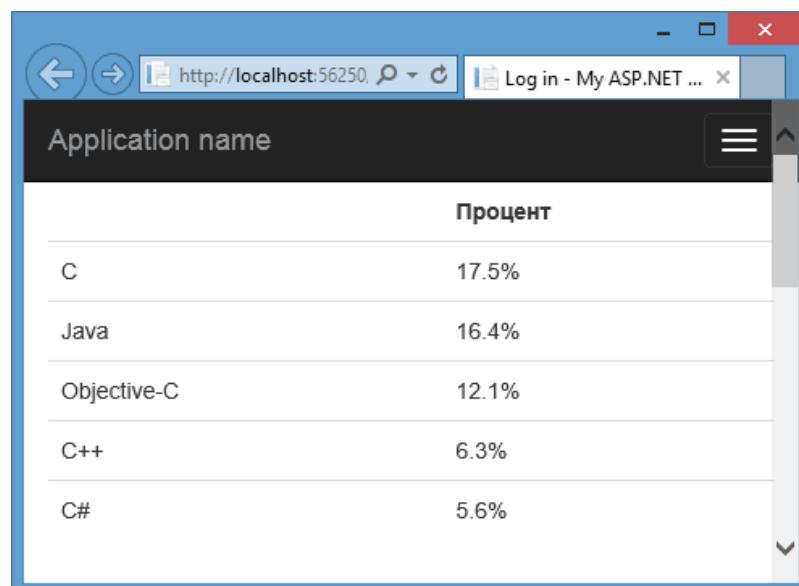
3.14.5 Робота з таблицями

Для оформлення таблиць в Bootstrap призначений клас table. Наприклад, нехай у нас є така таблиця:

```
<div class="row">
  <div class="col-md-6">
    <table class="table">
      <tbody>
        <tr>
          <th>

          </th>
          <th>
            Процент
          </th>
        </tr>
        <tr><td>C</td><td>17.5%</td></tr>
        <tr><td>Java</td><td>16.4%</td></tr>
        <tr><td>Objective-C</td><td>12.1%</td></tr>
        <tr><td>C++</td><td>6.3%</td></tr>
        <tr><td>C#</td><td>5.6%</td></tr>
      </tbody>
    </table>
  </div>
</div>
```

Тоді застосування класу table буде давати такий вивід:



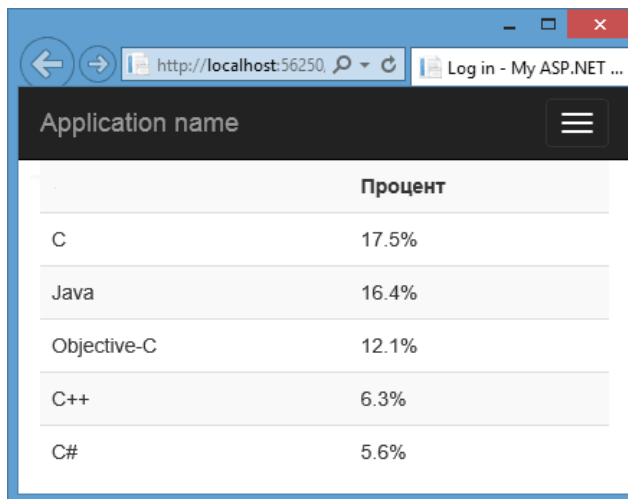
The screenshot shows a web browser window with the URL `http://localhost:56250`. The page displays a table with the following data:

Application name	Процент
C	17.5%
Java	16.4%
Objective-C	12.1%
C++	6.3%
C#	5.6%

Рисунок 3.101 - Табличний вивід

Крім цього класу ми можемо використовувати на додаток до нього ще ряд класів. Наприклад, клас **table-striped** дозволяє виділити парні і непарні рядки в таблиці:

```
<table class="table table-striped">
  <tbody>
    .....
  </tbody>
</table>
```

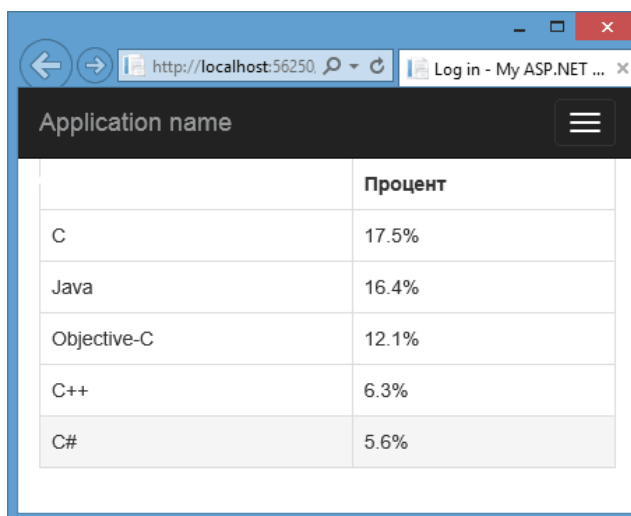


Application name	Процент
C	17.5%
Java	16.4%
Objective-C	12.1%
C++	6.3%
C#	5.6%

Рисунок 3.102 - Табличний вивід

Клас **table-bordered** створює кордону для таблиці і її комірок, а клас **table-hover** дозволяє виділяти рядок при наведенні на неї курсором:

```
<table class="table table-bordered table-hover">
  <tbody>
    .....
  </tbody>
</table>
```



Application name	Процент
C	17.5%
Java	16.4%
Objective-C	12.1%
C++	6.3%
C#	5.6%

Рисунок 3.103 - Табличний вивід

Отже, ми розглянули тільки невелику частину можливостей, які пропонує Bootstrap. Але поза сумнівом є те, що використання даного фреймворка дозволяє швидше створювати адаптивні веб-додатки і розширює арсенал веб-розробника.

3.14.6 Модальні вікна

Фреймворк Bootstrap пропонує прекрасні можливості по створенню модальних вікон. За своєю функціональністю вони нагадують модальні вікна в jQuery UI.

Для використання модальних вікон створимо новий проект. Наш проект буде використовувати модель, що описує модель комп'ютера:

```
public class Computer
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Company { get; set; }
    public int Year { get; set; }
}
```

Змінимо контролер HomeController наступним чином:

```
public class HomeController : Controller
{
    static List<Computer> comps=new List<Computer>();
    static HomeController()
    {
        comps.Add(new Computer { Id = 1, Name = "Apple II", Company = "Apple", Year = 1977 });
        comps.Add(new Computer { Id = 2, Name = "Macintosh", Company = "Apple", Year = 1983 });
        comps.Add(new Computer { Id = 3, Name = "IBM PC", Company = "IBM", Year = 1981 });
        comps.Add(new Computer { Id = 4, Name = "Altair", Company = "MITS", Year = 1975 });
    }
    public ActionResult Index()
    {
        return View(comps);
    }
    public ActionResult Details(int id)
    {
        Computer c = comps.FirstOrDefault(com => com.Id == id);
        if(c!=null)
            return PartialView(c);
        return HttpNotFound();
    }
}
```

Є простий метод Index, який повертає стандартне представлення зі списком, і є метод Details, який генерує часткове представлення з даними про обрану модель. Це часткове представлення і буде використовуватися модальним вікном. Тепер створимо представлення. Змінимо представлення Index.html наступним чином:

```
@model IEnumerable<ModalBootstrap.Models.Computer>
```

```

@{
    ViewBag.Title = "Комп'ютери";
}
<h3>Список моделей</h3>
<div id="modDialog" class="modal fade" >
    <div id="dialogContent" class="modal-dialog"></div>
</div>
<ul>
    @foreach (var c in Model)
    {
        <li>@Html.ActionLink(c.Name, "Details",
            new { id = c.Id }, new { @class = "compltem" })</li>
    }
</ul>
@section scripts
{
    <script type="text/javascript">

        $(function () {
            $.ajaxSetup({ cache: false });
            $(".compltem").click(function (e) {

                e.preventDefault();
                $.get(this.href, function (data) {
                    $('#dialogContent').html(data);
                    $('#modDialog').modal('show');
                });
            });
        })
    </script>
}

```

Натискання на будь-який елемент списку призведе до того, що спрацює обробник, визначений у кодi javascript. Цей обробник буде завантажувати вміст часткового представлення, яке ми трохи пізніше визначимо. Дані завантажуються в елемент з **id = "dialogContent"**. І щоб цей елемент розцінювався як модальне діалогове вікно, у нього встановлюється клас **modal-dialog**. Цей блок визначено всередині блоку **modDialog**, для якого встановлені класи **modal fade**. Тепер створимо часткове представлення **Details.cshtml**, яке буде використовуватися модальним вікном:

```

@model ModalBootstrap.Models.Computer
<div class="modal-content">
    <div class="modal-header">
        <button class="close" data-dismiss="modal" area-hidden="true">X</button>
        <h4>Модель комп'ютера</h4>
    </div>
    <div class="modal-body">
        <dl class="dl-horizontal">
            <dt>Назва моделі:</dt>
            <dd>@Html.DisplayFor(model => model.Name)</dd>

```

```

<dt>Виробник</dt>
<dd>@Html.DisplayFor(model => model.Company)</dd>
<dt>Рік випуску</dt>
<dd>@Html.DisplayFor(model => model.Year)</dd>
</dl>
</div>
</div>

```

Тут знову ж таки використовуються класи bootstrap для формування вмісту модального вікна.

Весь вміст поміщається в блок `div` з класом `modal-content`. Для створення заголовної частини модального вікна використовується блок з класом `modal-header`. Зокрема крім власне заголовка ми можемо визначити кнопку закриття вікна, присвоївши їй клас `close`. І також тут вказуємо за допомогою атрибута **`data-dismiss`** елемент, який повинен "закриватися" - тобто блок `div` з класом **`modal`**. Безпосередньо вміст розміщується в блок **`modal-body`**. Тепер ми можемо протестувати додаток. При натисканні на будь-яке посилання на моделі має з'явитися модальне вікно з докладною інформацією:

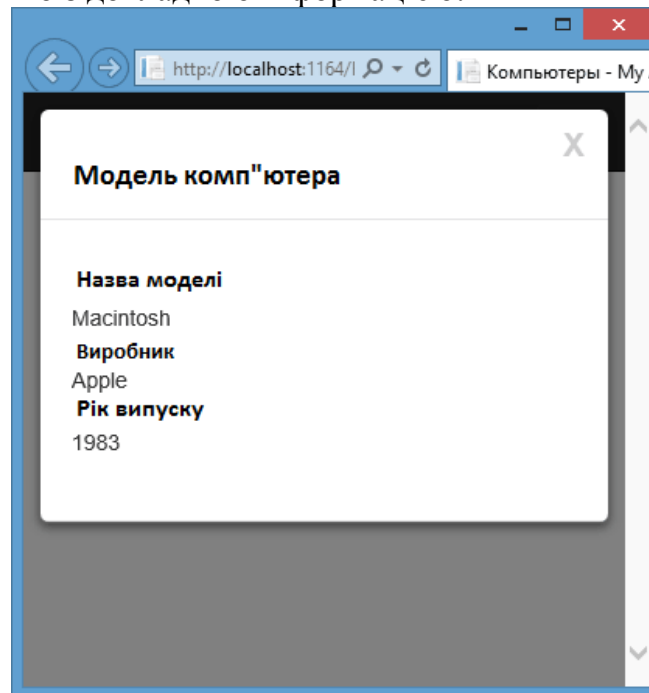


Рисунок 3.104 - Модальне вікно

І при натисканні на хрестик, модальне вікно закриється. Таким чином, ми можемо використовувати bootstrap для створення модальних вікон.

Контрольні питання.

1. Переваги використання Bootstrap.
2. Основні компоненти Bootstrap.
3. Принципи позиціонування елементів.
4. Принципи роботи з таблицями.
5. Модальні вікна Bootstrap.

3.15 SignalR 2

3.15.1 Вступ до SignalR 2

Останнім часом в веб-середовищі все частіше створюються веб-додатки, що використовують технології комунікацій в реальному часі: це і прості чати, і більш складні, розраховані на багато користувачів відеоконференції. Такі додатки можуть використовувати різні прийоми роботи: технологію Web Socket, опитування long polling і т.д. Для спрощення роботи з комунікаціями реального часу була створена спеціальна бібліотека під назвою SignalR.

При цьому SignalR не є бібліотекою, призначеною тільки для використання в ASP.NET MVC. Її також можна використовувати і в веб-формах, а також в консольних додатках, десктопних додатках і в WPF.

SignalR надає простий API для створення функціоналу, який дозволяє викликати функції JavaScript на стороні клієнта з серверного коду, написаного за допомогою мов платформи .NET. SignalR значно спрощує роботу з комунікаціями реального часу. Бібліотека обробляє всі підключення і автоматично розсилає повідомлення всім підключеним клієнтам, або яким-небудь специфічним клієнтам.

Фактично бібліотека SignalR складається з API серверної сторони, який застосовується в коді на C#, і з клієнтських бібліотек JavaScript. SignalR надає розробникам дві моделі: постійні підключення (**Persistent Connection**) і хаби (**Hubs**). Постійні підключення (Persistent Connection API) представляють розробникам прямий доступ до низкорівневого протоколу комунікацій.

Підключення в цій моделі представляють кінцеву точку, до якої підключаються клієнти, на зразок моделі підключень в WCF.

Хаби ж надають протокол взаємодії вищого рівня. Вони представляють верхній шар над Persistent Connection API і дозволяють клієнту і серверу безпосередньо викликати методи один одного. Якщо ви будете працювати з SignalR, то швидше за все будете переважно використовувати хаби, як більш зручні інструменти.

Підтримувані платформи

SignalR підтримується найбільш поширеними серверними платформами на базі ОС Windows:

- Windows Server 2012
- Windows Server 2008 r2
- Windows 8
- Windows 7
- Windows Azure

Для роботи на стороні сервера також необхідна версія фреймворку .net від 4.0 і вище. Підтримувані клієнтські платформи (браузери) теж різноманітні:

- Microsoft Internet Explorer 8, 9, 10, 11. Версії Modern, Desktop і Mobile
- Mozilla Firefox: як на ОС Windows, так і на Mac OS
- Google Chrome: як на ОС Windows, так і на Mac OS
- Safari: як на ОС Windows, так і на Mac OS

- Opera: ОС Windows
- Android-браузер

Також слід враховувати, що браузер повинен підтримувати бібліотеку jQuery 1.6.4 або більш пізні версії.

Транспорт передачі даних

Для обміну даними між клієнтом і сервером SignalR використовує той спосіб передачі або той транспорт, який найбільш підходить до даної ситуації. Однак розробники можуть мати найвищий пріоритет спосіб передачі. SignalR надає наступні типи технологій для взаємодії сервера і клієнта:

- WebSockets
- Server-sent events
- Forever Frames
- Long polling

Вибір транспорту

При створенні підключення SignalR вибирає, якщо є, технологію WebSocket, так як це найбільш оптимальна технологія для SignalR, яка найбільш ефективно використовує пам'ять сервера. У той же час WebSocket може використовуватися тільки на серверах під керуванням Windows Server 2012 або Windows 8 і при наявності встановленого .NET Framework 4.5. При цьому також технологія WebSocket повинна підтримуватися і браузером клієнта. І якщо технологія WebSocket недоступна на сервері або клієнті, то вибирається інший транспорт. Якщо WebSocket недоступний, то SignalR використовує технологію Server Sent Events, за її підтримки сервером і клієнтом. Якщо технологія Server Sent Events також недоступна, то використовуються приховані фрейми - Forever Frames. І, врешті, якщо Forever Frames також недоступні, то використовується Long Polling. Наприклад, якщо на стороні клієнта використовується браузер IE 8 і нижче, то використовується Long Polling. Тепер розглянемо роботу з бібліотекою на конкретних прикладах.

3.15.2 Перша програма з SignalR 2

Створимо додаток з використанням SignalR - невеликий чат. Отже, створимо новий додаток ASP.NET MVC5 з типом аутентифікації **No Authentication**. Назвемо свій додаток SignalRMvc. Після створення проекту знайдемо через пакетний менеджер NuGet бібліотеку Microsoft ASP.NET SignalR. Встановимо її. Після цього в папку бібліотек References буде додано ряд бібліотек SignalR, а в каталог скриптів Scripts буде додано клієнтський скрипт **jquery.signalR-[номер_версії].js** і його мінімізований аналог.

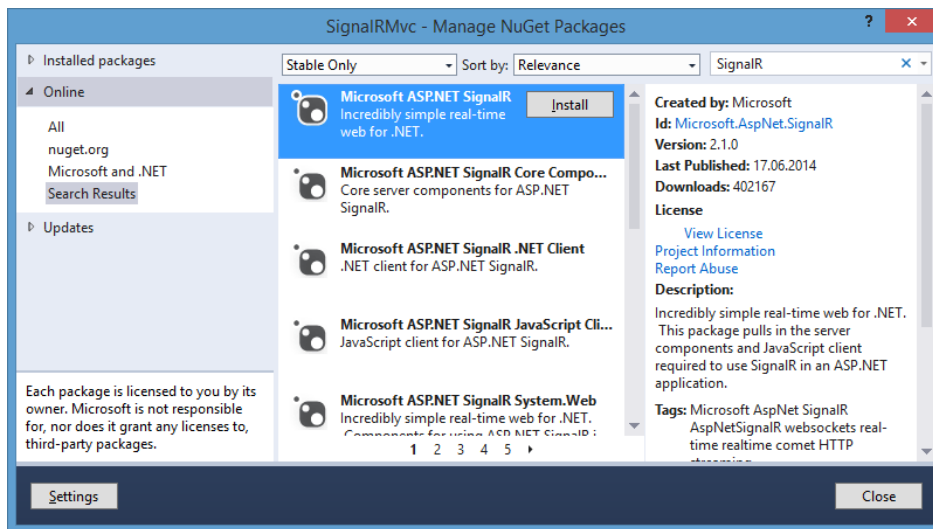


Рисунок 3.105 - Установка бібліотеки

Відразу після установки бібліотеки, щоб задіяти функціональність SignalR, додамо в проект наступний клас з назвою Startup:

```
using Microsoft.Owin;
using Owin;
[assembly: OwinStartup(typeof(SignalRMvc.Startup))]
namespace SignalRMvc
{
    public class Startup
    {
        public void Configuration(IAppBuilder app)
        {
            app.MapSignalR();
        }
    }
}
```

Для цього можна додати звичайний клас, а можна і спеціальний тип файлів OWIN Startup Class:

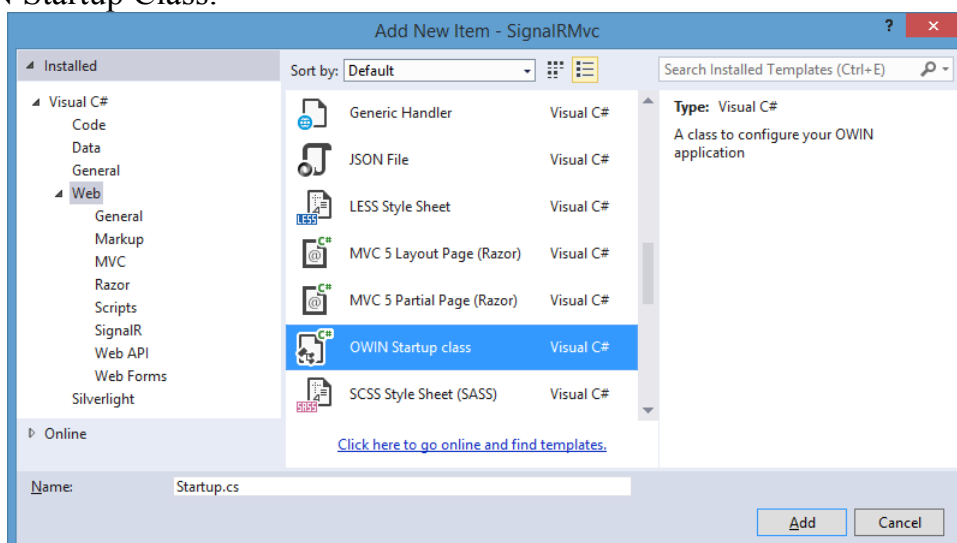


Рисунок 3.106 - Додавання класу

Так як додаток чату оперує користувачами, то створимо модель користувачів. Додамо в папку Models клас User:

```
public class User
{
    public string ConnectionId { get; set; }
    public string Name { get; set; }
}
```

Тепер додамо в проект нову папку. Назвемо її Hubs. У ній будуть знаходитися хаби нашої програми. У цю папку додамо новий клас ChatHub.cs. Знову ж ми можемо додати або просто клас, або скористатися вже готовим шаблоном файлу SignalR Hub Class (v2):

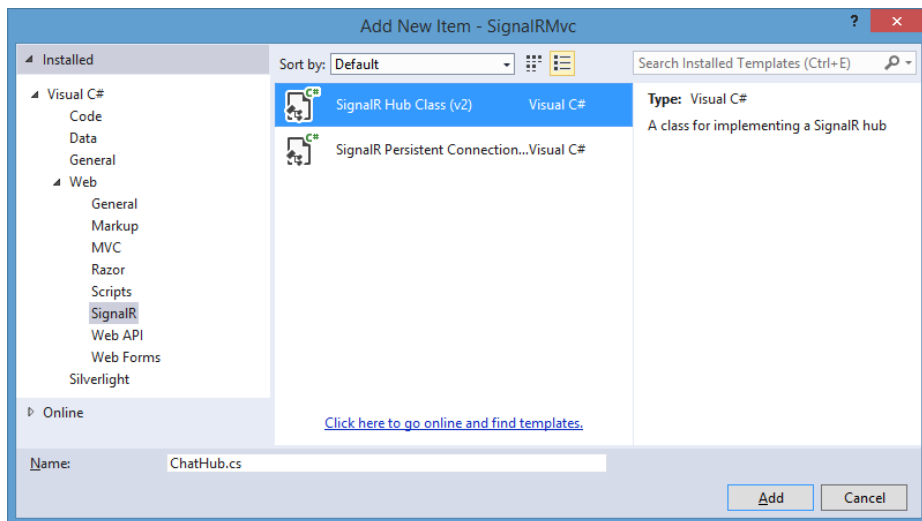


Рисунок 3.107 - Клас шаблону SignalR Hub Class

Новий клас матиме наступний код:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using Microsoft.AspNet.SignalR;
using SignalRMvc.Models;

namespace SignalRMvc.Hubs
{
    public class ChatHub : Hub
    {
        static List<User> Users = new List<User>();
        // Відправлення повідомлень
        public void Send(string name, string message)
        {
            Clients.All.sendMessage(name, message);
        }

        // Підключення нового користувача
    }
}
```

```

public void Connect(string userName)
{
    var id = Context.ConnectionId;
    if (!Users.Any(x => x.ConnectionId == id))
    {
        Users.Add(new User { ConnectionId = id, Name = userName });
        // Надсилаємо повідомлення поточному користувачу
        Clients.Caller.onConnected(id, userName, Users);
        // Надсилаємо повідомлення всім користувачам крім поточного
        Clients.AllExcept(id).onNewUserConnected(id, userName);
    }
}

// Відключення користувача
public override System.Threading.Tasks.Task OnDisconnected(bool stopCalled)
{
    var item = Users.FirstOrDefault(x => x.ConnectionId == Context.ConnectionId);
    if (item != null)
    {
        Users.Remove(item);
        var id = Context.ConnectionId;
        Clients.All.onUserDisconnected(id, item.Name);
    }
    return base.OnDisconnected(stopCalled);
}
}

```

Як вже зазначалось, SignalR використовує дві моделі взаємодії сервера і клієнта: Persistent Connection і хаби. В даному випадку ми використовуємо хаби. Для цього створюємо свій хаб ChatHub, який успадковується від класу Hub. По-перше, ми створюємо список, який буде зберігати підключених до чату користувачів. Далі у нас визначено ряд методів для відправки повідомлень, підключення та відключення користувачів. Розберемо метод Send, який призначений для цього. В ньому викликається єдиний метод: Clients.All.sendMessage (name, message);. Об'єкт Clients означає колекцію всіх користувачів хаба. Властивість All, що йде далі, говорить про те, що метод треба застосувати до всіх підключених клієнтів.

Формат виклику методів клієнта:

Виклик методу на всіх клієнтах: Clients.All.sendMessage (name, message);

- Виклик методу тільки на поточному клієнта, який звернувся до сервера: Clients.Caller.sendMessage (name, message);
- Виклик методу на всіх клієнтах, крім того, який звернувся до сервера: Clients.Others.sendMessage (name, message);
- Виклик методу тільки у клієнта з певним id: Clients.Client(Context.ConnectionId).sendMessage (name, message);
- Виклик методу на всіх клієнтах, крім клієнта з певним id: Clients.AllExcept(connectionId).sendMessage (name, message);

- Виклик методу на всіх клієнтах зазначеної групи: `Clients.Group(groupName).addMessage(name, message);`
- Виклик методу на всіх клієнтах зазначеної групи, за винятком клієнта, у якого `id` - `connectionId`: `Clients.Group(groupName, connectionId).addMessage(name, message);`
- Виклик методу на всіх клієнтах зазначеної групи, за винятком звернувся до сервера клієнта: `Clients.OthersInGroup(groupName).addMessage(name, message);`

Залежно від того, кому треба передати повідомлення, ми можемо вибрати один з варіантів. Далі в вираженні слід метод `addMessage`. Цей метод оголошується на стороні клієнта в коді javascript. Трохи пізніше ми додамо код клієнтської сторони. А поки просто треба знати, що ці методи знаходяться не на серверній частині, а на стороні клієнта.

У методі `Connect` ми спочатку отримуємо `id` поточного користувача, який і звернувся до методу `Connect`, через об'єкт `Context.ConnectionId`. Цей `id` задається середовищем і зберігає рядкове значення (або не числове). Потім викликаємо методи на клієнті через об'єкт `Clients`.

В кінці переобумовленої метод `OnDisconnected`, який виконує відключення поточного клієнта в асинхронному режимі. Наприклад, при закритті вкладки браузера клієнт по суті виходить з програми і викликається метод `OnDisconnected` для поточного клієнта.

Тепер створимо представлення для чату. За замовчуванням в проєкті вже є контролер `HomeController` з декількома методами, для яких визначені представлення. Ми можемо використовувати, наприклад, представлення `Index.cshtml`. Змінимо його вміст в такий спосіб:

```
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <h2>Чат-кімната</h2>
    <div class="main">
        <div id="loginBlock">
            Введіть логін:<br />
            <input id="txtUserName" type="text" />
            <input id="btnLogin" type="button" value="Ввійти" />
        </div>
        <div id="chatBody">
            <div id="header"></div>
            <div id="inputForm">
                <input type="text" id="message" />
                <input type="button" id="sendmessage" value="Відправити" />
            </div>
        </div>
    </div>
</body>
</html>
```

```

</div>
<div id="chatroom"></div>
<div id="chatusers">
  <p><b>Всі користувачі</b></p>
</div>
</div>
<input id="hId" type="hidden" />
<input id="username" type="hidden" />
</div>
<script src="~/Scripts/jquery-1.10.2.min.js"></script>
<!--Ссылка на библиотеку SignalR -->
<script src="~/Scripts/jquery.signalR-2.2.0.min.js"></script>
<!--Посилання на автоматично згенерований скрипт хаба SignalR -->
<script src="~/signalr/hubs"></script>
<script src="~/Scripts/util.js"></script>
</body>
</html>

```

Розмітка містить по суті два блоки: login Block (блок введення логіна) та chatBody (сам чат). В один момент часу видимий тільки один блок, тому, якщо користувач вдало підключився, то ми перемикаємо видимість, роблячи видимим блок chatBody.

Блок chatBody містить три підблоки: inputForm (форма написання повідомлення), chatroom (поле, куди будуть додаватися повідомлення) і chatusers (список всіх користувачів, крім поточного).

Основні дії на клієнтській частині буде виконувати код javascript. Причому перед підключенням всіх інших файлів і скриптів йде підключення бібліотеки jQuery, оскільки скрипт SignalR має залежності від jQuery.

Далі підключається сама бібліотека SignalR. Конкретна версія бібліотеки може відрізнятися від використаної вище. Потім підключається скрипт signalr / hubs, хоча насправді поки його немає, але в процесі запуску програми він буде автоматично генеруватися і після цього використовуватися.

Останнім йде підключення скрипта util.js, якого поки що немає. Отже, додамо файл з даними назвою в папку Scripts і визначимо в ньому наступний код:

```

$(function () {
  $('#chatBody').hide();
  $('#loginBlock').show();
  // Посилання на автоматично-згенерований проксі хаба
  var chat = $.connection.chatHub;
  // Оголошення функції, який хаб викликає при отриманні повідомлень
  chat.client.addMessage = function (name, message) {
    // Додавання повідомлень на веб-сторінку
    $('#chatroom').append('<p><b>' + encodeURIComponent(name)
      + '</b>:' + encodeURIComponent(message) + '</p>');
  };
  // Функція, що викликається при підключенні нового користувача
  chat.client.onConnected = function (id, userName, allUsers) {

```

```

$('#loginBlock').hide();
$('#chatBody').show();
// установка в схованих полях імені та id поточного користувача
$('#hId').val(id);
$('#username').val(userName);
$('#header').html('<h3>Ласкаво просимо, ' + userName + '</h3>');
// Додавання всіх користувачів
for (i = 0; i < allUsers.length; i++) {
    AddUser(allUsers[i].ConnectionId, allUsers[i].Name);
}
}

// Додаємо нового користувача
chat.client.onNewUserConnected = function (id, name) {
    AddUser(id, name);
}

// Видаляємо користувача
chat.client.onUserDisconnected = function (id, userName) {
    $('##' + id).remove();
}

// Відкриваємо з'єднання
$.connection.hub.start().done(function () {
    $('#sendmessage').click(function () {
        // Викликаємо у хаба метод Send
        chat.server.send($('#username').val(), $('#message').val());
        $('#message').val('');
    });

    // обробка логіна
    $("#btnLogin").click(function () {
        var name = $("#txtUserName").val();
        if (name.length > 0) {
            chat.server.connect(name);
        }
        else {
            alert("Введіть ім'я");
        }
    });
});

// Кодування тегів
function htmlEncode(value) {
    var encodedValue = $('<div />').text(value).html();
    return encodedValue;
}

//Додавання нового користувача
function AddUser(id, name) {
    var userId = $('#hId').val();

```

```

if (userId != id) {
    $('#chatusers').append('<p id="' + id + '"><b>' + name + '</b></p>');
}
}

```

Весь код на стороні клієнта знаходиться в функції jQuery, за її межами визначені дві функції - `htmlEncode` (для кодування тегів, щоб припинити можливі спроби вставок коду javascript і інші проблеми) і `AddUser` (для додавання даних користувачів в список).

Щоб взаємодіяти з хабом, отримуємо проксі хаба:

`var chat = $.connection.chatHub;`. Потім визначаємо ряд функцій клієнта `chat.client.addMessage`, `chat.client.onConnected` і т.д. Вище в хабі в коді C# у нас було визначено звернення до функцій клієнта: `Clients.All.addMessage(name, message);`. Функція `addMessage` - це і є функція, визначена для `chat.client.addMessage`. Подібним чином ми можемо звертатися на сервері і до інших функцій клієнта.

Щоб з'єднатися, ми викликаємо метод `$.connection.hub.start().Done()`, передаючи в нього функцію. У цій функції ми вішаємо обробники кнопок, після натискання на які відбувається звернення на сервер.

Наприклад, відправка повідомлення проводиться за допомогою виклику `chat.server.send($('#username').Val(), $('#message').Val())`; . Вираз `chat.server` є звернення до методів хаба на сервері. Тобто в даному випадку буде йти звернення до методу `public void Send (string name, string message)`, визначеного в класі `ChatHub`.

Подібним чином спрацьовує виклик методу `chat.server.connect (name)`; . Зверніть увагу, що хоча в коді C# методи оголошені з великої літери, в коді javascript в їх назві використовується мала літера.

Тепер ми можемо перевірити в дії. Після запуску нам буде запропоновано ввести логін. Після успішного логіна форма введення логіна стане невидимою, зате стане доступна форма самого чату. Можна відкрити декілька вкладок в браузері з даними додатком, підключивши тим самим кілька клієнтів.

3.15.3 Persistent Connection API

`Persistent Connection API` пропонує нам низькорівневий інтерфейс для створення додатків. Можливо, багатьом він здасться менш зручним, ніж використання хабів, але він теж може використовуватися як варіант побудови програми. Отже, створимо проект програми, з якою буде використовувати `Persistent Connection`.

Створимо проект ASP.NET 5 з типом аутентифікації `No Authentication`. Назвемо його `PersistentConMvc`. Потім через менеджер NuGet додамо в проект бібліотеку `SignalR`.

Потім додамо в папку `Models` модель `Data`. Ця модель буде представляти об'єкт повідомлення, що передається. В принципі ми могли б обмінюватися простими рядковими об'єктами. Однак використання більш складних об'єктів

дозволить нам структурувати дані, що передаються, і легше ними управляти. Отже, модель Data матиме наступний код:

```
using System;
namespace PersistentConMvc.Models
{
    public class Data
    {
        public string Name { get; set; } // ім'я користувача
        public string Message { get; set; } // Повідомлення користувача
    }
}
```

Цей об'єкт буде передаватися від користувача сервера. Тепер створимо серверну інфраструктуру для взаємодії з клієнтом. Створимо в проєкті папку Connection і додамо в неї новий клас ChatConnection:

```
using System;
using System.Collections.Generic;
using System.Web;
using Microsoft.AspNet.SignalR;
using PersistentConMvc.Models;
using System.Threading.Tasks;
using Newtonsoft.Json;

namespace PersistentConMvc.Connection
{
    public class ChatConnection : PersistentConnection
    {
        protected override Task OnConnected(IRequest request, string connectionId)
        {
            Data chatData = new Data() { Name = "Повідомлення сервера", Message = "Користувач " + connectionId + " приєднався до чату" };
            return Connection.Broadcast(chatData);
        }

        protected override Task OnReceived(IRequest request, string connectionId, string data)
        {
            Data chatData = JsonConvert.DeserializeObject<Data>(data);
            return Connection.Broadcast(chatData);
        }

        protected override Task OnDisconnected(IRequest request, string connectionId)
        {
            Data chatData = new Data() { Name = "Повідомлення сервера", Message = "Користувач " + connectionId + " покинув чат" };
            return Connection.Broadcast(chatData);
        }
    }
}
```

При створенні класу ми можемо вибрати вже наявний шаблон файлу:

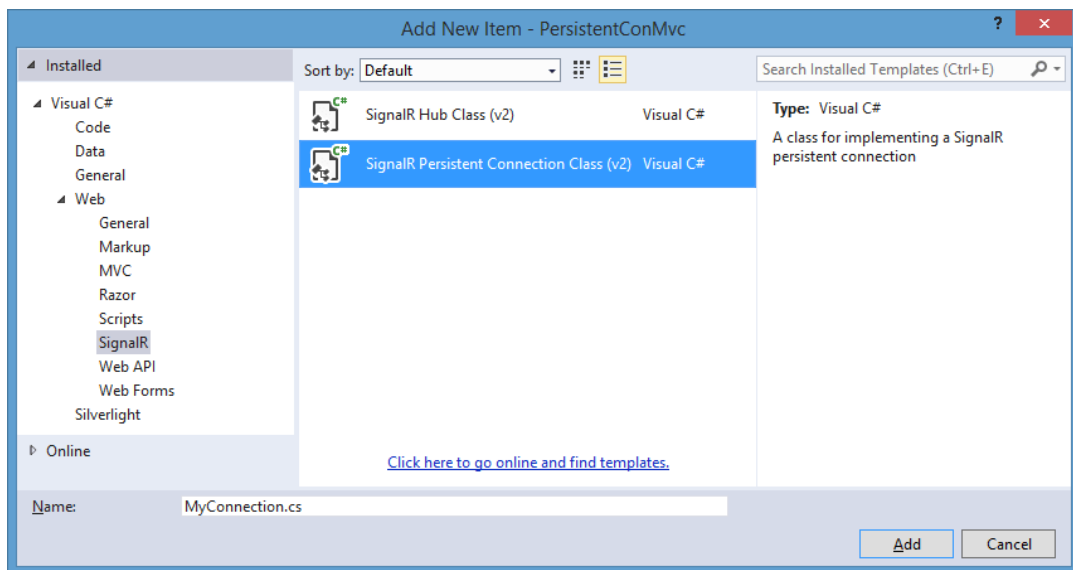


Рисунок 3.108 - Вибір шаблону файлу

Клас Chat Connection успадковується від класу Persistent Connection і перевизначає деякі його методи. Метод OnConnected спрацьовує при кожному підключенні клієнта. У нього передаються два параметри: IRequest request, який представляє об'єкт запиту, і connectionId, який представляє id, автоматично присвоюються клієнту який звернувся до сервера.

Тут ми надсилаємо клієнту в методі Connection.Broadcast об'єкт Data. Метод Broadcast може приймати будь-який об'єкт, в тому числі і просту рядок, головне, щоб в коді клієнта ми могли розпарсити належним чином це повідомлення. Метод OnDisconnected аналогічний методу OnConnected, тільки виконується при відключенні клієнта.

При отриманні повідомлення від клієнта спрацьовує метод OnReceived: отримане повідомлення передається в третьому параметрі data. Оскільки це повідомлення являє собою рядок, ми його десеріалізуємо в об'єкт Data і потім розсилаємо його всім підключеним клієнтам.

Щоб підключити функціональність SignalR, додамо в проект новий клас Startup:

```
using System;
using System.Threading.Tasks;
using Microsoft.Owin;
using Owin;
[assembly: OwinStartup(typeof(PersistentConMvc.Startup))]
namespace PersistentConMvc
{
    public class Startup
    {
        public void Configuration(IApplicationBuilder app)
        {
            app.MapSignalR<ChatConnection>("/chat");
        }
    }
}
```

```
}
```

В даному випадку ми реєструємо маршрут, по якому потім будемо звертатися з клієнтської частини.

Тепер клієнтська частина. Так як в проєкті за замовчуванням є контролер HomeController, то ми можемо використовувати призначені для його методів представлення. Змінимо представлення Index.cshtml наступним чином:

```
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <h2>Чат</h2>

    <div id="loginBlock">
        Введіть логін:<br />
        <input id="txtUserName" type="text" />
        <input id="btnLogin" type="button" value="Ввійти" />
    </div>
    <div id="chatBody">
        <div id="header"></div>
        <div id="inputForm">
            <input type="text" id="message" />
            <input type="button" id="sendMessage" value="Відправити" />
        </div>
        <div id="chatroom">
            <ul></ul>
        </div>

    </div>
    <input id="username" type="hidden" />

    <script src="~/Scripts/jquery-1.10.2.min.js"></script>
    <!--Посилання на бібліотеку SignalR -->
    <script src="~/Scripts/jquery.signalR-2.1.0.min.js"></script>
    <script>
        $(function () {
            $('#chatBody').hide();
            // одержуємо з'єднання
            var myConnection = $.connection("/chat");
            // обробка одержаних даних від сервера
            myConnection.received(function (data) {
                $("#chatroom ul").append("<li><strong>" + htmlEncode(data.Name) +
                    '</strong>: ' + htmlEncode(data.Message) + "</li>");
            });
        });
    </script>
</body>
</html>
```

```

// обробка логіна
$("#btnLogin").click(function () {

    var userName = $("#txtUserName").val().replace(/s/g, "");
    if (userName.length > 0) {
        $('#username').val(userName);

        //відключаємо поля вводу логіна
        $('#btnLogin').attr('disabled', 'disabled');
        $('#txtUserName').attr('disabled', 'disabled');

        $('#chatBody').show();
        // відкриваємо з'єднання
        myConnection.start().done(function () {
            //обробник відправки повідомлення
            $('#sendmessage').click(function () {
                // надсилаємо серіалізований об'єкт на сервер
                myConnection.send(JSON.stringify({ name: $('#username').val(), message:
$('#message').val() }));
                $('#message').val("");
            });
        });
    }
    else {
        alert("Введіть ім'я");
    }
});

});
// Кодування тегів
function htmlEncode(value) {
    var encodedValue = $('<div />').text(value).html();
    return encodedValue;
}
</script>
</body>
</html>

```

Отже, розмітка html містить два блоки: блок введення ника login Block і блок самого чату chatBody, в якому є форма написання повідомлення і список для отриманих повідомлень.

У коді javascript насамперед ми створюємо підключення: `var myConnection=$.connection("/chat");`. Як параметр вказується той маршрут, який ми вище зареєстрували в файлі Global.asax.

Функція `myConnection.received` викликається при отриманні клієнтом від сервера даних. В даному випадку оскільки сервер передає клієнтові об'єкт Data, у якого є властивості Name і Message, то ми можемо їх отримати в даній функції і додати на сторінку в список повідомлень.

Далі визначено основну логіку. Її суть: після введення ніка користувачем відкривається з'єднання з сервером, і користувач може відправляти повідомлення. Спочатку ми отримуємо введений нік, прибираємо з нього всі пробіли, і якщо він проходить перевірку, додаємо його в приховане поле, з якого нік потім буде додаватися до повідомлення, що відправляється на сервер. Щоб користувач не міг вдруге ввести нік і підключитися, відключаємо поля введення і нарешті відкриваємо з'єднання (`myConnection.start ()`). `Done (...)`.

Для відправлення на сервер використовується метод `myConnection.send`, в який в якості параметра передаються введені повідомлення і збережений в прихованому полі нік. За допомогою функції `JSON.stringify` дані перетворюються в об'єкт `json`, який потім сервер витягує в методі `protected override Task OnReceived (IRequest request, string connectionId, string data)` з останнього параметра.

3.15.4 Малювання на SignalR

Чат - найбільш показове рішення, яке можна зробити за допомогою SignalR. Однак чатами все не обмежується. Спробуємо створити графічний додаток з використанням SignalR, в якому є загальне полотно, і кожен підключений до додатка може щось намалювати, і всі будуть це бачити.

Отже, створимо проект ASP.NET MVC5 з типом аутентифікації No Authentication. Назвав свій проект SignalRDraw. І відразу ж через менеджер NuGet додамо в проект бібліотеку SignalR.

Відразу ж створимо клієнта для програми. Для цього додамо в проект новий файл `index.html` прямо в корінь проекту. Потім натиснемо на цей файл правою кнопкою миші і в контекстному меню виберемо `Set as Start Page` (Зробити початковою сторінкою). Таким чином, ми не будемо використовувати ні контролери, ні їхні представлення. Весь функціонал клієнта у нас буде містити ця веб-сторінка. А каталоги `Controllers` і `Views` в принципі в загальному можна і видалити. Тепер змінимо код файлу `index.html` на наступний:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title></title>
  <style>
    canvas {
      position: relative;
      background-color:#ffd800;
    }
  </style>
</head>
<body>
  <canvas id='drawingpad' width='400' height='300'></canvas>
  <script src="/Scripts/jquery-1.10.2.min.js"></script>
  <!--Посилання на бібліотеку SignalR -->
  <script src="/Scripts/jquery.signalR-2.1.0.min.js"></script>
```

```

<!-- Посилання на автоматично згенерований скрипт хаба SignalR -->
<script src="/signalr/hubs"></script>
<script>

$(function () {

    var drawGame = {
        // вказує, чи відбувається відмальовка
        isDrawing: false,
        // початкова точка наступної лінії
        startX: 0,
        startY: 0,
    };
    // модель лінії
    var data = {
        startX: 0,
        startY: 0,
        endX: 0,
        endY: 0
    };
    // контекст елемента canvas
    var canvas = document.getElementById('drawingpad');
    var ctx = canvas.getContext('2d');

    // Посилання на автоматично-згенероване проксі хаба
    var chat = $.connection.drawHub;
    // Оголошення функції, яку хаб викликає коли одаержав повідомлення
    chat.client.addLine = function (data) {

        // Додавання ліній
        drawLine(ctx, data.startX, data.startY, data.endX, data.endY, 1);
    };

    // Відкриваємо з'єднання
    $.connection.hub.start().done(function () {
        // після відкриття з'єднання встановлюємо обробник миші
        canvas.addEventListener("mousedown", mousedown, false);
        canvas.addEventListener("mousemove", mousemove, false);
        canvas.addEventListener("mouseup", mouseup, false);
    });
    // просто малюємо лінію
    function drawLine(ctx, x1, y1, x2, y2, thickness) {
        ctx.beginPath();
        ctx.moveTo(x1, y1);
        ctx.lineTo(x2, y2);
        ctx.lineWidth = thickness;
        ctx.strokeStyle = "#444";
        ctx.stroke();
    }
    // натиск миші
    function mousedown(e) {

```

```

        // одержуємо позиції x і y відносно верхнього лівого кутка елемента canvas
        var mouseX = e.layerX || 0;
        var mouseY = e.layerY || 0;
        drawGame.startX = mouseX;
        drawGame.startY = mouseY;
        drawGame.isDrawing = true;
    };

    // переміщення миші
    function mousemove(e) {

        // малюємо лінію, якщо isdrawing==true
        if (drawGame.isDrawing) {
            // одержуємо позиції x та y відносно верхнього лівого кутка елемента canvas
            var mouseX = e.layerX || 0;
            var mouseY = e.layerY || 0;
            if (!(mouseX == drawGame.startX &&
                mouseY == drawGame.startY)) {
                drawLine(ctx, drawGame.startX,
                    drawGame.startY, mouseX, mouseY, 1);
                data.startX = drawGame.startX;
                data.startY = drawGame.startY;
                data.endX = mouseX;
                data.endY = mouseY;
                chat.server.send(data);
                drawGame.startX = mouseX;
                drawGame.startY = mouseY;
            }
        }
    };

    function mouseup(e) {
        drawGame.isDrawing = false;
    }

    });
</script>
</body>
</html>

```

Розмітка html містить всього один елемент canvas - полотно для малювання. У коді javascript спочатку визначаються дві моделі: drawGame - для використання всередині клієнта і data - для взаємодії з сервером. Щоб непосредствено малювати примітиви на полотні, нам треба отримати canvas і його контекст:

```

var canvas = document.getElementById('drawingpad');
var ctx = canvas.getContext('2d');

```

Далі отримати посилання на хаб. В даному випадку клас хаба на сервері, який ми пізніше створимо, буде називатися DrawHub. А за допомогою функції addLine клієнт буде отримувати з сервера дані і по ним малювати лінії.

```
var chat = $.connection.drawHub;  
chat.client.addLine = function (data) {  
  
    drawLine(ctx, data.startX, data.startY, data.endX, data.endY, 1);  
};
```

Потім в функції \$.connection.hub.start().Done (function () ... додаються обробники подій миші, з допомогою яких можна малювати лінії і відправляти повідомлення про ці лінії на сервер.

Тепер визначимо на рівні сервера модель, яка буде представляти дані, одержувані від клієнта. Додамо в папку Models новий клас Data:

```
using System;  
using Newtonsoft.Json;  
  
namespace SignalRDraw.Models  
{  
    public class Data  
    {  
        [JsonProperty("startX")]  
        public int StartX { get; set; }  
        [JsonProperty("startY")]  
        public int StartY { get; set; }  
        [JsonProperty("endX")]  
        public int EndX { get; set; }  
        [JsonProperty("endY")]  
        public int EndY { get; set; }  
    }  
}
```

Кожна властивість для зв'язку з моделлю data, визначеної в javascript, має атрибут JsonProperty. Тепер додамо в проект прямо в корінь новий клас хаба:

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Web;  
using Microsoft.AspNet.SignalR;  
using SignalRDraw.Models;  
  
namespace SignalRDraw  
{  
    public class DrawHub : Hub  
    {  
        public void Send(Data data)  
        {
```

```

        Clients.AllExcept(Context.ConnectionId).addLine(data);
    }
}
}

```

За допомогою методу `Clients.AllExcept()` ми можемо виключити з розсилки повідомлень того клієнта, який власне і надіслав повідомлень, і таким чином, уникнути ситуації, коли у нього буде дублюватися намальована лінія. А метод `addLine` власне і виконує розсилку всім підключеним клієнтам і задіює на клієнті однойменний метод `addLine`, який виконує малювання. Також додамо в проект клас `Startup`, щоб задіяти `SignalR`:

```

using System;
using System.Threading.Tasks;
using Microsoft.Owin;
using Owin;
[assembly: OwinStartup(typeof(SignalRDraw.Startup))]
namespace SignalRDraw
{
    public class Startup
    {
        public void Configuration(IAppBuilder app)
        {
            app.MapSignalR();
        }
    }
}

```

У підсумку вийде наступна структура:

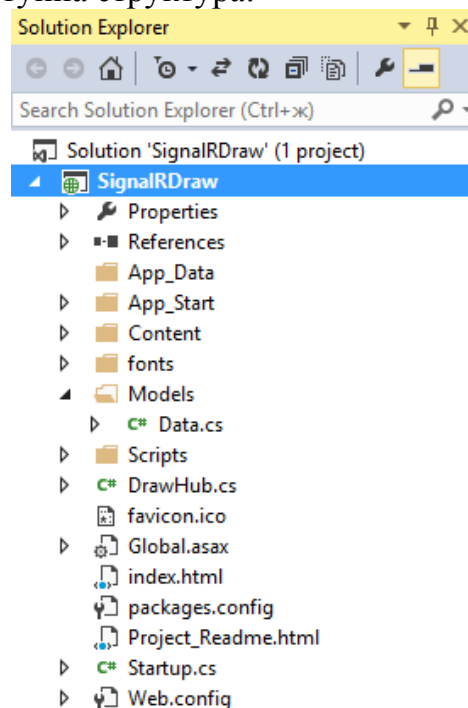


Рисунок 3.109 - Одержана структура проекту

Тепер ми можемо запустити проект і відкрити запущену сторінку в різних браузерях і протестувати додаток:

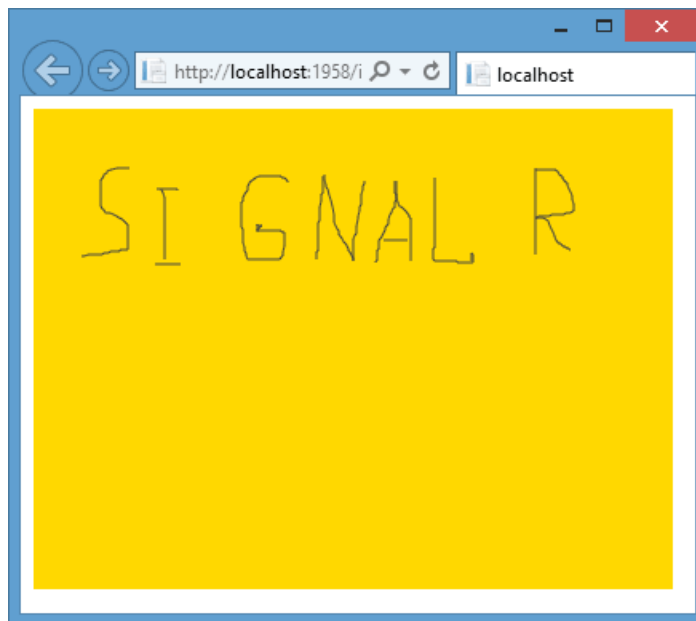


Рисунок 3.110 - Тестовий запуск програми

3.15.5 Push-повідомлення

Push-повідомлення дозволяють моментально сповістити користувача про якусь подію. За допомогою SignalR зробити подібний функціонал досить просто. Припустимо, ми хочемо, щоб при додаванні нового об'єкта в базу даних відбувалося повідомлення всіх користувачів. Для цього створимо проект ASP.NET MVC 5 з типом аутентифікації No Authentication, який буде називатися PushNotifyApp і через менеджер NuGet додамо в проект бібліотеку SignalR.

Додамо в корінь проекту клас Startup:

```
using Microsoft.Owin;  
using Owin;  
[assembly: OwinStartup(typeof(PushNotifyApp.Startup))]  
namespace PushNotifyApp  
{  
    public class Startup  
    {  
        public void Configuration(IAppBuilder app)  
        {  
            app.MapSignalR();  
        }  
    }  
}
```

Також додамо в проект папку Hub, яка буде містити хаби SignalR. І в ній визначимо клас хаба:

```
using Microsoft.AspNet.SignalR;  
namespace PushNotifyApp.Hubs  
{  
    public class NotificationHub : Hub  
    {
```

```
}  
}
```

Для створення повідомлень не треба визначати в класі ніяких методів. Можна залишити його порожнім. Тоді контролер буде виглядати наступним чином:

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Web;  
using System.Web.Mvc;  
using PushNotifyApp.Models;  
using PushNotifyApp.Hubs;  
  
namespace PushNotifyApp.Controllers  
{  
    public class HomeController : Controller  
    {  
        BookContext db = new BookContext();  
        public ActionResult Index()  
        {  
            return View(db.Books.ToList());  
        }  
  
        public ActionResult Create()  
        {  
            return View();  
        }  
        [HttpPost]  
        public ActionResult Create(Book book)  
        {  
            db.Books.Add(book);  
            db.SaveChanges();  
            SendMessage("Доданий новий об'єкт");  
            return RedirectToAction("Index");  
        }  
        private void SendMessage(string message)  
        {  
            // Одержуємо контекст хаба  
            var context =  
                Microsoft.AspNet.SignalR.GlobalHost.ConnectionManager.GetHubContext<NotificationHub>();  
            //Одержуємо повідомлення  
            context.Clients.All.displayMessage(message);  
        }  
    }  
}
```

Ключовим моментом тут є метод `SendMessage()`, в якому отримуємо контекст хаба і відправляємо повідомлення всім підключеним клієнтам. Нехай нам треба вивести це повідомлення у представленні `Index.cshtml`:

```
@model IEnumerable<PushNotifyApp.Models.Book>

@{
    ViewBag.Title = "Index";
}
<div><span id="notification" style="color:red;font-size:15px;"></span></div>
<h2>Список книг</h2>

<table class="table">
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.Name)
        </th>
    </tr>

    @foreach (var item in Model) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Name)
            </td>
        </tr>
    }
</table>
@section scripts{
<script src="~/Scripts/jquery.signalR-2.2.0.min.js"></script>
<!--Посилання на автоматично згенерований скрипт хаба SignalR -->
<script src="~/signalr/hubs"></script>
<script type="text/javascript">
    $(function () {

        var notificationhub = $.connection.notificationHub;
        notificationhub.client.displayMessage = function (message) {
            $('#notification').html(message);
        };
        $.connection.hub.start();
    });
</script>
}
```

Для виведення повідомлень визначається елемент ``.

Весь необхідний код `javascript` підключається в секції `scripts`. Передбачається, що дане представлення використовує майстер-сторінку `_Layout`, і на майстер-сторінці перед секцією `scripts` підключиться бібліотека `jquery`. Тепер після додавання нового об'єкта в методі `Create` всі підключені клієнти в представленні `Index.cshtml` зможуть отримати повідомлення і побачити його в блоці ``.

Контрольні питання.

1. Призначення бібліотеки SignalR .
2. Основні принципи передачі інформації в реальному часі.
3. Особливості роботи із графікою в SignalR.

3.16 Робота з jQuery UI і візуальними елементами

3.16.1 jQuery UI і Autocomplete

Крім стандартних елементів html ми можемо використовувати більш багаті по функціональності і графічній візуалізації елементи, які, як правило, реалізуються у вигляді додаткових плагінів на javascript і css. Найбільш популярним плагіном є бібліотека jQuery UI. Ця бібліотека містить набір таких віджетів, як accordion, autocomplete, button, datepicker, dialog, progressbar, slider і ряд інших.

Для роботи з jQuery UI необхідно перш за все підключити бібліотеку. Якщо в проектах для попередніх версій MVC файли jquery ui підключалися за замовчуванням, то в MVC 5 нам треба самим додавати всі необхідні файли в проект. Для цього скористаємося пакетним менеджером NuGet:

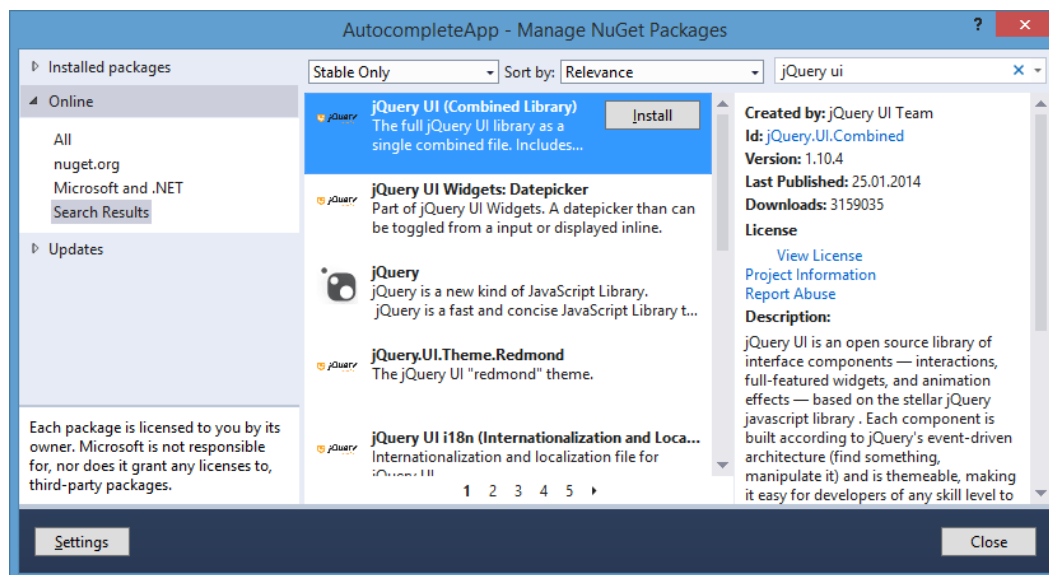


Рисунок 3.111 - Підключення jQuery UI

Необхідний пакет називається jQuery UI (Combined Library). Також потрібно враховувати, що дана бібліотека залежить від головної бібліотеки jQuery, яка також повинна бути встановлена.

Після установки в проект в каталог Content/themes/base буде додано ряд файлів css jquery ui, а в папку Scripts - її скрипт в повній і мінімізованій версії.

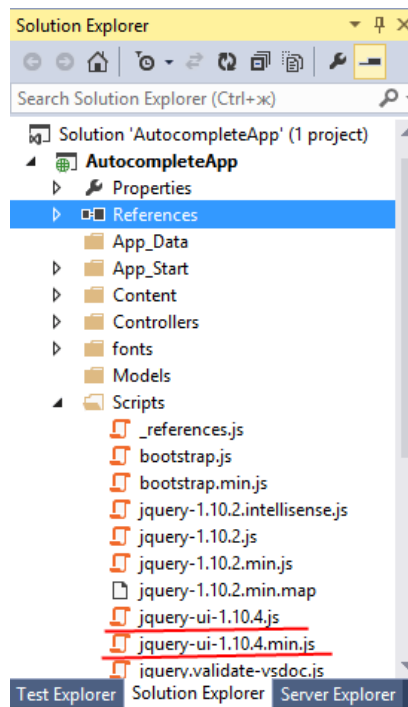


Рисунок 3.112 - Перевірка наявності пакету jquery-ui

Тепер щоб використовувати бібліотеку, нам треба в представленні підключити її основний файл стилів all.css, інакше плагін просто не буде працювати, а також наступний скрипт:

```
<link href='@Url.Content("~/Content/themes/base/all.css")' rel="stylesheet" type="text/css" />
<script src='@Url.Content("~/Scripts/jquery-ui-1.11.3.min.js")' type="text/javascript"></script>
```

Розглянемо дію jQuery UI на прикладі віджета Autocomplete, який реалізує функціональність автозаповнення. Для початку визначимо клас моделі, який буде використовуватися в додатку:

```
public class Computer
{
    public int Id { get; set; }
    public string Model { get; set; }
    public int Year { get; set; }
}
```

Потім змінюємо стандартний контролер HomeController наступним чином:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using AutocompleteApp.Models;

namespace AutocompleteApp.Controllers
{
    public class HomeController : Controller
```

```

{
    static List<Computer> comps = new List<Computer>();

    static HomeController()
    {
        comps.Add(new Computer { Id = 1, Model = "IBM PC", Year = 1981 });
        comps.Add(new Computer { Id = 2, Model = "Apple II", Year=1977 });
        comps.Add(new Computer { Id = 3, Model = "Apple III", Year = 1980 });
        comps.Add(new Computer { Id = 4, Model = "Macintosh", Year=1984 });
    }

    public ActionResult Index()
    {
        return View();
    }

    public ActionResult AutocompleteSearch(string term)
    {
        var models = comps.Where(a => a.Model.Contains(term))
            .Select(a => new { value = a.Model })
            .Distinct();

        return Json(models, JsonRequestBehavior.AllowGet);
    }
}

```

Для стислості прикладу в якості сховища даних ми використовуємо статичний список. Основна робота проводиться методом `AutocompleteSearch`, який ми розглянемо пізніше. Тепер змінимо представлення `Index.cshtml`, визначивши в ньому елемент вводу, для якого і буде здійснюватися автозаповнення:

```

@{
    Layout = null;
}
<html>
<head>
    <link href='@Url.Content("~/Content/themes/base/all.css")' rel="stylesheet" type="text/css" />
</head>
<body>
    <input type="text" name="name" data-autocomplete-
source='@Url.Action("AutocompleteSearch", "Home")' />
    <input type="submit" value="Поиск" />

    <script src='@Url.Content("~/Scripts/jquery-1.10.2.min.js")' type="text/javascript"></script>
    <script src='@Url.Content("~/Scripts/jquery-ui-1.11.3.min.js")' type="text/javascript"></script>
    <script type="text/javascript">
    $(function () {
        $('[data-autocomplete-source]').each(function () {

```

```

        var target = $(this);
        target.autocomplete({ source: target.attr("data-autocomplete-source") });
    });
</script>
</body>
</html>

```

Отже, щоб підключити автозаповнення, ми додаємо до елемента введення атрибут HTML5 `data-autocomplete-source`. Цей атрибут вказує на джерело даних. В даному випадку джерелом даних буде дія `AutocompleteSearch` в контролері `Home`.

```

public ActionResult AutocompleteSearch(string term)
{
    var models = comps.Where(a => a.Model.Contains(term))
        .Select(a => new { value = a.Model })
        .Distinct();

    return Json(models, JsonRequestBehavior.AllowGet);
}

```

У методі витягуються всі об'єкти списку, і по ним формується новий анонімний об'єкт - модель комп'ютера. Підсумковий масив нових об'єктів відправляється назад клієнту в форматі JSON. Причому кожен з створюваних анонімних об'єктів повинен мати властивість **label** або властивість **value** (як в нашому випадку), або обидві ці властивості. Властивість **label** застосовується для відображення тексту користувачеві. При виборі користувачем певного елемента в списку об'єктів віджет поміщає **value** виділеного елемента в елемент введення на сторінці. Якщо або властивість **label**, або **value** не вказано, віджет буде використовувати значення однієї зазначеного властивості як для `value`, так і для `label`. Наприкінці представлення визначений скрипт, який викликає функцію `autocomplete` для елемента заповнення:

```

$(function () {
    $('[Data-autocomplete-source]').each(function () {
        var target = $(this);
        target.autocomplete ({source: target.attr ("data-autocomplete-source")});
    });
});

```

Функція **each** викликає для кожного знайденого по селектору елемента функцію, зазначену в параметрі. У цій функції викликається метод плагіна **autocomplete** для кожного елемента. Параметр, який передається в метод **autocomplete**, являє собою об'єкт, в якому визначено одну властивість - властивість **source**. Але можна встановити і більшу кількість властивостей, наприклад, затримку після натискання клавіші або мінімальну кількість символів, необхідне для спрацювання автозаповнення і відправки запиту до джерела даних.

В результаті при зверненні до джерела даних (в даному випадку - до дії AutocompleteSearch контролера Home) плагін передає поточне значення елемента введення в якості параметра term в рядку запиту, а назад клієнту відсилається масив об'єктів в форматі JSON, які на клієнті перетворюються в форму для відображення :

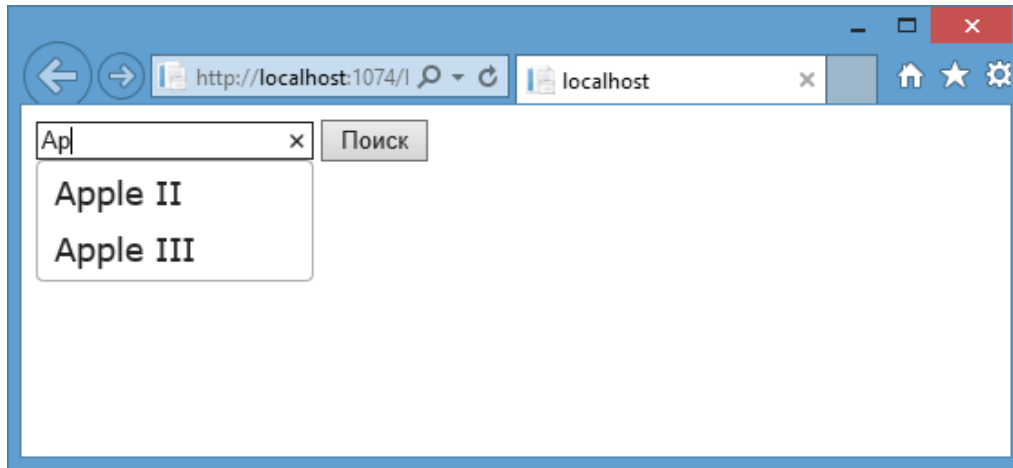


Рисунок 3.113 - Результат виконання програми

3.16.2 Діалогові вікна і CRUD-інтерфейс

У цій темі розглянемо створення на основі віджета dialog з бібліотеки jQuery UI CRUD-інтерфейсу, тобто створення, редагування і видалення через діалогові вікна. Отже, створимо новий проект ASP.NET MVC5. І спочатку додамо в проект через NuGet пакети Entity Framework і jQuery UI, а також налаштуємо підключення до бази даних. І також визначимо клас моделі, з якою ми будемо працювати:

```
public class Computer
{
    public int Id { get; set; }

    [Display(Name = "Модель")]
    public string Model { get; set; }

    [Display(Name = "Год выпуска")]
    public int Year { get; set; }
}
```

Додамо клас контексту даних:

```
public class CompContext : DbContext
{
    public DbSet<Computer> Computers { get; set; }
}
```

Потім змінюємо стандартний контролер HomeController наступним чином:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using CRUDDialogApp.Models;
using System.Data.Entity;

namespace CRUDDialogApp.Controllers
{
    public class HomeController : Controller
    {
        CompContext db = new CompContext();
        public ActionResult Index()
        {
            return View(db.Computers);
        }
        // Перегляд детальних даних про книгу
        public ActionResult Details(int id)
        {
            Computer comp = db.Computers.Find(id);
            if (comp != null)
            {
                return PartialView("Details", comp);
            }
            return View("Index");
        }
        // Додавання
        public ActionResult Create()
        {
            return PartialView("Create");
        }
        [HttpPost]
        [ValidateAntiForgeryToken]
        public ActionResult Create(Computer comp)
        {
            db.Computers.Add(comp);
            db.SaveChanges();
            return RedirectToAction("Index");
        }
        // Редагування
        public ActionResult Edit(int id)
        {
            Computer comp = db.Computers.Find(id);
            if (comp != null)
            {
                return PartialView("Edit", comp);
            }
        }
    }
}
```

```

        return View("Index");
    }
    [HttpPost]
    [ValidateAntiForgeryToken]
    public ActionResult Edit(Computer comp)
    {
        db.Entry(comp).State = EntityState.Modified;
        db.SaveChanges();
        return RedirectToAction("Index");
    }
    // Видалення
    public ActionResult Delete(int id)
    {
        Computer comp = db.Computers.Find(id);
        if (comp != null)
        {
            return PartialView("Delete", comp);
        }
        return View("Index");
    }
    [HttpPost]
    [ValidateAntiForgeryToken]
    [ActionName("Delete")]
    public ActionResult DeleteRecord(int id)
    {
        Computer comp = db.Computers.Find(id);

        if (comp != null)
        {
            db.Computers.Remove(comp);
            db.SaveChanges();
        }
        return RedirectToAction("Index");
    }
}
}

```

Всі методи представляють стандартні дії по додаванню/редагування/видалення об'єктів. Їх особливістю є лише те, що вони повертають часткове представлення. Додамо всі часткові представлення. Часткове представлення Create.cshtml:

```

@model CRUDDialogApp.Models.Computer
<div>
@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()
    <div>
        @Html.LabelFor(model => model.Model)
        <p>@Html.EditorFor(model => model.Model)</p>

```

```

</div>

<div>
    @Html.LabelFor(model => model.Year)
    <p>@Html.EditorFor(model => model.Year)</p>
</div>
<p><input type="submit" value="Додати" /></p>
}
</div>

```

Часткове представлення Edit.cshtml:

```

@model CRUDDialogApp.Models.Computer
<div>
    @using (Html.BeginForm())
    {
        @Html.AntiForgeryToken()
        @Html.HiddenFor(model => model.Id)
        <div>
            @Html.LabelFor(model => model.Model)
            <p>@Html.EditorFor(model => model.Model)</p>
        </div>
        <div>
            @Html.LabelFor(model => model.Year)
            <p>@Html.EditorFor(model => model.Year)</p>
        </div>
        <p><input type="submit" value="Зберегти" /></p>
    }
</div>

```

Часткове представлення Details.cshtml:

```

@model CRUDDialogApp.Models.Computer
<div>
    <h3>Ід: @Html.DisplayFor(m => m.Id)</h3>
    <p>Модель: @Html.DisplayFor(m => m.Model)</p>
    <p>Рік випуску: @Html.DisplayFor(m => m.Year)</p>
</div>

```

І часткове представлення Delete.cshtml:

```

@model CRUDDialogApp.Models.Computer
<h3>Ви дійсно хочете видалити дану модель?</h3>
<div>
    <p>Ід.: @Html.DisplayFor(m => m.Id)</p>
    <p>Модель: @Html.DisplayFor(m => m.Model)</p>
    <p>Год выпуска: @Html.DisplayFor(m => m.Year)</p>
    @using (Html.BeginForm()) {
        @Html.AntiForgeryToken()
    }

```



```

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

```

Всі ці часткові представлення будуть вбудовані в головне представлення Index.cshtml, яке ми змінимо наступним чином:

```

@model IEnumerable<CRUDDialogApp.Models.Computer>
@{
    Layout = null;
}
<html>
<head>
    <title></title>
    <link href="@Url.Content("~/Content/themes/base/jquery-ui.css")" rel="stylesheet"
type="text/css" />
</head>
<body>
    <h2>Список компьютеров</h2>
    <table>
        @foreach (var c in Model)
        {
            <tr><td>@c.Model</td><td>
                @Html.ActionLink("Дивитись", "Details", "Home", new { id = c.Id },
                    new { @class = "viewDialog", data_dialog_title = "Опис моделі" }) |
                @Html.ActionLink("Змінити", "Edit", "Home", new { id = c.Id },
                    new { @class = "viewDialog", data_dialog_title = "Редагування моделі" }) |
                @Html.ActionLink("Видалити", "Delete", "Home", new { id = c.Id },
                    new { @class = "viewDialog", data_dialog_title = "Видалення моделі" }) </td></tr>
            }
        </table>
        <p> @Html.ActionLink("Додати", "Create", "Home", new { }, new { @class = "viewDialog",
data_dialog_title = "Створення моделі" })</p>

        <script src="@Url.Content("~/Scripts/jquery-1.10.2.min.js")" type="text/javascript"></script>
        <script src="@Url.Content("~/Scripts/jquery-ui-1.10.4.min.js")" type="text/javascript"></script>
        <script>
            $(document).ready(function () {

                $.ajaxSetup({ cache: false });

                $(".viewDialog").on("click", function (e) {
                    e.preventDefault();

                    $("<div></div>")
                        .addClass("dialog")
                        .appendTo("body")
                        .dialog({
                            title: $(this).attr("data-dialog-title"),
                            close: function () { $(this).remove() },

```

```

        modal: true
    })
    .load(this.href);
});
$(".close").on("click", function (e) {
    e.preventDefault();
    $(this).closest(".dialog").dialog("close");
});
});
</script>
</body>
</html>

```

Всі посилання мають атрибут класу `class = "view Dialog"`, за допомогою якого на ці посилання прикріплюється обробник натискання `$(".ViewDialog").On ("click", function (e)` А в обробнику відбувається створення діалогового вікна, яке буде містити потрібне часткове представлення.

3.16.3 Валідація в діалогових вікнах

У попередній темі було розглянуто створення діалогових вікон для додавання і редагування моделей. Але не було порушено питання валідації моделей при їх створенні в діалогових вікон. Розглянемо, як ми можемо додати валідацію. Візьмемо модель `Computer` з минулого теми і додамо їй кілька атрибутів валідації:

```

using System.ComponentModel.DataAnnotations;

public class Computer
{
    public int Id { get; set; }

    [Display(Name = "Модель")]
    [Required]
    [MaxLength(20, ErrorMessage = "Перевищена допустима довжина рядка")]
    public string Model { get; set; }

    [Display(Name = "Рік випуску")]
    [Required]
    [Range(1970, 2014, ErrorMessage = "Недопустимий рік")]
    public int Year { get; set; }
}

```

Тепер змінимо код контролера для прийняття і виведення моделей:

```

public class HomeController : Controller
{
    static List<Computer> comps = new List<Computer>();
    public ActionResult Index()
    {

```

```

        return View(comps);
    }

    // Додавання
    public ActionResult Create()
    {
        return PartialView("Create");
    }
    [HttpPost]
    [ValidateAntiForgeryToken]
    public ActionResult Create(Computer comp)
    {
        if (ModelState.IsValid)
        {
            comps.Add(comp);
            return PartialView("Success");
        }
        return PartialView(comp);
    }
}

```

Для простоти замість БД використовується статичний список, і є всього лише два дії - для виведення списку і для створення нової моделі. При створенні моделі в залежності від результату повертається одне з часткових представлень. По-перше, додамо в проект бібліотеку jQuery-UI і додамо представлення. Основне представлення Index.cshtml для виведення списку:

```
@model IEnumerable<DialogValidationApp.Models.Computer>
```

```

@{
    Layout = null;
}
<html>
<head>
    <title></title>
    <link href="@Url.Content("~/Content/Site.css")" rel="stylesheet" type="text/css" />
    <!--бібліотека jQuery-UI-->
    <link href="@Url.Content("~/Content/themes/base/all.css")" rel="stylesheet" type="text/css" />
</head>
<body>
    <h2>Список комп'ютерів</h2>
    <table>
        @foreach (var c in Model)
        {
            <tr>
                <td>@c.Model</td>
                <td>@c.Year</td>
            </tr>
        }
    </table>

```

```

<p> @Html.ActionLink("Додати", "Create", "Home", new { }, new { @class = "viewDialog",
data_dialog_title = "Створення моделі" })</p>

<script src='@Url.Content("~/Scripts/jquery-1.10.2.min.js")' type="text/javascript"></script>
<script src="//code.jquery.com/ui/1.11.0/jquery-ui.js"></script>
<script>
$(document).ready(function () {

    $.ajaxSetup({ cache: false });

    $(".viewDialog").on("click", function (e) {
        e.preventDefault();

        $("

На відміну від механізму створення діалогового вікна в минулій темі, тут до діалогового вікна додається кнопка, і цій кнопці призначається обробник натискання, в якому спрацьовує аякс-запит за допомогою функції $.ajax(). Функція серіалізує дані форми і відправляє за адресою, вказаною в параметрі url.



465


```

При успішній відправці за допомогою функції `success: function (result) {...` ми можемо обробити відповідь від сервера. В даному випадку відповідь сервера просто додається до блоку діалогового вікна: `$("# dialogContent").Html(result)`. Тепер додамо часткове представлення `Create.cshtml`:

```
@model DialogValidationApp.Models.Computer

@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()
    <div>
        <h4>Нова модель</h4>

        @Html.ValidationSummary(true)
        <div>
            @Html.LabelFor(model => model.Model)
            <div>
                @Html.EditorFor(model => model.Model)
                @Html.ValidationMessageFor(model => model.Model, "")
            </div>
        </div>

        <div>
            @Html.LabelFor(model => model.Year)
            <div>
                @Html.EditorFor(model => model.Year)
                @Html.ValidationMessageFor(model => model.Year, "")
            </div>
        </div>
    </div>
}
```

Тут стандартна форма з хелперами валідації. Єдина відмінність від стандартних форм - ВІДСУТНІСТЬ кнопки відправки, так як ми її створюємо в коді `javascript` в представленні `Index.cshtml`. Додамо часткове представлення `Success.cshtml`, яке буде генеруватися при успішному додаванні моделі:

```
<h2>Модель додана</h2>
<script>
    $(document).ready(function () {
        setTimeout(function() {
            window.location.href = "@Url.Action("Index", "Home")";
        }, 2000);
    })
</script>
```

Тут виводиться повідомлення про додавання, і потім запускається скрипт, який перенаправляє знову ж на головну сторінку, щоб додані дані відобразилися на сторінці. В якості альтернативи можна було б зробити завантаження даних

таблиці моделей через аjax, щоб не перевантажувати всю сторінку. Отже, запусимо програму і спробуємо ввести такі дані, які не пройдуть валідацію – ми одержимо повідомлення про помилку. Якщо ж модель пройде валідацію, то ми отримаємо повідомлення про її додавання, і сторінка Index.cshtml буде перезавантажена.

3.16.4 JqGrid в ASP.NET MVC

Для створення простих таблиць, які просто виводять дані, на стороні клієнта досить використовувати стандартний елемент html <table>. Однак створення більш складних по функціоналу таблиць даного елемента вже недостатньо, і доводиться вдаватися до більш комплексних рішень.

У ASP.NET MVC, як і для інших серверних технологій, є безліч рішень таблиць, і одним з таких рішень є ґрид JQGrid.

Отже, використовуємо JQGrid в проєкті ASP.NET MVC. Для цього створимо новий проєкт ASP.NET MVC. Назвемо свій проєкт JQGridApp. І відразу додамо в нього всі необхідні бібліотеки. Всі потрібні файли можна знайти на сайті безпосереднього розробника:

http://www.trirand.com/blog/?page_id=6.

Іншим способом - можна виконати установку через NuGet:

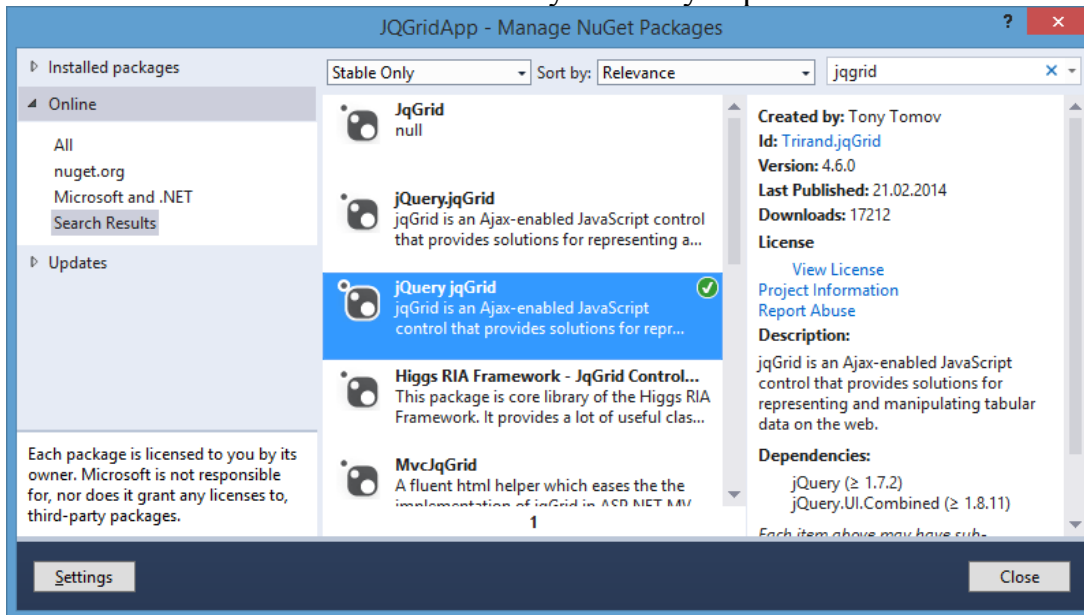


Рисунок 3.114 - Установка JQGrid за допомогою NuGet

Як видно на скріншоті, пакет JqGrid має залежність від бібліотек jQuery і jQuery-UI, тому при установці пакету ці бібліотеки в разі відсутності також будуть встановлені.

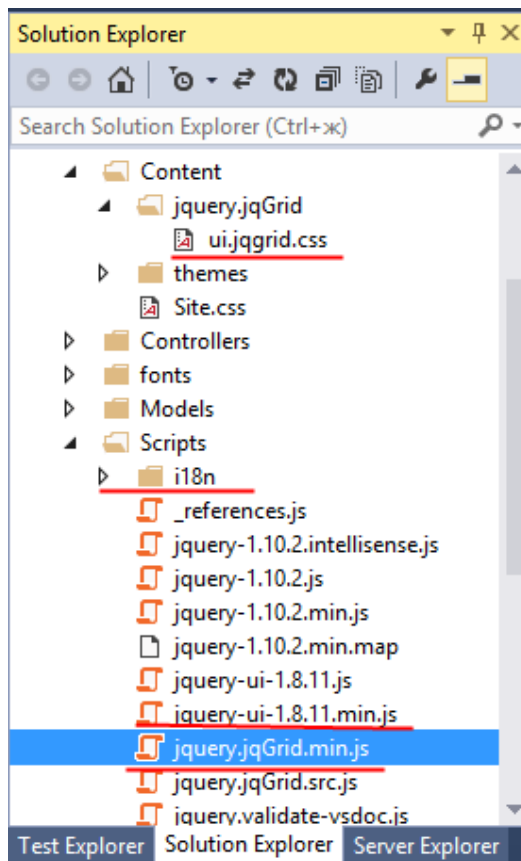


Рисунок 3.115 - Перевірка наявності скриптів

Тепер в папку Models додамо модель, об'єкти якої будуть вводитися в сітку. Це буде модель Book:

```
public class Book
{
    public int Id { get;set;}
    public string Name { get; set; }
    public string Author { get; set; }
    public int Year { get; set; }
    public int Price { get; set; }
}
```

Змінимо стандартний контролер HomeController наступним чином:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using JQGridApp.Models;
using Newtonsoft.Json;

namespace JQGridApp.Controllers
{
    public class HomeController : Controller
    {
```

```

static List<Book> books = new List<Book>();
static HomeController()
{
    books.Add(new Book { Id = 1, Name = "Кобзар", Author = "Т. Шевченко", Year = 1863, Price
= 220 });
    books.Add(new Book { Id = 2, Name = "Коли ще звірі говорили ", Author = "І. Франко", Year
= 1896, Price = 195 });
    books.Add(new Book { Id = 3, Name = "Природа", Author = "О. Кобилянська", Year = 1888,
Price = 158 });
    books.Add(new Book { Id = 4, Name = "Устим Кармалюк", Author = "В. Кучер", Year = 1983,
Price = 210 });
}

public ActionResult Index()
{
    return View();
}

public string GetData()
{
    return JsonConvert.SerializeObject(books);
}
}
}

```

Для спрощення прикладу база даних не використовується, а список об'єктів створюється безпосередньо в контролері. У методі `GetData` цей список серіалізується в формат `json`.

Змінимо представлення `Index.html`:

```

@{
    Layout = null;
}

<html>
<head>
    <title>jqGrid</title>
    <script src="~/Scripts/jquery-1.10.2.min.js" type="text/javascript"></script>
    <link href="~/Content/themes/base/jquery.ui.all.css" rel="stylesheet" type="text/css" />
    <link href="~/Content/jquery.jqGrid/ui.jqgrid.css" rel="stylesheet" type="text/css" />
    <script src="~/Scripts/jquery-ui-1.8.11.min.js" type="text/javascript"></script>
    <script src="~/Scripts/jquery.jqGrid.min.js" type="text/javascript"></script>
</head>
<body>
<h2>jQGrid</h2>
<table id="jqg"></table>
<script type="text/javascript">
$(document).ready(function () {
    $("#jqg").jqGrid({
        url: '@Url.Action("GetData")',
        datatype: "json",

```



```

colNames: ['Id', 'Назва', 'Автор', 'Рік', 'Ціна'],
colModel: [
{ name: 'Id', index: 'Id', width: 30, stype: 'text' },
{ name: 'Name', index: 'Name', width: 150, sortable: true},
{ name: 'Author', index: 'Author', width: 150, sortable: true },
{ name: 'Year', index: 'Year', width: 100, sortable: false },
{ name: 'Price', index: 'Price', width: 80, align: "right", sortable: true }
],
rowNum: 5, // число рядків що відображається
loadonce: true, // завантаження тільки один раз
sortname: 'Id', // сортування за замовчуванням за столбчиком Id
sortorder: "desc", // порядок сортування
caption: "Список Книг"
});
});
</script>
</body>
</html>

```

Розберемо код. Спочатку йде підключення раніше встановлених скриптів і файлів css. Далі в функції jQuery створюється сітка на базі елемента `<table id = "jqg"> </ table>`. При завантаженні сторінки сітка буде завантажувати дані, звертаючись до методу `GetData` і отримуючи від нього дані в форматі json. Параметр `colNames` встановлює заголовки стовпців, а параметр `colModel` модель стовпців. Кожен стовпець визначається в такому вигляді `{name: 'Id', index: 'Id', width: 30}`, де опції `name` і `index` співпадають з назвами властивостей моделі, до яких будуть прив'язані дані стовпці. І також можна визначити додаткові опції, як `width` - ширину, `sortable` - включає сортування по стовпцю, `align` - вирівнювання тексту і ряд інших. Повний набір параметрів, як для ґріда в цілому, так і для окремих стовпців, можна знайти в документації.

3.16.5 Посторінковий вивід в JqGrid

Змінимо представлення з попередньої теми наступним чином:

```

@{
    Layout = null;
}

<html>
<head>
    <title>jqGrid</title>
    <script src="~/Scripts/jquery-1.10.2.min.js" type="text/javascript"></script>
    <link href="~/Content/themes/base/jquery.ui.all.css" rel="stylesheet" type="text/css" />
    <link href="~/Content/jquery.jqGrid/ui.jqgrid.css" rel="stylesheet" type="text/css" />
    <script src="~/Scripts/jquery-ui-1.8.11.min.js" type="text/javascript"></script>
    <script src="~/Scripts/jquery.jqGrid.min.js" type="text/javascript"></script>
    <script src="~/Scripts/i18n/grid.locale-ru.js" type="text/javascript"></script>
    <style>
        .ui-jqgrid .ui-pg-input {height:20px;}
    </style>

```

```

</style>
</head>
<body>
<h2>jQGrid</h2>
<table id="jqg"></table>
<div id="jpager"></div>
<script type="text/javascript">
$(document).ready(function () {
    $("#jqg").jqGrid({
        url: '@Url.Action("GetData")',
        datatype: "json",
        colNames: ['Id', 'Назва', 'Автор', 'Рік', 'Ціна'],
        colModel: [
            { name: 'Id', index: 'Id', width: 30, stype: 'text' },
            { name: 'Name', index: 'Name', width: 150, sortable: true },
            { name: 'Author', index: 'Author', width: 150, sortable: true },
            { name: 'Year', index: 'Year', width: 100, sortable: false },
            { name: 'Price', index: 'Price', width: 80, align: "right", sortable: true }
        ],
        rowNum: 5, // число рядків які відображаються
        rowList: [10, 20, 30],
        pager: '#jpager',
        loadonce: true, // завантаження тільки один раз
        sortname: 'Id', // сортування за замовчуванням за столбчиком Id
        sortorder: "desc", // порядок сортування
        caption: "Список Книг"
    });
});
</script>
</body>
</html>

```

На відміну від попереднього прикладу тут доданий ще один скрипт - скрипт локалізації `grid.locale-ru.js`, який встановився в проект разом з іншими файлами JqGrid в каталог `Scripts/i18n`. При необхідності можна вибрати будь-який інший файл локалізації. Зокрема, є також локалізації для білоруської та української мов. Крім того, можна самим створити на основі наявної власну локалізацію. Також на сторінку доданий новий елемент, який буде містити функціонал пагінації: `<div id = "jpager"> </ div>`

А у визначенні сітки вказується, що саме в цьому елементі буде застосовуватися пагінація для сітки: `pager: '#jpager'`. У підсумку ми отримаємо сітку з елементами посторінкової навігації:

3.16.6 Пошук по таблиці в JqGrid

Тепер додамо пошук по таблиці. Для цього змінимо представлення з попередньої теми наступним чином:

```
@{
```

```

Layout = null;
}

<html>
<head>
  <title>jqGrid</title>
  <script src="../../Scripts/jquery-1.10.2.min.js" type="text/javascript"></script>
  <link href="../../Content/themes/base/jquery.ui.all.css" rel="stylesheet" type="text/css" />
  <link href="../../Content/jquery.jqGrid/ui.jqgrid.css" rel="stylesheet" type="text/css" />
  <script src="../../Scripts/jquery.jqGrid.min.js" type="text/javascript"></script>
  <script src="../../Scripts/i18n/grid.locale-ru.js" type="text/javascript"></script>
  <style>
    body {background-color:#f0f0ffa;}
    .ui-jqgrid .ui-pg-input {height:20px;}
    .ui-jqgrid .ui-jqgrid-pager {height:50px;}
  </style>
</head>
<body>
<h2>jQGrid</h2>
<table id="jqg"></table>
<div id="jpager"></div>
<script type="text/javascript">
$(document).ready(function () {
  $("#jqg").jqGrid({
    url: '@Url.Action("GetData")',
    datatype: "json",
    colNames: ['Id', 'Назва', 'Автор', 'Рік', 'Ціна'],
    colModel: [
      { name: 'Id', index: 'Id', width: 30, stype: 'text' },
      { name: 'Name', index: 'Name', width: 150, sortable: true },
      { name: 'Author', index: 'Author', width: 150, sortable: true },
      { name: 'Year', index: 'Year', width: 100, sortable: false },
      { name: 'Price', index: 'Price', width: 80, align: "right", sortable: true }
    ],
    rowNum: 10,
    rowList: [10, 20, 30],
    pager: '#jpager',
    loadonce: true,
    sortname: 'Id',
    sortorder: "desc", // порядок сортування,
    caption: "Список Книг"
  });
  $("#jqg").jqGrid('navGrid', '#jpager', {

    add: false,
    del: false,
    edit: false,
    search: true,
    searchtext: "Пошук",
    refresh: true,
    refreshtext: "Оновити"
  });
});

```

```

});
});
</script>
</body>
</html>

```

Ми можемо управляти візуальними властивостями елементів сітки, встановлюючи стилі `.ui-jqgrid .ui-jqgrid-pager {height: 50px;}`. У деяких випадках це бажано робити, так як вбудовані стилі можуть виявитися не зовсім оптимальними. Елемент `.ui-jqgrid-pager` якраз представляє нижню панель з кнопками і пагінацією.

Для управління нижньою панеллю використовується функція `$("#jqg").JqGrid ('navGrid', '#jpager',)`. В ній задаються параметри нижньої панелі. Так, за замовчуванням на панель додаються кнопки для додавання, видалення і т.д. Але поки нам вони не потрібні, і тому для них встановлюється значення `false`: `add: false`.

Наразі нам потрібні тільки кнопки пошуку і оновлення, тому для них задається значення `true`, і опціонально ми можемо поставити і текст (значення тексту також можна залишити порожнім: `refreshText: ""`)

Оновлення в даному випадку не звертається до сервера, а оновлює дані в сітці, які могли зазнати сортування, і потім при натисканні на кнопку пошуку відобразиться вікно, в якому за різними критеріями ми зможемо знайти потрібні об'єкти.

Вікно пошуку має набір полів з деякими значеннями, які при бажанні ми можемо змінити. Для цього нам треба перейти до файлу `grid.locale-ru.js` і знайти параметр `search`, який і задає всі опції вікна пошуку.

```

search : {
  caption: "Пошук.",
  Find: "Знайти",
  Reset: "Скид",
  odata: [{ oper:'eq', text:"рівно"},{ oper:'ne', text:"не рівне"},....],
  groupOps: [ { op: "AND", text: "все" }, { op: "OR", text: "будь-який" } ],
  operandTitle : "Click to select search operation.",
  resetTitle : "Reset Search Value"
},

```

3.16.7 CRUD-операції в JqGrid

CRUD операції передбачають можливість читання окремих записів, додавання, редагування, а також видалення. Додамо подібну функціональність і в JqGrid. Для цього змінимо представлення наступним чином:

```

@{
  Layout = null;
}
<html>
<head>

```

```

<title>jqGrid</title>
<script src="../../Scripts/jquery-1.10.2.min.js" type="text/javascript"></script>
<link href="../../Content/themes/base/jquery.ui.all.css" rel="stylesheet" type="text/css" />
<link href="../../Content/jquery.jqGrid/ui.jqgrid.css" rel="stylesheet" type="text/css" />
<script src="../../Scripts/jquery.jqGrid.min.js" type="text/javascript"></script>
<script src="../../Scripts/i18n/grid.locale-ru.js" type="text/javascript"></script>
<style>
    .ui-jqgrid .ui-pg-input {height:20px;}
    .ui-jqgrid .ui-jqgrid-pager {height:40px;}
    .ui-jqgrid .ui-jqgrid-pager .ui-pg-div {font-size:14px;}
</style>
</head>
<body>
<h2>jQGrid</h2>
<table id="jqg"></table>
<div id="jpager"></div>
<script type="text/javascript">
$(document).ready(function () {
    $("#jqg").jqGrid({
        url: '@Url.Action("GetData")',
        datatype: "json",
        colNames: ['Id', 'Назва', 'Автор', 'Рік', 'Ціна'],
        colModel: [
            { name: 'Id', index: 'Id', width: 30, stype: 'text', key:true },
            { name: 'Name', index: 'Name', width: 150, editable: true, edittype: 'text', sortable: true },
            { name: 'Author', index: 'Author', width: 150, editable: true, edittype: 'text', sortable: true },
            { name: 'Year', index: 'Year', width: 100, editable: true, edittype: 'text', sortable: false },
            { name: 'Price', index: 'Price', width: 80, align: "right", editable: true, edittype: 'text', sortable:
true}
        ],
        rowNum: 10,
        rowList: [10, 20, 30],
        width:800,
        pager: '#jpager',
        loadonce:true,
        sortname: 'Id',
        sortorder: "desc",
        caption: "Список Книг"
    });
    $("#jqg").jqGrid('navGrid', '#jpager', {

        search: true,
        searchtext: "Пошук",
        refresh: false,
        add: true, // додавання
        del: true, // видалення
        edit: true, // редагування
        view: true, //перегляд запису
        viewtext: "Дивитись",
        viewtitle: "Вибраний запис",
        addtext: "Додати",
    });
});

```

```

        edittext: "Змінити",
        deltext: "Видалити"
    },
    update("edit"), // оновлення
    update("add"), // додавання
    update("del") // видалення
);
function update(act) {
    return {
        closeAfterAdd: true, // закрити після оновлення
        height: 250,
        width: 400,
        closeAfterEdit: true, // закрити після редагування
        reloadAfterSubmit: true, // оновлення
        drag: true,
        onclickSubmit: function (params) {
            var list = $("#jqg");
            var selectedRow = list.getGridParam("selrow");
            rowData = list.getRowData(selectedRow);
            if (act === "add")
                params.url = '@Url.Action("Create")';
            else if (act === "del")
                params.url = '@Url.Action("Delete")';
            else if (act === "edit")
                params.url = '@Url.Action("Edit")';
        },
        afterSubmit: function (response, postdata) {
            // оновлення сітки
            $(this).jqGrid('setGridParam', { datatype: 'json' }).trigger('reloadGrid')
            return [true, "", 0]
        }
    };
};
});
</script>
</body>
</html>

```

На відміну від представлення минулої теми у визначенні стовпців додана наступна опція: `editable: true`. Вона дозволить при генерації форм додавання/редагування створити для даного стовпця поле. Оскільки властивість `Id` нам не треба міняти або встановлювати вручну, то для нього ця опція не вказується.

Також тут підключена функціональність перегляду окремих записів, додавання, редагування та видалення: `add:true`, `del:true`, `edit:true`, `view:true`, крім того, також встановлюється текст міток. Знову ж можна було б і не встановлювати текст, тоді б були просто іконки для даних функціональностей. Далі визначаються функції для кожної дії, крім перегляду. Так як реалізація всіх трьох дій схожа, то весь код визначено в одній функції `update`. При цьому виклик

функцій йде в певному порядку: спочатку редагування, потім додавання і в кінці видалення. Як параметр функція `update` приймає строкове опис дії, то в залежності від його значення в обробнику відправки форми `onclickSubmit` звертається до того чи іншого методу контролера:

```
onclickSubmit: function (params) {  
    var list = $("#jqg");  
    var selectedRow = list.getGridParam("selrow");  
    rowData = list.getRowData(selectedRow);  
    if (act === "add")  
        params.url = '@Url.Action("Create")';  
    else if (act === "del")  
        params.url = '@Url.Action("Delete")';  
    else if (act === "edit")  
        params.url = '@Url.Action("Edit")';  
},
```

В даному випадку змінна `params.url` зберігає адресу, за якою форма з даними відправлятиметься. При цьому нам не треба явним чином створювати будь-яку форму, плагін `JqGrid` все зробить за нас. Також визначена функція `afterSubmit`, що викликається після отримання відповіді від сервера після відправки форми. Вона виконує оновлення сітки:

```
afterSubmit: function (response, postdata) {  
    $(this).jqGrid('setGridParam', { datatype: 'json' }).trigger('reloadGrid')  
    return [true, "", 0]  
}
```

Вона повертає об'єкт у вигляді масиву, де перший елемент показує, що запит виконався вдало, і тому `true`. В разі помилки можна буде вказати `false`. Другий параметр вказує на повідомлення від сервера. А третій параметр повинен містити `id` доданої, оновленого або віддаленого запису. Для простоти прикладу ми зробили повідомлення порожнім, а замість `id` вказали число `0`.

Тепер змінимо код контролера, додавши в нього нові методи, до яких буде йти звернення з представлення. Отже, додамо наступні методи:

```
[HttpPost]  
public void Edit(Book book)  
{  
    // дії за потреби редагування  
}
```

```
[HttpPost]  
public void Create(Book book)  
{  
    // дії за потреби додавання  
}
```

```
[HttpPost]  
public void Delete(int id)  
{
```

```
// дії за потреби видалення
}
```

Тепер запустивши додаток, ми можемо подивитися окремий запис, додати нову або відредагувати стару (форма редагування аналогічна формі додавання). Таким чином, ми розглянули деякі аспекти використання плагіна JqGrid для створення таблиць з розширеною функціональністю.

3.16.8 Посторінковий вивід

Пагінація або посторінковий вивід дозволяє розбити набір об'єктів на кілька сторінок і встановити посилання на спрощеного доступу до конкретних сторінок. У ASP.NET MVC пагінацію зробити дуже легко. Можна, звичайно, самим спробувати визначити логіку посторінкового виведення. Однак ми можемо спростити собі роботу, скориставшись наявними плагінами. Один з таких плагінів являє пакет PagedList.Mvc.

Отже, додамо цей пакет в проект ASP.NET MVC5:

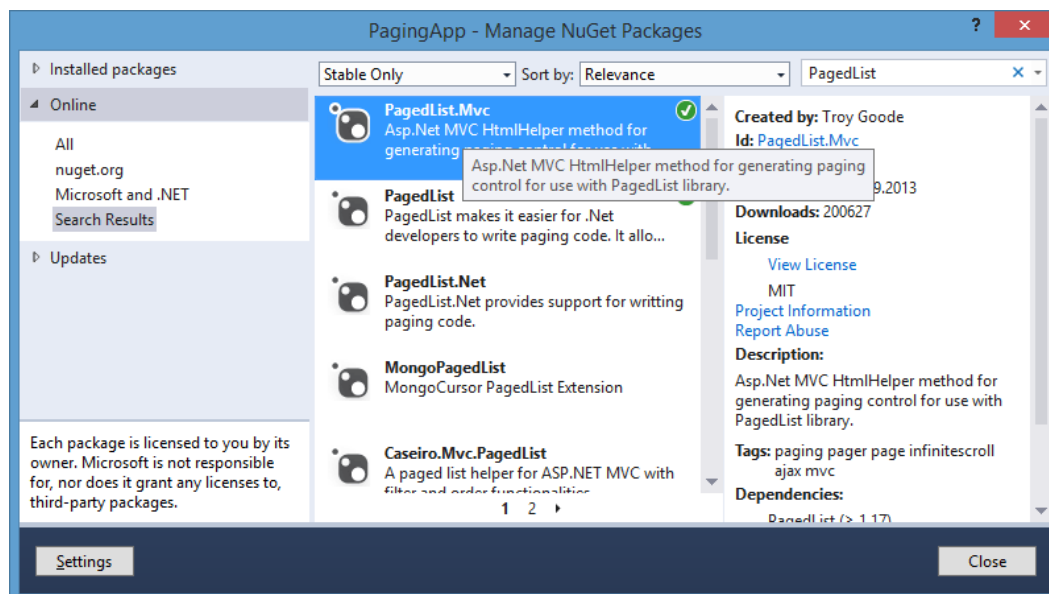


Рисунок 3.116 - Додавання пакету посторінкового виводу

Після установки в проект в вузол References будуть додані дві нові бібліотеки PagedList і PagedList.Mvc. Крім того, в вузол Content буде додано файл стилів PagedList.css, який буде використовуватися при створенні посторінкового виведення.

Для посторінкового виведення будуть виводитися об'єкти моделі Phone:

```
public class Phone
{
    public int Id { get; set; }
    public string Name { get; set; }
}
```

Тепер передамо список моделей для посторінкового виведення з контролера в представлення:


```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using PagingApp.Models;
using PagedList.Mvc;
using PagedList;

namespace PagingApp.Controllers
{
    public class HomeController : Controller
    {
        List<Phone> phones;
        public HomeController()
        {
            phones = new List<Phone>();
            phones.Add(new Phone { Id = 1, Name = "Samsung Galaxi" });
            phones.Add(new Phone { Id = 2, Name = "Samsung Galaxi II" });
            phones.Add(new Phone { Id = 3, Name = "Samsung Galaxi II" });
            phones.Add(new Phone { Id = 4, Name = "Samsung ACE" });
            phones.Add(new Phone { Id = 5, Name = "Samsung ACE II" });
            phones.Add(new Phone { Id = 6, Name = "HTC One S" });
            phones.Add(new Phone { Id = 7, Name = "HTC One X" });
            phones.Add(new Phone { Id = 8, Name = "Nokia N9" });
        }
        public ActionResult Index(int? page)
        {
            int pageSize = 3;
            int pageNumber = (page ?? 1);
            return View(phones.ToPagedList(pageNumber, pageSize));
        }
    }
}

```

Для спрощення прикладу ми створюємо звичайний список моделей в конструкторі контролера.

Щоб скористатися функціональністю пагінації, треба підключити до класу простору імен PagedList.Mvc і PagedList.

В представлення при цьому передається посторінковий список у вигляді об'єкта IPagedList<T>, який створюється за допомогою методу розширення ToPagedList(). Цей метод приймає два параметри: номер сторінки і число об'єктів на сторінку. Тут номер сторінки передається у вигляді параметра методу, а якщо його немає (тобто він має значення null), то використовується перша сторінка. А в якості кількості об'єктів на сторінку використовується число 3.

Посторінковий список представляє звичайний список з тими ж методами, тільки додаються кілька додаткових властивостей, наприклад, PageNumber (номер сторінки), PageCount (кількість сторінок), PageSize (максимальна кількість об'єктів на сторінці) і ряд інших.

А представлення Index.cshtml буде виглядати наступним чином:

```

@model PagedList.IPagedList<PagingApp.Models.Phone>
@using PagedList.Mvc;
<link href="~/Content/PagedList.css" rel="stylesheet" type="text/css" />
@{
    ViewBag.Title = "Index";
}

<h2>Список телефонів</h2>
<table class="table">
    <tr>
        <th>Назва</th>
    </tr>

    @foreach (var item in Model) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Name)
            </td>
        </tr>
    }
</table>
<br />
Сторінка @(Model.PageCount < Model.PageNumber ? 0 : Model.PageNumber) з
@Model.PageCount
@Html.PagedListPager(Model, page => Url.Action("Index", new { page }))

```

Для стилізації посилань посторінкового виводу на початку представлення підключається файл стилів PagedList.css. Для самого посторінкового виводу і створення посилань використовується хелпер Html.PagedListPager().

Контрольні питання.

1. Призначення бібліотеки jQuery UI .
2. Особливості валідації в діалогових вікнах jQuery UI.
3. Посторінковий вивід в JqGrid.
4. Особливості посторінкового виводу.

4 Засоби рефакторингу та інші технологічні моменти.

4.1 ReSharper

ReSharper (R#) — додаток (add-on), розроблений компанією JetBrains для збільшення продуктивності роботи та автоматизації рефакторингу в середовищі Microsoft Visual Studio (підтримуються версії: 2003, 2005, 2008, 2010, 2012, 2013, 2015).

Здійснює миттєвий статичний аналіз коду (без потреби компіляції), передбачає додаткові засоби автозаповнення, навігації, пошуку, виділення

синтаксису, форматування, оптимізації та генерації коду, надає близько 40 автоматизованих рефакторингів, спрощує модульне тестування в середовищах MSTest та NUnit.

Остання версія ReSharper 10 була випущена в 2015. Забезпечено підтримку мов C#, C++, VB, JavaScript, CSS, HTML, ASP.NET, ASP.NET MVC, Razor, XML, XAML, WinRT, сценаріїв MSBuild та NAnt.

ReSharper - поширений інструмент для підвищення продуктивності роботи, що дозволяє істотно збільшити функціональність Microsoft Visual Studio. Тисячі .NET розробників по всьому світу використовують ReSharper для перевірки коду, здійснення автоматичного рефакторингу та одержання допомоги в написанні якісного програмного коду. В арсеналі ReSharper передбачено понад 1300 перевірок коду. Програма підкреслює знайдені помилки просто у вікні редактора Visual Studio, і додатково відображає їх за допомогою відповідних маркерів. Індикатор у верхній частині панелі маркерів змінює свій колір та значок, якщо досліджуваний файл містить помилки або проблеми. Помилки в програмі ReSharper відстежуються в режимі реального часу та виділяються червоним шрифтом або хвилястим підкресленням. Якщо навести вказівник миші на помилку, її опис буде виведено за допомогою відповідної підказки.

```
if(FRead == null)
    throw new ArgumentNullException("FRead");
new TextReader(stream).WithDispose(FRead);
```

Cannot create an instance of the abstract class 'System.IO.TextReader'

Крім того, ReSharper здійснює детектування ділянок коду, що не стають на заваді компіляції, але можуть суттєво знижувати якість розробленого продукту та перетворювати його на неефективний. Наприклад, ReSharper показує інформацію про посилання на простори імен та складання що не використовуються, неправильний формат рядків, оголошені, але не використані змінні тощо. Відповідні конструкції позначаються сірим кольором.

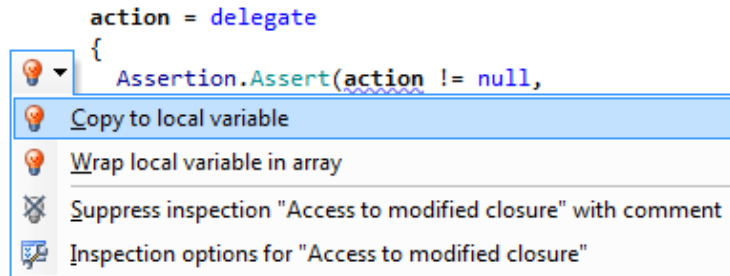
```
else
{
    ctor = info.Type.GetConstructor(Type.EmptyTypes);
    externalizable = (IXmlExternalizable) ctor.Invoke(Type.EmptyTypes);
}
```

Code is heuristically unreachable

ReSharper також може здійснювати рекомендації по оптимізації коду. В цьому випадку ReSharper здійснює більш глибокий аналіз коду. За даними з офіційного сайту, ReSharper лише привертає увагу розробника до тих конструкцій, які потенційно можуть бути покращені. Наприклад, ReSharper може запропонувати перетворити цикл до LINQ-виразу, використати ключове слово "var" та ін. Відповідні рекомендації виділяються зеленим кольором.

Найменш помітними повідомленнями ReSharper є «поради». На відміну від помилок, попереджень та рекомендацій, вони лише показують можливі шляхи збільшення якості програмних конструкцій.

ReSharper надає можливість швидкого усунення більшості помилок, та дозволяє розв'язувати всі можливі проблеми. Швидкі виправлення представлено червоними (для помилок) та жовтими (в інших випадках) лампочками, які з'являються автоматично ліворуч від обраного рядка коду. Натиснувши на символ лампочки або ввівши комбінацію Alt + Enter, обираємо необхідну позицію зі списку і проблему буде вирішено в найбільш зручному вигляді:



Набір функцій ReSharper для рефакторингу значно перевищує за кількістю та зручністю використання вбудовані засоби середовища Visual Studio. У програмі забезпечено можливість відповідного покращення коду в мові C#, переважна більшість рефакторингів також доступні в VB.NET, деякі в JavaScript, XAML та інших підтримуваних мовах. Для здійснення рефакторингу необхідно лише навести курсор на відповідний блок коду.

Механізм зберігання налаштувань ReSharper дозволяє синхронізувати їх усередині команди, при необхідності комбінуючи персональні та командні настройки. Цей механізм особливо корисний для налаштувань, що стосуються стандартів оформлення коду, таких як іменування символів і форматування. Функція очищення коду дозволить швидко переформатувати код після зміни налаштувань, вставити або оновити заголовки файлів, а також застосувати ряд інших дій, щоб привести існуючий код у відповідність з внутрішніми стандартами.

ReSharper значно спрощує роботу з багатомовними .NET додатками, підтримуючи як .resx файли, так і використання ресурсів у коді C#, VB.NET, ASP.NET і XAML. Ви зможете легко переносити рядки в ресурси і перевизначати їх значення для інших культур, швидко знаходити використання ресурсу в коді, перейменовувати і безпечно видаляти ресурси.

ReSharper пропонує нові можливості для навігації і пошуку у вихідному коді і збірках, включаючи збірки платформи .NET Framework. Для пошуку будь-якого файлу, типу, методу, змінної вам буде потрібно натиснути всього кілька клавіш. Від будь-якого символу в коді ви зможете миттєво переходити до базових і похідних символів, для використання, реалізації та перевизначення. Для будь-якого типу ReSharper побудує візуальну ієрархію спадкування з можливістю навігації до будь-якого вузла. Якщо шуканий символ знаходиться у зовнішній збірці, ReSharper звернеться до сервера вихідного коду або декомпілює збірку.

У ReSharper доступні декілька різних функцій для роботи з повторюваними конструкціями коду. Генерація коду для типу допоможе швидко

додати конструктори, перевизначення методів, властивості, делегати та інші члени. Вбудовані шаблони покривають більшість типових конструкцій, таких як, наприклад, try ... catch і if ... else, а спеціальний редактор шаблонів допоможе вам створити будь-які інші необхідні шаблони. Крім того, ReSharper дозволяє використовувати методи, змінні і навіть класи, що не були оголошені: всі необхідні оголошення будуть згенеровані автоматично.

Набір рефакторингов, пропонований ReSharper, перевершує аналогічну функціональність Visual Studio як за кількістю функцій, так і за зручністю їх використання. На додаток до рефакторингів, які, в основному, орієнтовані на масштабні трансформації, ReSharper включає сотні контекстних дій для невеликих перетворень.

ReSharper розпізнає тести NUnit, MSTest, QUnit і Jasmine, і дозволяє запускати або налагоджувати їх прямо з редактора або зі спеціалізованого вікна Unit Test Sessions. Тестові сесії можуть комбінуватися з довільної кількості тестів і запускатися паралельно. Якщо ви віддасте перевагу тестові фреймворки xUnit.net або MSpec, вони також підтримуються в ReSharper через розширення.

Контрольні питання.

1. Призначення ReSharper .
2. Основні можливості ReSharper.

4.2 Тестування програмного забезпечення

4.2.1 Вступ в тестування. Test-Driven Development:

Одна з переваг розробки на платформі - ASP.NET MVC надає широкі можливості по тестуванню веб-додатків. Можна самим виконувати тестування тих чи інших моментів вручну, а можна використовувати спеціальні невеликі програми, які називаються юніт-тести.

Юніт-тести дозволяють швидко і автоматично протестувати окремі ділянки коду незалежно від іншої частини програми. При належному складанні юніт-тести цілком можуть покрити більшу частину коду програми.

Більшість юніт-тестів так чи інакше мають ряд таких ознак:

Тестування невеликих ділянок коду ("юнітів")

При створенні юніт-тестів вибираються невеликі ділянки коду, які треба протестувати. Як правило, тестований ділянка повинна бути менше класу, а в більшості випадків тестується окремий метод класу. Орієнтація на невеликі ділянки дозволяє досить швидко писати нескладні тести.

Одного разу написаний код нерідко читають багато разів, тому важливо писати зрозумілий код. Особливо це важливо в юніт-тестах, де в разі невдачі при тестуванні розробник повинен швидко прочитати вихідний код і зрозуміти в чому проблема і як її виправити. А використання невеликих ділянок коду значно спрощує подібну роботу.

Тестування в ізоляції від решти коду

При тестуванні потрібно ізолювати тестований код від решти програми, з якою він взаємодіє, щоб потім чітко визначити можливість помилок саме в цьому ізольованому коді, що спрощує і підвищує контроль над окремими компонентами програми.

Тестування тільки загальнодоступних кінцевих точок

Всього лише невеликі зміни в класі можуть привести до невдачі багатьох юніт-тестів, оскільки реалізація використовуваного класу змінилася. Тому при написанні юніт-тестів обмежуються тільки загальнодоступними кінцевими точками, що дозволяє ізолювати юніт-тести від багатьох деталей внутрішньої реалізації компонента. В результаті зменшується ймовірність, що зміни в класах можуть привести до провалу юніт-тестів.

Автоматизація тестів

Написання юніт-тестів для невеликих ділянок коду веде до того, що кількість цих юніт-тестів стає дуже велике. І якщо процес отримання результатів і проведення тестів не автоматизовано, то це може привести до непродуктивної витраті робочого часу і зниження продуктивності. Тому важливо, щоб результати юніт-тестів представляли собою просте рішення, що означає, пройдений тест чи ні. Для автоматизації процесу розробники зазвичай звертаються до фреймворків юніт-тестування.

Фреймворки тестування

- MS Test: фреймворк юніт-тестування від компанії Microsoft, який за замовчуванням включений в Visual Studio (починаючи з VS 2012 у всі версії)
- NUnit: портований фреймворк з JUnit для платформи .NET
- xUnit.net: ще один фреймворк тестування від творців NUnit для платформи .NET

Розробка через тестування (Test-Driven-Development)

Розробка через тестування (TDD) процес застосування юніт-тестів, при якому спочатку пишуться тести, а потім вже програмний код, достатній для виконання цих тестів. Використання TDD дозволяє знизити кількість потенційних помилок в додатку. Створюючи тести перед написанням коду, ми тим самим описуємо спосіб поведінки майбутніх компонентів, не зв'язуючи себе при цьому з конкретною реалізацією цих тестованих компонентів (тим більше що реалізація на момент створення тесту ще не існує). Таким чином, тести допомагають оформити і описати API майбутніх компонентів.

Порядок написання коду при TDD досить простий:

- Пишемо юніт-тест
- Запускаємо його і бачимо, що він завершився невдачею (програмний код адже ще не написаний)
- Пишемо програмний код, достатній для запуску тесту

- Знову запускаємо тест і бачимо його результати

Цей цикл повторюється знову і знову, поки не буде закінчено роботу над програмним кодом. Так як більшість фреймворків юніт-тестування позначають невдалі тести червоним кольором (наприклад, виводиться текст червоного кольору), а пройдений тест позначається зеленим кольором (виводиться текст зеленого кольору), то даний цикл часто називають **червоним/зеленим циклом**.

Інтеграційні тести

Навіть якщо ми охопимо юніт-тестами практично всю логіку додатка, все одно можуть бути моменти, які будуть працювати не так, як треба. Крім того, складно протестувати представлення. Подібні речі тестуються за допомогою інтеграційних тестів, що виконуються на рівні веб-браузера. Для створення подібних тестів в Visual Studio 2013 версіях Ultimate і Premium доступний такий тип проекту як Coded UI Test Project. Крім того, доступні open-source рішення для створення інтеграційних тестів:

- WatiN - емулює поведінку користувача в браузері. Підтримує в тому числі сайти, які використовують аяx
- Lightweight Test Automation Framework - фреймворк, розроблений командою QA ASP.NET спеціально для платформи ASP.NET
- SeleniumHQ - набір інструментів для тестування в браузерах на різних платформах

Навантажувальні тести

Ще один вид тестів представляють навантажувальні тести, які покликані протестувати роботу сайту в умовах високого навантаження. Подібні тести дозволяють виявити вузькі місця при роботі з базою даних або при зверненні до диска і ряд інших проблем, які складно виявити іншими способами. Visual Studio Ultimate також має інструментарій для створення навантажувальних тестів.

4.2.2 Створення проекту для юніт-тестів

Подивимося на прикладі, як створювати юніт-тести. За замовчуванням при створенні проекту в будь-якої версії Visual Studio 2013 для створюваного проекту веб-додатку нам вже пропонується включити додатковий проект з тестами за допомогою опції Add unit tests:

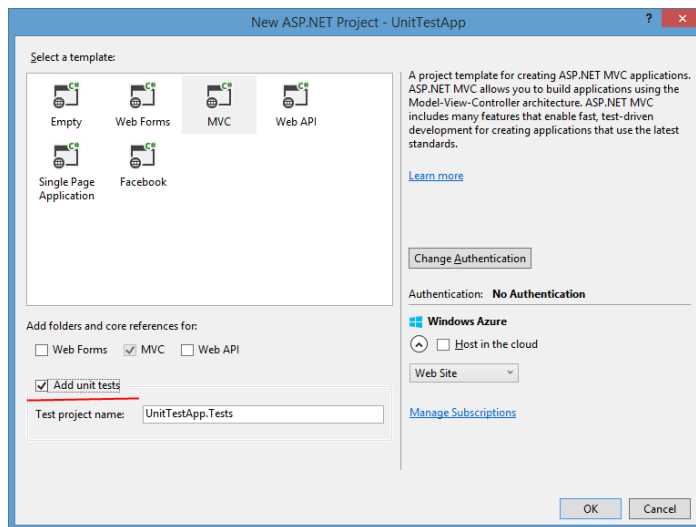


Рисунок 4.1 - Створення тестового проекту

Відзначимо даний чекбокс і створимо новий проект. Якщо раптом нам треба додати тести до вже існуючого проекту, то ми можемо додати в рішення новий тип проекту Unit Test Project. Як правило, для проекту тестів назва вибирається за наступною схемою [назва_головного_проекту].Tests:

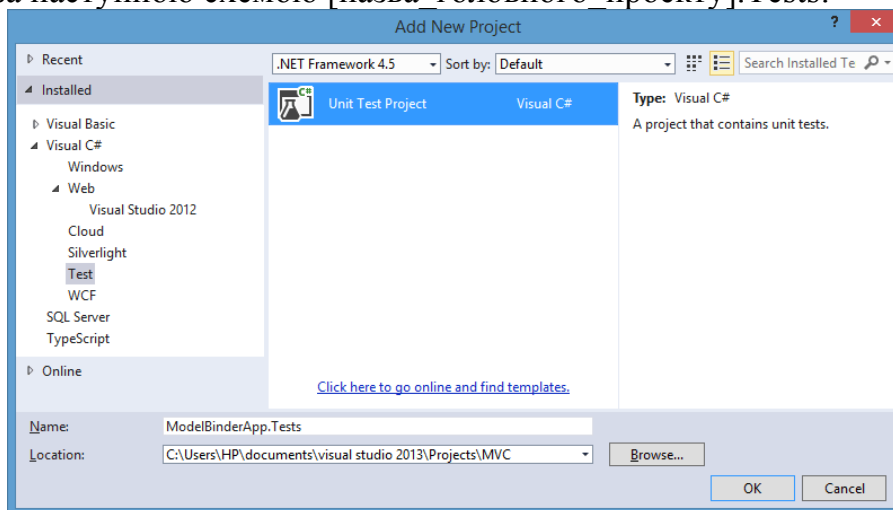


Рисунок 4.2 - Створення тестового проекту

При одночасному створенні обох проектів за замовчуванням тестовий проект вже містить ряд тестів:

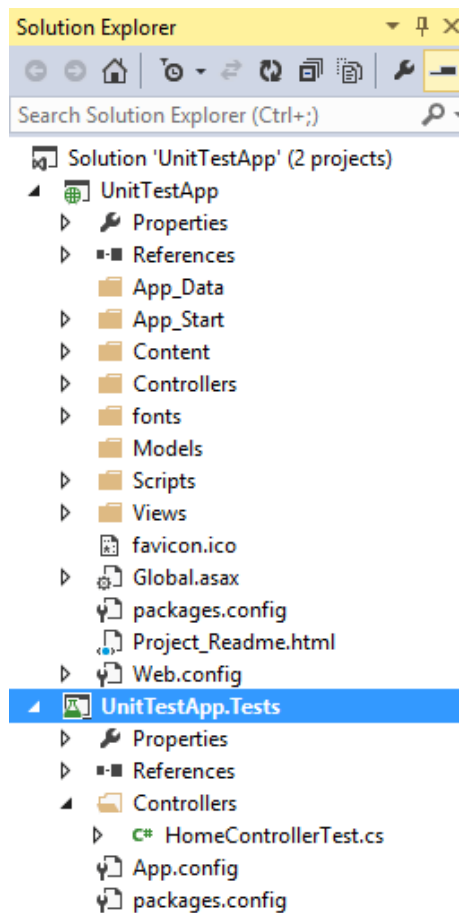


Рисунок 4.3 - Перевірка наявності тестів в проекті

Зокрема для контролера HomeController в проекті тестів буде створений клас HomeControllerTest:

```
[TestClass]
public class HomeControllerTest
{
    [TestMethod]
    public void Index()
    {
        // Arrange
        HomeController controller = new HomeController();
        // Act
        ViewResult result = controller.Index() as ViewResult;
        // Assert
        Assert.IsNotNull(result);
    }

    [TestMethod]
    public void About()
    {
        // Arrange
        HomeController controller = new HomeController();

        // Act
```

```

        ViewResult result = controller.About() as ViewResult;
        // Assert
        Assert.AreEqual("Your application description page.", result.ViewBag.Message);
    }

    [TestMethod]
    public void Contact()
    {
        // Arrange
        HomeController controller = new HomeController();
        // Act
        ViewResult result = controller.Contact() as ViewResult;
        // Assert
        Assert.IsNotNull(result);
    }
}

```

Клас тестів має атрибут [TestClass], а кожен метод, який проводить тестування, - атрибут [TestMethod]. Кожен метод містить три логічних частини - Arrange, Act і Assert. Вони позначені відповідними коментарями. Розглянемо, що вони роблять:

Секція Arrange виконує початкову ініціалізацію контексту тестів, а саме створює об'єкт контролера: `HomeController controller = new HomeController()`

Далі секція Act виконує дію, яку треба протестувати, а саме генерацію представлення: `ViewResult result = controller.Index() as ViewResult`. Так як метод `Index` контролера повертає об'єкт `ActionResult`, то його ще треба привести до об'єкта `ViewResult`.

Щоб перевірити на виконання попередню дію, в секції Assert викликається метод `Assert.IsNotNull (result)`. Цей метод перевіряє, чи був згенерований об'єкт представлення в попередньому розділі. Клас `Assert` ще має ряд методів, які дозволяють верифікувати результат.

Модель тестів **Arrange-Act-Assert** представляє собою не просто особливість тестування в Visual Studio, а цілу парадигму тестування:

Arrange: підготовка середовища, в якій виконується код

Act: тестування коду (зазвичай представляє один рядок коду)

Assert: переконуємося, що результат тесту саме той, що ми і очікували. Цей найпростіший стандартний тест не охоплює всіх можливих помилок, наприклад, чи було згенеровано потрібне представлення. Він просто покликаний дати початкове розуміння роботи тестів.

Тепер запустимо тест на виконання. Для цього перейдемо у вікно Test Explorer і натиснемо в ньому на кнопку Run All:

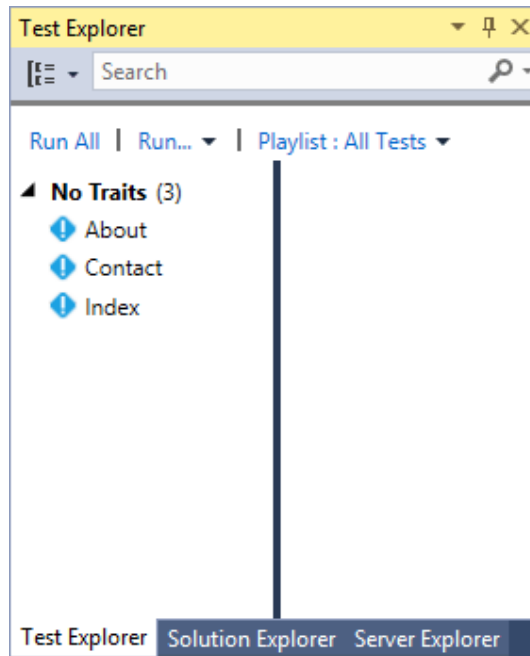


Рисунок 4.4 - Тест експлорер

Якщо воно за замовчуванням не відкрито, запустити тести можна через меню Test-> Run-> All Tests:

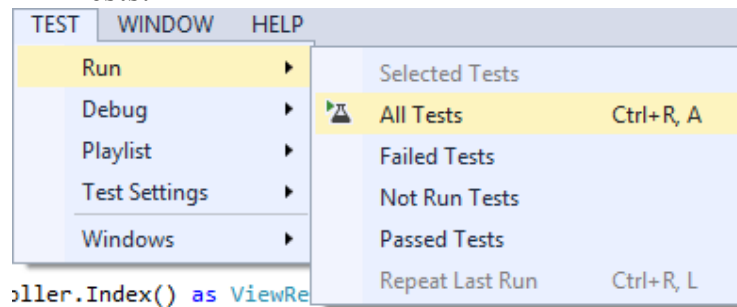


Рисунок 4.5 – Запуск тестів

Якщо все нормально, то оглядач тестів сигналізує нам зеленим кольором, що всі тести успішно пройдені:

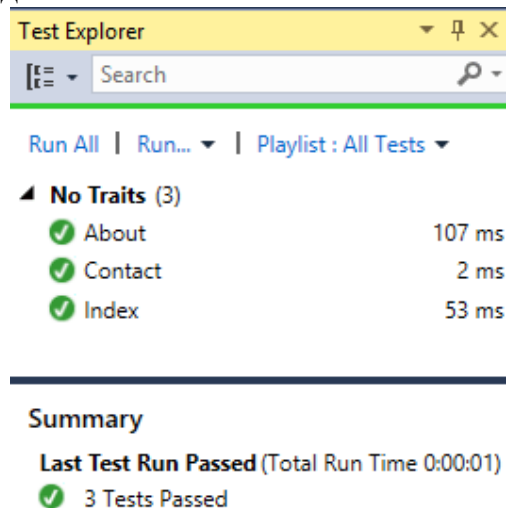


Рисунок 4.6 - Проходження тестів

4.2.3 Створення юніт-тестів

Візьмемо проект з минулої теми (або створимо новий) і додамо в головний проект веб-додатки в папку `Contollers` новий контролер `StoreController`:

```
public class StoreController : Controller
{
    public ActionResult Index()
    {
        ViewBag.Message = "Hello world!";
        return View();
    }
}
```

Контролер має тільки один метод, який встановлює властивість `ViewBag.Message` і генерує об'єкт `ActionResult`. А також додамо для методу `Index` представлення. Тепер перейдемо до проекту тестів і додамо в нього новий клас тестів. Для цього ми можемо додати або стандартний клас, або використовувати спеціальний шаблон файлів. Для цього в проекті тестів натиснемо правою кнопкою миші на каталог `Controllers` і в контекстному меню виберемо `Add->Unit Test ...`:

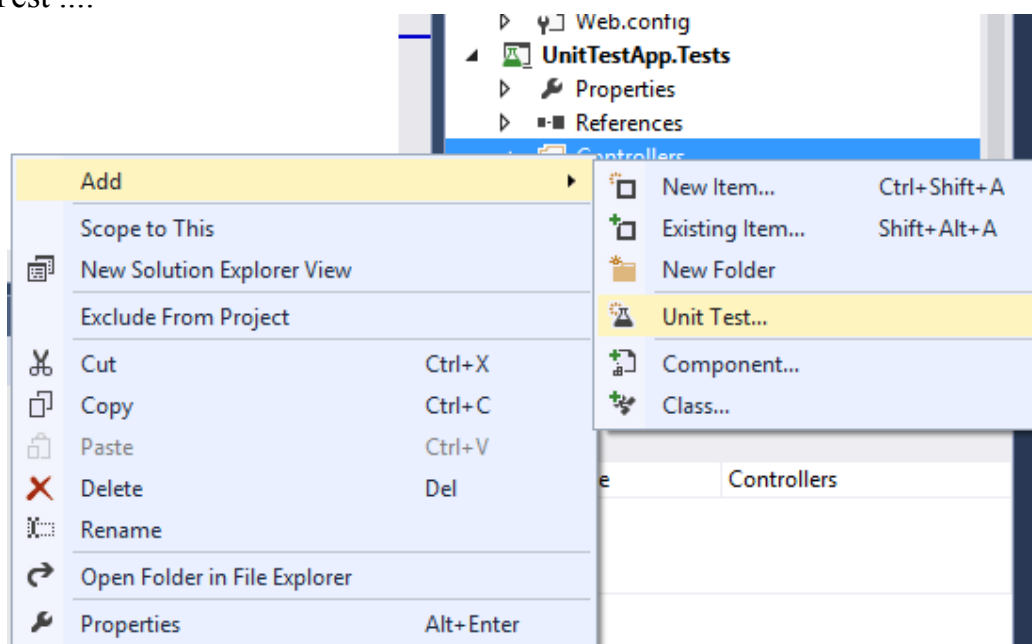


Рисунок 4.7 - Додавання юніт-тестів

За замовчуванням додає клас `Unit Test1`. По-перше, змінимо назву класу і файлу на `StoreControllerTest`. Потім змінимо наступним чином сам клас:

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using UnitTestApp.Controllers;
using System.Web.Mvc;

namespace UnitTestApp.Tests.Controllers
{
    [TestClass]
```

```

public class StoreControllerTest
{
    [TestMethod]
    public void IndexViewResultNotNull()
    {
        StoreController controller = new StoreController();
        ViewResult result = controller.Index() as ViewResult;
        Assert.IsNotNull(result);
    }

    [TestMethod]
    public void IndexViewEqualIndexCshtml()
    {
        StoreController controller = new StoreController();

        ViewResult result = controller.Index() as ViewResult;

        Assert.AreEqual("Index", result.ViewName);
    }

    [TestMethod]
    public void IndexStringInViewbag()
    {
        StoreController controller = new StoreController();

        ViewResult result = controller.Index() as ViewResult;

        Assert.AreEqual("Hello world!", result.ViewBag.Message);
    }
}

```

Метод `Index ViewResult Not Null` тестує результат методу - `ViewResult`, об'єкт, що повертається, не повинен мати значення `null`. Метод `IndexViewEqualIndexCshtml` перевіряє назву викликаного представлення за допомогою виклику `Assert.AreEqual`. А метод `IndexStringInViewbag` перевіряє значення рядка у властивості `ViewBag.Message`.

Хоча у нас тільки один метод в контролері, для нього ми створили три тестових методи для тесту кожного окремого тестового дії. Подібна ізоляція полегшує тестування окремих ділянок коду.

Перед запуском тестів перебудуємо головний проект. І запустимо тести. В цьому випадку ми побачимо, що один тест не пройдений - той, який верифікує представлення:

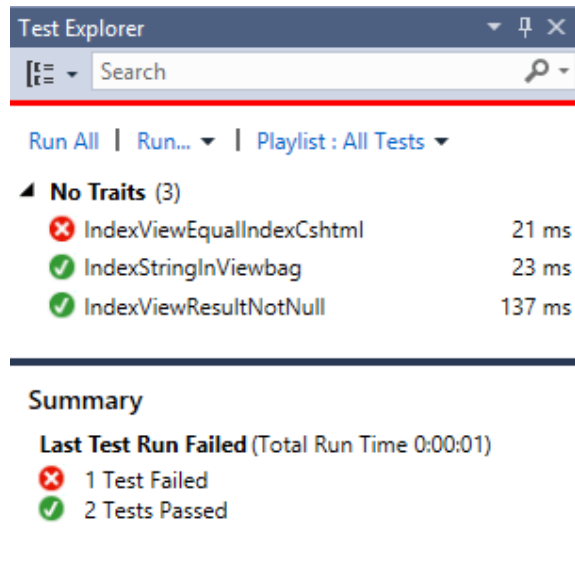


Рисунок 4.8 - Приклад непроходження тестів

Тест не пройдений, тому що при виклику методу View нам треба явно вказувати представлення. Тому змінимо метод Index в головному проекті наступним чином:

```
public ActionResult Index()
{
    ViewBag.Message = "Hello world!";
    return View("Index");
}
```

Знову запусимо тести. І тепер вже всі тести повинні бути успішно пройдені. Всі три дії мають одну і ту ж секцію Arrange, і, можливо, було б непогано відразу встановити всі початкові настройки для всіх методів. Для цього змінимо в тестовому проекті клас StoreControllerTest наступним чином:

```
[TestClass]
public class StoreControllerTest
{
    private StoreController controller;
    private ViewResult result;

    [TestInitialize]
    public void SetupContext()
    {
        controller = new StoreController();
        result = controller.Index() as ViewResult;
    }

    [TestMethod]
    public void IndexViewResultNotNull()
    {
        Assert.IsNotNull(result);
    }
}
```

```

}

[TestMethod]
public void IndexViewEqualIndexCshtml()
{
    Assert.AreEqual("Index", result.ViewName);
}

[TestMethod]
public void IndexStringInViewbag()
{
    Assert.AreEqual("Hello world!", result.ViewBag.Message);
}
}

```

Атрибут `TestInitialize` дозволяє задати метод, який виконує початкову ініціалізацію класу тестів. Завдяки цьому код скорочений, а в тестових методах залишені тільки частини `Assert`. Однак подібний підхід треба приймати з обережністю, так як він ускладнює можливості змінити код. В даному випадку загальний контекст дуже простий, але якщо при зміні методів буде змінюватися і їх контекст, то доведеться вносити великі зміни у всіх клас тестів, а не тільки в окремий метод для тестів.

Клас `Assert` і тестування результату

Клас `Assert` з простору імен `Microsoft.VisualStudio.TestTools.UnitTesting` за допомогою своїх статичних методів дозволяє верифікувати результат виконання деякого дії. Раніше вже було розглянуто кілька методів, зокрема, метод `Assert.IsNotNull()`, який перевіряє, чи не рівний деякий об'єкт значенням `null`. Крім того, при тестуванні нам доступний ще ряд методів:

- `AreEqual (object expected, object actual)`: перевіряє, чи рівні обидва об'єкти. Має різні перевантажені версії, що дозволяють порівнювати різні типи об'єктів
- `AreEqual <T> (T expected, T actual)`: узагальнена версія попереднього методу. Наприклад, `Assert.AreEqual<string>("Index", result.MasterName)`
- `AreNotEqual (object expected, object actual)`: перевіряє, чи не рівні обидва об'єкти. Тест проходить успішно, якщо об'єкти не рівні
- `AreNotEqual<T>(T expected, T actual)`: узагальнена версія попереднього методу
- `AreSame(object expected, object actual)`: перевіряє, чи вказують обидва об'єкти на один і той же об'єкт в пам'яті
- `AreNotSame(object expected, object actual)`: перевіряє, чи вказують обидва об'єкти на різні об'єкти в пам'яті. Якщо вони вказують на один і той же об'єкт, то тест закінчується невдало
- `Equals (object objA, object objB)`: перевіряє на рівність обидва об'єкти
- `IsFalse (bool condition)`: перевіряє, чи дорівнює умова `condition` значенням `false`

- IsTrue (bool condition): перевіряє, чи дорівнює умова condition значенням true
- IsNull (object value): перевіряє, чи має об'єкт value значення null
- InstanceOf (object value, Type expectedType): перевіряє, чи представляє об'єкт value тип expectedType

Використовуючи ці методи, ми можемо перевірити різні ситуації в своїй програмі.

4.2.4 Слабо пов'язані об'єкти і тестування роботи з БД

Більшість додатків так чи інакше взаємодіють з базою даних. Подивимося, як ми можемо протестувати дану проблему. Спочатку створимо новий проект, і разом з ним проект тестів:

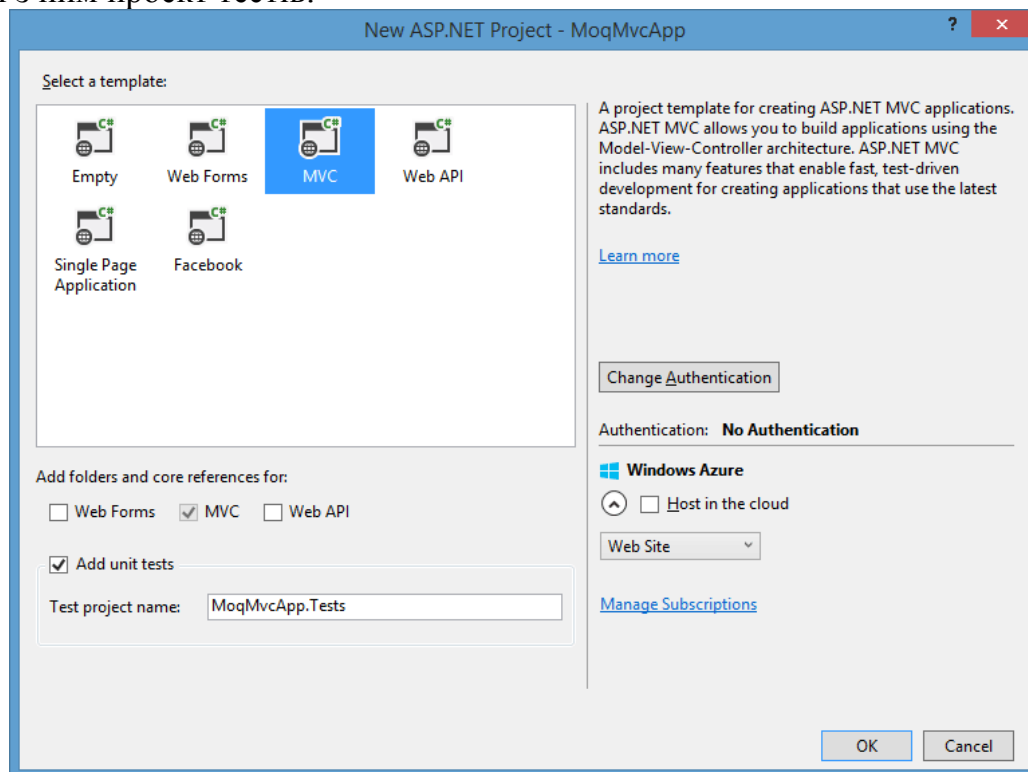


Рисунок 4.9 - Створення проекту з юніттестами

Потім визначаємо в проекті модель Computer і клас контексту CompContext, за допомогою яких йде взаємодія з базою даних:

```
// модель
public class Computer
{
    public int Id { get; set; }
    public string Name { get; set; }
}
// контекст
public class CompContext : DbContext
{
    public DbSet<Computer> Computers { get; set; }
}
```


Тепер відповідно до принципів TDD спочатку напишемо тест, який буде перевіряти модель представлення. У проекті тестів змінимо клас HomeControllerTest наступним чином:

```
[TestClass]
public class HomeControllerTest
{
    [TestMethod]
    public void IndexViewModelNotNull()
    {
        // Arrange
        HomeController controller = new HomeController();

        // Act
        ViewResult result = controller.Index() as ViewResult;

        // Assert
        Assert.IsNotNull(result.Model);
    }
}
```

Тут перевіряємо модель, яку передаємо з бази даних в представлення. Запустимо тест на виконання і побачимо, що він закінчується невдачею, так як в методі Index ми ще нічого не отримуємо з БД і не передаємо в представлення. Тепер пишемо мінімальний код, в самому контролері, який виробляє цю дію. Для цього змінимо контролер HomeController наступним чином:

```
public class HomeController : Controller
{
    CompContext db = new CompContext();
    public ActionResult Index()
    {
        return View(db.Computers);
    }
}
```

Перебудуємо проект і знову запустимо тести. Тепер зелений колір тестів сигналізує про їх успішне закінчення. Однак тут є одна проблема. Такий метод складно тестувати в силу жорсткого зв'язку з базою даних. Якщо щось трапиться з базою даних, і вона виявиться недоступна, тоді тест завершиться невдачею. Але це не означає, що щось не так у самому коді або що логіка побудована неправильно. Тому таких ситуацій зазвичай уникають. І для цього замість жорсткого зв'язку об'єктів використовують слабкозв'язані об'єкти (loosely coupled objects).

Безпосередньо в даному випадку проблема вирішується перенесенням рівня взаємодії з БД в спеціальний клас, який представляє реалізацію патерну репозиторій. Для цього додамо в головний проект новий інтерфейс і клас, його реалізує:

```
interface IRepository : IDisposable
```

```

{
    List<Computer> GetComputerList();
    Computer GetComputer(int id);
    void Create(Computer item);
    void Update(Computer item);
    void Delete(int id);
    void Save();
}

class ComputerRepository : IRepository
{
    private CompContext db;
    public ComputerRepository()
    {
        this.db = new CompContext();
    }
    public List<Computer> GetComputerList()
    {
        return db.Computers.ToList();
    }
    public Computer GetComputer(int id)
    {
        return db.Computers.Find(id);
    }

    public void Create(Computer c)
    {
        db.Computers.Add(c);
    }

    public void Update(Computer c)
    {
        db.Entry(c).State = EntityState.Modified;
    }

    public void Delete(int id)
    {
        Computer c = db.Computers.Find(id);
        if(c!=null)
            db.Computers.Remove(c);
    }

    public void Save()
    {
        db.SaveChanges();
    }

    private bool disposed = false;

    public virtual void Dispose(bool disposing)
    {

```

```

        if(!this.disposed)
        {
            if(disposing)
            {
                db.Dispose();
            }
        }
        this.disposed = true;
    }

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }
}

```

Тепер в контролері встановимо взаємодію з базою даних через репозиторій:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using MoqMvcApp.Models;

namespace MoqMvcApp.Controllers
{
    public class HomeController : Controller
    {
        IRepository repo;

        public HomeController(IRepository r)
        {
            repo = r;
        }
        public HomeController()
        {
            repo = new ComputerRepository();
        }

        public ActionResult Index()
        {
            return View();
        }
        protected override void Dispose(bool disposing)
        {
            repo.Dispose();
            base.Dispose(disposing);
        }
    }
}

```

```

    }
}
}

```

Тепер ми позбулися жорсткого зв'язку з базою даних. Але тепер нам треба буде в тестах зімітувати об'єкт сховища. І в цьому нам допоможе Moq-фреймворк.

4.2.5 Фреймворк Moq

Крім власне фреймворків для створення і проведення юніт-тестів при тестуванні часто бувають корисні такі фреймворки, які дозволяють імітувати або емулювати якусь функціональність або створювати мок-об'єкти. Подібних фреймворків існує безліч, і одним з найпопулярніших є Moq.

Отже, візьмемо проект з минулого теми і підключимо Moq через NuGet в проект тестів (а не в проект веб-додатку):

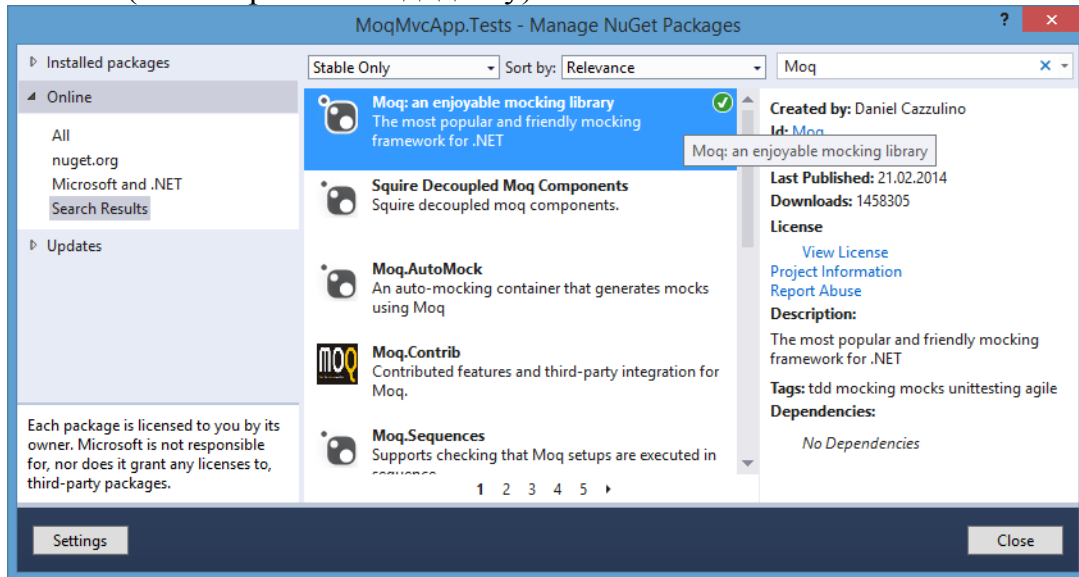


Рисунок 4.10 - Додавання мок-об'єктів

Після цього в вузол References проекту тестів буде додана бібліотека Moq, що містить всю основну функціональність.

Тепер змінимо клас тестів наступним чином:

```
using Moq;
```

```
.....
```

```
[TestClass]
```

```
public class HomeControllerTest
```

```
{
```

```
    [TestMethod]
```

```
    public void IndexViewModelNotNull()
```

```
    {
```

```
        // Arrange
```

```
        var mock = new Mock<IRepository>();
```

```
        mock.Setup(a => a.GetComputerList()).Returns(new List<Computer>());
```

```

HomeController controller = new HomeController(mock.Object);

// Act
ViewResult result = controller.Index() as ViewResult;

// Assert
Assert.IsNotNull(result.Model);
}
}

```

Для доступу до функціональності Moq спочатку підключається відповідний простір імен using Moq; Moq призначений для імітації об'єктів. В даному випадку імітується функціональність сховища. Для цього об'єкт Mock типізується відповідним типом: `var mock = new Mock <IRepository> ();`

Потім виконується настройка mock об'єкта за допомогою методу Setup. Так як нам треба імітувати повернення методом GetComputerList() набору об'єктів, то даний метод викликається в методі Setup, а за допомогою методу Returns визначаємо даний набір об'єктів.

Оскільки контролер HomeController тепер в конструкторі приймає об'єкт сховища, то ми можемо передати в конструктор мок-об'єкт, який імітує функціональність сховища: `HomeController controller = new HomeController (mock.Object);`

Тепер запусимо тест. Він повинен завершитися невдачею, так як у нас не передається в методі Index в представлення ніякої моделі. Тому змінимо метод Index:

```

public ActionResult Index()
{
    var model = repo.GetComputerList();
    return View(model);
}

```

І якщо ми зараз запусимо тест, то він пройде успішно. Тепер для прикладу додамо в клас тестів ще один метод:

```

[TestMethod]
public void IndexViewBagMessage()
{
    // Arrange
    var mock = new Mock<IRepository>();
    mock.Setup(a => a.GetComputerList()).Returns(new List<Computer>() { new Computer()});
    HomeController controller = new HomeController(mock.Object);
    string expected = "В базі даних 1 об'єкт";

    // Act
    ViewResult result = controller.Index() as ViewResult;
    string actual = result.ViewBag.Message as string;

    // Assert
    Assert.AreEqual(expected, actual);
}

```

```
}
```

Мета даного методу - перевірити повідомлення, передане через ViewBag. Причому ми хочемо, щоб повідомлення передавалося, якщо в базі даних більше 0 об'єктів. Для цього список, що повертається методом GetComputerList(), ініціалізується одним об'єктом. А метод Assert.AreEqual перевіряє обидва повідомлення.

Запустимо тест і побачимо, що він завершився невдачею. Тепер нам треба змінити метод Index в головному проекті, щоб він відповідав тесту:

```
public ActionResult Index()
{
    var model = repo.GetComputerList();
    if (model.Count > 0)
        ViewBag.Message = String.Format("В базі даних {0} об'єкт", model.Count);
    return View(model);
}
```

Оскільки за допомогою Моq ми імітували в методі контролера список з одним елементом, то якщо ми зараз запустимо тест, то він пройде успішно, так як рядок в тесті і значення ViewBag.Message будуть збігатися.

4.2.6 Тестування створення моделі і переадресації

Розглянемо на прикладі тестування створення моделі і переадресації. По-перше, нам потрібен мінімальний код для запуску тестів. Для цього візьмемо проект веб-додатку з минулого теми і визначимо в ньому стандартний метод Create:

```
public ActionResult Create()
{
    return View();
}
[HttpPost]
public ActionResult Create(Computer c)
{
    return View();
}
```

Це мінімальний код, достатній для написання тестів. Тепер в клас тестів HomeControllerTest додамо ряд методів:

```
[TestMethod]
public void CreatePostAction_ModelError()
{
    // arrange
    string expected = "Create";
    var mock = new Mock<IRepository>();
    Computer comp = new Computer();
    HomeController controller = new HomeController(mock.Object);
    controller.ModelState.AddModelError("Name", "Назва моделі не встановлена");
}
```

```
// act
ViewResult result = controller.Create(comp) as ViewResult;
// assert
Assert.IsNotNull(result);
Assert.AreEqual(expected, result.ViewName);
}
```

В даному методі перевіряється механізм валідації. Через об'єкт контролера ми можемо додати ряд помилок до властивості ModelState. І в разі помилок, відбудеться звернення до представлення Create.

Якщо ми запустимо, то тест не спрацює, тому що ніякої валідації в методі контролера немає. Тому виправимо метод Create:

```
[HttpPost]
public ActionResult Create(Computer c)
{
    if(ModelState.IsValid)
    {}
    return View("Create");
}
```

Запустимо тест знову. Тепер він спрацює успішно, і ми можемо переходити до нового тесту. Додамо в клас тестів наступний метод:

```
[TestMethod]
public void CreatePostAction_RedirectToIndexView()
{
    // arrange
    string expected = "Index";
    var mock = new Mock<IRepository>();
    Computer comp = new Computer();
    HomeController controller = new HomeController(mock.Object);
    // act
    RedirectToRouteResult result = controller.Create(comp) as RedirectToRouteResult;

    // assert
    Assert.IsNotNull(result);
    Assert.AreEqual(expected, result.RouteValues["action"]);
}
```

Тестування переадресації робиться також просто, як і тестування проедставлень. Об'єкт RedirectToRouteResult, який представляє переадресацію, має властивість RouteValues, яка дозволяє отримувати дані маршруту - в даному випадку отримуємо метод і порівнюємо його з очікуваним. І оскільки даний тест знову завершиться невдало, знову змінимо метод Create:

```
[HttpPost]
public ActionResult Create(Computer c)
{
```

```

    if(ModelState.IsValid)
    {
        return RedirectToAction("Index");
    }
    return View("Create");
}

```

Запустимо ще раз текст і переконаємося, що він завершився успішно. І тепер додамо третій метод тестів:

```

[TestMethod]
public void CreatePostAction_SaveModel()
{
    // arrange
    var mock = new Mock<IRepository>();
    Computer comp = new Computer();
    HomeController controller = new HomeController(mock.Object);
    // act
    RedirectToRouteResult result = controller.Create(comp) as RedirectToRouteResult;
    // assert
    mock.Verify(a => a.Create(comp));
    mock.Verify(a => a.Save());
}

```

У тестах ми легко можемо взяти результат тестованих методів і порівняти цей результат з певним значенням, щоб зрозуміти, чи правильно все працює. Але не всі методи повертають певні значення. Деякі методи повертають тип void, однак їх теж необхідно тестувати. І щоб протестувати подібні методи, в класі Mock визначено метод Verify. Його синтаксис:

```
mock.Verify(a=>a.Method (parameter)),
```

де mock - це об'єкт Mock, а parameter - значення, що передається в якості параметра в метод. В даному випадку викликається метод Create, в який передається модель для збереження, і метод Save. Знову ж тест не спрацює, тому змінимо метод Create наступним чином:

```

[HttpPost]
public ActionResult Create(Computer c)
{
    if(ModelState.IsValid)
    {
        repo.Create(c);
        repo.Save();
        return RedirectToAction("Index");
    }
    return View("Create");
}

```


Запустимо тест і переконаємося, що він завершиться успішно. Таким чином, ми можемо протестувати збереження моделі і механізм переадресації.

4.2.7 Управління життєвим циклом додатка

Життєвий цикл програми та запиту

Як правило, для вирішення більшості завдань в ASP.NET MVC не потрібно управління життєвим циклом додатку. Однак розуміння роботи програми та обробки запиту дає нам додаткові можливості по налаштуванню веб-додатку.

Життєвий цикл додатку ASP.NET MVC починається з запуску веб-додатку для обробки HTTP-запитів. І коли додаток завершує свою роботу, завершується і його життєвий цикл. Ключову роль в запуску програми грає файл Global.asax. Він визначає глобальний клас додатку. Наприклад, стандартний вміст файлу Global.asax може виглядати наступним чином:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Optimization;
using System.Web.Routing;

namespace LifeCycleApp
{
    public class MvcApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            AreaRegistration.RegisterAllAreas();
            FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
            RouteConfig.RegisterRoutes(RouteTable.Routes);
            BundleConfig.RegisterBundles(BundleTable.Bundles);
        }
    }
}
```

За замовчуванням глобальний клас додатку називається `MvcApplication`. Він успадковується від класу `System.Web.HttpApplication`, і під час вступу запиту до додатку фреймворк ASP.NET створює об'єкт класу `MvcApplication`.

Клас додатку підтримує два методи, які дозволяють управляти життєвим циклом програми: `Application_Start` і `Application_End`. Метод `Application_Start` викликається при початку роботи з програмою, а метод `Application_End` - перед завершенням його роботи.

За замовчуванням в класі програми визначається тільки метод `Application_Start`, в якому проводиться конфігураційна настройка: реєстрація маршрутів, фільтрів і т.д.

Метод `Application_End` викликається перед завершенням роботи програми, але, як правило, використовується для звільнення пов'язаних ресурсів, на зразок підключень до баз даних, але зараз, як правило, звільнення ресурсів можна зробити в коді самого додатка, і тому в методі `Application_End` немає великої необхідності.

При отриманні запиту додатком знову ж створюється об'єкт додатку у вигляді об'єкта класу `HttpApplication`, визначеного в `Global.asax`. Для обробки кожного запиту створюється свій об'єкт додатку, і потім починається життєвий цикл запиту.

Тобто фактично створення об'єкта додатка відбувається два рази - в рамках життєвого циклу програми та в рамках життєвого циклу запиту. У об'єкта додатка, який створюється для обробки конкретного запиту, методи `Application_Start` і `Application_End` не викликаються. Подібна конфігурація полегшує обробку запитів і дозволяє зберігати в нестатичних глобальних змінних запиту ті дані, які є специфічними для кожного запиту.

Життєвий цикл запиту передбачає виклик ряду подій в наступній послідовності:

1. **BeginRequest**: подія виникає, коли додаток отримує новий запит
2. **AuthenticateRequest/PostAuthenticateRequest**: подія `AuthenticateRequest` виникає при ідентифікації (аутентифікації) користувача, який зробив запит. А після його обробки спрацьовує подія `PostAuthenticateRequest`
3. **AuthorizeRequest/PostAuthorizeRequest**: `AuthorizeRequest` виникає при авторизації запиту, після цього спрацьовує подія `PostAuthorizeRequest`
4. **ResolveRequestCache/PostResolveRequestCache**: подія `ResolveRequestCache` виникає, коли додаток звертається до кешу для отримання даних. При отриманні даних їх кеша потім спрацьовує подія `PostResolveRequestCache`
5. **MapRequestHandler/PostMapRequestHandler**: `MapRequestHandler` спрацьовує при визначенні обробника запиту. Після вибору обробника спрацьовує подія `PostMapRequestHandler`
6. **AcquireRequestState/PostAcquireRequestState**: подія `AcquireRequestState` виникає при отриманні даних стану, пов'язаних із запитом (наприклад, дані сесії). І після нього спрацьовує подія `PostAcquireRequestState`
7. **PreRequestHandlerExecute/PostRequestHandlerExecute**: подія `PreRequestHandlerExecute` відбувається безпосередньо перед початком роботи обробника запиту, а подія `PostRequestHandlerExecute` - після його роботи
8. **ReleaseRequestState/PostReleaseRequestState**: подія `ReleaseRequestState` виникає, коли з додатком більше не потрібні

дані, асоційовані із запитом. І після звільнення цих даних відбувається подія `PostReleaseRequestState`

9. **UpdateRequestCache**: виникає при оновленні даних в кеші
10. **LogRequest/PostLogRequest**: подія `LogRequest` відбувається безпосередньо перед кожним логуванням, а `PostLogRequest` - після завершення всіх обробників подій
11. **EndRequest**: виникає при завершенні обробки запиту, коли дані для відповіді вже готові до відправки клієнтові `PreSendRequestHeaders`: виникає перед відправкою HTTP-заголовків браузеру клієнта
12. **PreSendRequestContent**: виникає після відправки заголовків, але перед відправкою основного вмісту відповіді, але якщо на певній стадії виникне помилка, то спрацює подія `Error`.

При необхідності ми можемо обробити ці події. Наприклад, у файлі `Global.asax`. Для цього нам треба визначити в класі `MvcApplication` методи, які мають таке найменування: `Application_ [Назва_події]`.

Наприклад, змінимо клас в `Global.asax` наступним чином:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Optimization;
using System.Web.Routing;

namespace LifeCycleApp
{
    public class MvcApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            AreaRegistration.RegisterAllAreas();
            FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
            RouteConfig.RegisterRoutes(RouteTable.Routes);
            BundleConfig.RegisterBundles(BundleTable.Bundles);
        }
        // обробка події BeginRequest
        protected void Application_BeginRequest()
        {
            AddEvent("BeginRequest");
        }
        // обробка події AuthenticateRequest
        protected void Application_AuthenticateRequest()
        {
            AddEvent("AuthenticateRequest");
        }
        // обробка події PreRequestHandlerExecute
        protected void Application_PreRequestHandlerExecute()
```

```

    {
        AddEvent("PreRequestHandlerExecute");
    }
    private void AddEvent(string name)
    {
        List<string> eventList = Application["events"] as List<string>;
        if (eventList == null)
        {
            Application["events"] = eventList = new List<string>();
        }
        eventList.Add(name);
    }
}

```

Обробка кожної події зведена до додавання інформації в метод AddEvent. Для зберігання інформації про події створюється список, що зберігається у властивості Application["events"]. Об'єкт Application є екземпляром класу HttpSessionState, який представляє стан додатку, і ми можемо визначити в ньому властивість events.

Після цього ми також можемо звернутися до цієї властивості в додатку, наприклад, в контролері HomeController. Нехай в ньому відбувається виведення всіх відпрацьованих подій на сторінку:

```

public class HomeController : Controller
{
    public string Index()
    {
        List<string> events = HttpContext.Application["events"] as List<string>;
        string result = "<ul>";
        foreach (string e in events)
            result += "<li>" + e + "</li>";
        result += "</ul>";
        return result;
    }
}

```

Крім спеціальних методів ми також можемо обробити події стандартним для C# способом - через підключення обробників подій. Для цього змінимо клас MvcApplication в Global.asax наступним чином:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Optimization;
using System.Web.Routing;

namespace LifeCycleApp

```

```

{
public class MvcApplication : System.Web.HttpApplication
{
    public MvcApplication()
    {
        BeginRequest += (src, args) => AddEvent("BeginRequest");
        AuthenticateRequest += (src, args) => AddEvent("AuthenticateRequest");
        PreRequestHandlerExecute += (src, args) => AddEvent("PreRequestHandlerExecute");
    }

    protected void Application_Start()
    {
        AreaRegistration.RegisterAllAreas();
        FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
        RouteConfig.RegisterRoutes(RouteTable.Routes);
        BundleConfig.RegisterBundles(BundleTable.Bundles);
    }
    private void AddEvent(string name)
    {
        List<string> eventList = Application["events"] as List<string>;
        if (eventList == null)
        {
            Application["events"] = eventList = new List<string>();
        }
        eventList.Add(name);
    }
}
}

```

За допомогою лямбда-виразів встановлюються обробники подій :

(src, args) => AddEvent ("BeginRequest").

Як параметри передаються об'єкт, що викликав подію - src, і параметр EventArgs - args. Результат виконання методу в контролері HomeController буде тим же. В Global.asax ми можемо використовувати контекст запиту HttpContext і його властивості. Наприклад, опрацюємо подію:

```

public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        AreaRegistration.RegisterAllAreas();
        FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
        RouteConfig.RegisterRoutes(RouteTable.Routes);
        BundleConfig.RegisterBundles(BundleTable.Bundles);
    }

    protected void Application_PostRequestHandlerExecute(object sender, EventArgs e)
    {
        HttpContext.Current.Response.Write("Application_PostRequestHandlerExecute");
    }
}

```

В даному випадку у відповідь за допомогою методу `Response.Write()` додається назва методу, який спрацював.

4.2.8 Модулі

У ASP.NET MVC модулі представляють спеціальні класи, що реалізують інтерфейс `System.Web.IHttpModule`. Їх ініціалізація відбувається після створення об'єкта додатку. За допомогою модулів ми можемо управляти обробкою запиту в додатку. Фреймворк ASP.NET MVC вже використовує ряд вбудованих модулів для різних завдань:

- **AnonymousIdentification:** представлений класом `System.Web.Security.AnonymousIdentificationModule`. Відповідає за ідентифікацію запитів навіть у тих випадках, коли користувач не аутентифікований - тобто для анонімних запитів
- **DefaultAuthentication:** представлений класом `System.Web.Security.DefaultAuthenticationModule`. Відповідає за установку властивості `User` у об'єкта `HttpContext`
- **FileAuthorization:** представлений класом `System.Web.Security.FileAuthorizationModule`, який використовується при аутентифікації Windows
- **FormsAuthentication:** представлений класом `System.Web.Security.FormsAuthenticationModule`. Даний модуль використовується при аутентифікації форм і встановлює значення властивості `HttpContext.User`, за допомогою якого ми в додатку можемо отримати аутентифікованого користувача
- **OutputCache:** представлений класом `System.Web.Caching.OutputCacheModule`. Відповідає за кешування відповіді клієнта
- **PageInspectorHttpModule:** цей модуль підтримує функціональність Page Inspector, яка є в Visual Studio і за допомогою якої здійснюється налагодження HTML і CSS
- **Profile:** представляє клас `System.Web.Profile.ProfileModule`, який пов'язує дані профілю користувача з даними в запиті
- **RoleManager:** представлений класом `System.Web.Security.RoleManagerModule`, який управляє призначенням ролей
- **ScriptModule-4.0:** представлений класом `System.Web.Handlers.ScriptModule`. Призначений для підтримки Ajax-запитів
- **ServiceModel-4.0:** представлений класом `System.ServiceModel.Activation.ServiceHttpModule`. Цей модуль використовується веб-службами ASP.NET
- **Session:** реалізований класом `System.Web.SessionState.SessionStateModule`. Призначений для зв'язку даних сесії з запитами

- **UrlAuthorization:** представлений класом System.Web.Security.UrlAuthorizationModule, який забезпечує авторизований доступ до ресурсів
- **UrlMappingsModule:** представлений класом System.Web.UrlMappingsModule. Призначений для підтримки функції URL Mappings, яка в ASP.NET MVC не застосовується
- **UrlRoutingModule-4.0:** реалізований класом System.Web.Routing.UrlRoutingModule, який використовується системою маршрутизації
- **WebPageHttpModule:** даний модуль зіставляє запити з файлами представлень
- **WindowsAuthentication:** представляє клас System.Web.Security.WindowsAuthenticationModule. Відповідає за аутентифікацію Windows

В принципі ми можемо самі отримати список використовуваних модулів в своєму додатку. Для цього в контролері отримаємо всі модулі через властивість HttpContext.ApplicationInstance.Modules. Припустимо, у нас наступний метод отримує модулі і передає їх в представлення:

```
public ActionResult Modules()
{
    var modules = HttpContext.ApplicationInstance.Modules;
    string[] modArray = modules.AllKeys;
    return View(modArray);
}
```

І в поданні здійснимо простий вивід масиву:

```
@model IEnumerable<string>
```

```
<ul>
    @foreach(string s in Model)
    {
        <li>@s</li>
    }
</ul>
```

4.2.9 Створення власного модуля

Інтерфейс Http Module визначає ряд методів, які класи модулів повинні реалізувати:

- **Init(app):** викликається при створенні модуля і його передачі об'єкту HttpApplication, в якому цей модуль реєструється
- **Dispose():** викликається, коли обробка запиту завершена. У цьому методі відбувається звільнення пов'язаних ресурсів

Створимо простий модуль, який буде вимірювати час обробки запиту. Для цього додамо в проект новий каталог Modules, в якому будуть знаходитися наші модулі. Потім додамо в цей каталог новий клас. Назвемо його TimerModule:

```

using System.Diagnostics;
//.....
public class TimerModule : IHttpModule
{
    private Stopwatch timer;
    public void Init(HttpApplication app)
    {
        app.BeginRequest += HandleBeginRequest;
        app.EndRequest += HandleEndRequest;
    }
    private void HandleBeginRequest(object src, EventArgs args)
    {
        timer = Stopwatch.StartNew();
    }
    private void HandleEndRequest(object src, EventArgs args)
    {
        HttpContext context = HttpContext.Current;
        context.Response.Write(string.Format(
            "<div style='color:red;'>Час обробки запиту: {0:F5} секунд</div>",
            ((float)timer.ElapsedTicks) / Stopwatch.Frequency));
    }
    public void Dispose()
    {}
}

```

Для замірів часу використовується клас `Stopwatch` з простору імен `System.Diagnostics`. У методі `Init()` встановлюються обробники для подій початку і завершення обробки запиту. Обидва обробники приймають два параметра: `src` - об'єкт докладання `HttpApplication` і `args` - аргументи події `EventArgs`.

Оброблювач початку обробки запиту просто запускає таймер. А обробник завершення обробки запиту отримує минулий час і додає інформацію про нього у відповідь клієнту через метод `Response.Write()`.

Метод `Dispose()` залишений порожнім, так як нам не треба звільняти ніяких ресурсів, пов'язаних з модулем. Після визначення модуля його треба зареєструвати в файлі `web.config`. Для цього додамо в файл в межах вузла `configuration` наступну секцію:

```

<system.webServer>
  <modules>
    <add name="Timer" type="LifeCycleApp.Modules.TimerModule"/>
  </modules>
</system.webServer>

```

В даному випадку у нас додаток називається `LifeCycleApp`, а модуль `TimerModule` знаходиться в папці `Modules`, тому повна назва типу буде `LifeCycleApp.Modules.TimerModule`.

4.2.10 Взаємодія між модулями. Події модулів

При реєстрації модулів всі вони потрапляють в колекцію `Modules`, яка є у класу додатка `HttpApplication`. Отримавши один модуль з цієї колекції, ми можемо з ним взаємодіяти - отримувати його поля і властивості і підписуватися на події. Наприклад, змінимо модуль з попередньої теми наступним чином:

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Web;

namespace LifeCycleApp.Modules
{
    public class RequestTimerEventArgs : EventArgs
    {
        public float Duration { get; set; }
    }
    public class TimerModule : IHttpModule
    {
        private Stopwatch timer;
        public event EventHandler<RequestTimerEventArgs> RequestTimed;
        public void Init(HttpApplication app)
        {
            app.BeginRequest += HandleBeginRequest;
            app.EndRequest += HandleEndRequest;
        }
        private void HandleBeginRequest(object src, EventArgs args)
        {
            timer = Stopwatch.StartNew();
        }
        private void HandleEndRequest(object src, EventArgs args)
        {
            HttpContext context = HttpContext.Current;
            float duration = ((float)timer.ElapsedTicks) / Stopwatch.Frequency;
            context.Response.Write(string.Format(
                "<div style='color:red;'>Час обробки запиту: {0:F5} секунд</div>", duration));
            if (RequestTimed != null)
            {
                RequestTimed(this,
                    new RequestTimerEventArgs { Duration = duration });
            }
        }
        public void Dispose()
        {}
    }
}
```

На відміну від старої версії модуля з минулої теми тут додається подія і здійснюється його виклик в обробнику події `EndRequest`. Тепер додамо ще один

модуль. Назвемо його TotalTimeModule. Він буде розраховувати кількість запитів і загальний час обробки запитів на основі даних з попереднього модуля:

```
public class TotalTimeModule : IHttpModule
{
    private static float totalTime = 0;
    private static int requestCount = 0;
    public void Init(HttpApplication app)
    {
        IHttpModule module = app.Modules["Timer"];
        if (module != null && module is TimerModule)
        {
            TimerModule timerModule = (TimerModule)module;
            timerModule.RequestTimed += HandleRequestTimed;
        }
        app.EndRequest += HandleEndRequest;
    }
    private void HandleRequestTimed(object src, RequestTimerEventArgs e)
    {
        totalTime += e.Duration;
        requestCount++;
    }
    private void HandleEndRequest(object src, EventArgs e)
    {
        HttpContext context = HttpContext.Current;
        string result=string.Format("<div style='color:red;'>Кількість запитів: {0} </div>" +
            "<div style='color:red;'>Загальний час обробки запитів: {1:F5} сек. </div>",
            requestCount, totalTime);
        context.Response.Write(result);
    }
    public void Dispose()
    {}
}
```

Так як попередній модуль називається у нас Timer (як визначено в web.config), то з цього імені ми можемо отримати цей модуль: app.Modules ["Timer"]. І потім підписатися на його подію. Обробка події проста: змінюємо значення змінних totalTime і requestCount. Також додамо новий модуль в файл web.config:

```
<system.webServer>
<modules>
    <add name="Timer" type="LifeCycleApp.Modules.TimerModule"/>
    <add name="Total" type="LifeCycleApp.Modules.TotalTimeModule"/>
</modules>
</system.webServer>
```

І оскільки другий модуль залежить від першого - бере його дані для підрахунку своїх полів, то модуль Total оголошується другим.

4.2.11 HTTP-обробники

HTTP-обробники (HTTP Handlers) використовуються для генерації вмісту відповіді на HTTP-запит. Ми можемо використовувати HTTP-обробники для настройки обробник запитів. При обробці одного запиту ми можемо задіяти кілька різних модулів, але тільки один http-обробник може бути зіставлений із запитом. HTTP-обробник представляє клас, який реалізує інтерфейс `System.Web.IHttpHandler`. Даний інтерфейс визначає один метод і одну властивість:

- `ProcessRequest (context)`: даний метод як параметр приймає об'єкт контексту запиту `HttpContext` і генерує відповідь клієнту
- `IsReusable`: це властивість вказує, чи буде даний обробник використовуватися іншими запитами

HTTP-обробник вибирається системою після виникнення події `MapRequestHandler`. І відразу після вибору обробника спрацьовує подія `PostMapRequestHandler`. Безпосередня генерація відповіді оброблювачем відбувається після події `PreRequestHandlerExecute` - після цієї події відбувається виклик методу `ProcessRequest` і генерація відповіді. А відразу після генерації відповіді виникає подія `PostRequestHandlerExecute`. Розглянемо на прикладі: Створимо свій Http-обробник. Для цього спочатку визначимо в проекті каталог `Handlers`, який буде містити файли обробників. Потім в цей каталог додамо новий клас `UserInfoHandler`:

```
public class UserInfoHandler : IHttpHandler
{
    public void ProcessRequest(HttpContext context)
    {
        string result = "<p>Ваш IP: "+context.Request.UserHostAddress+"</p>";
        result+="<p>UserAgent: "+context.Request.UserAgent+"</p>";
        context.Response.Write(result);
    }
    public bool IsReusable
    {
        get { return false; }
    }
}
```

Оброблювач просто повертає ip і дані рядка user-agent клієнта. Тепер цей обробник треба підключити до обробки запитів. Це можна зробити двома способами. Перший спосіб полягає у виклику обробника через обробник маршрутів. Для цього відкриємо файл `RouteConfig` і змінимо його зміст наступним чином:

```
using LifecycleApp.Handlers;
using System;
```

```

using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace LifeCycleApp
{
    public class RouteConfig
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

            routes.Add(new Route("handler/{*path}", new CustomRouteHandler()));

            routes.MapRoute(
                name: "Default",
                url: "{controller}/{action}/{id}",
                defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }
            );
        }
    }
    // обробник маршруту
    class CustomRouteHandler : IRouteHandler
    {
        public IHttpHandler GetHttpHandler(RequestContext requestContext)
        {
            return new UserInfoHandler();
        }
    }
}

```

Вираз `routes.Add (new Route ("handler / { * path }", new Custom Route Handler ()))` додає новий маршрут, який обробляє клас `CustomRouteHandler`. У свою чергу цей обробник маршрутів буде викликати визначений вище HTTP-обробник. Другий спосіб полягає у визначенні обробника в файлі `web.config`. Для цього в файлі конфігурації в вузлі `system.webServer` (якщо його немає, то треба створити) пропишемо визначення обробника:

```

<?xml version="1.0" encoding="utf-8"?>
<configuration>
<!--вміст-->
<system.webServer>
    <handlers>
        <add name="MyHttpHandler" path="/handler/" verb="GET"
type="LifeCycleApp.Handlers.UserInfoHandler"/>
    </handlers>
</system.webServer>
</configuration>

```

Все обробники задаються в елементі handlers. Кожна пропозиція має такі атрибути:

- name: унікальна назва обробника path: запит URL, який буде оброблятися оброблювачем
- verb: тип запит, наприклад, GET. Ми також можемо поставити будь-який тип запиту - в цьому випадку використовується зірочка (*)
- type: повний тип класу обробника

Крім реєстрації в web.config нам треба також додати ігнорування маршруту в файлі RouteConfig.cs, інакше система маршрутизації буде неправильно обробляти даний маршрут:

```
public class RouteConfig
{
    public static void RegisterRoutes(RouteCollection routes)
    {
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

        routes.IgnoreRoute("handler/{*path}");

        routes.MapRoute(
            name: "Default",
            url: "{controller}/{action}/{id}",
            defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }
        );
    }
}
```

Скориставшись одним із способів реєстрації HTTP-обробника, ми зможемо звернутися до сервера з запитом типу `http://localhost:3406/handler/`, і тоді браузер виведе інформацію з ip-адресою і user-agentом.

Створення асинхронних HTTP-обробників

При обробці різних запитів є сенс зробити обробники асинхронними, наприклад, коли мова йде про отримання даних з БД або від зовнішнього ресурсу. Асинхронні обробники відрізняються від синхронних тим, що клас обробника успадковується від класу `HttpTaskAsyncHandler`.

Отже, створимо новий клас `UserInfoAsyncHandler`:

```
public class UserInfoAsyncHandler : HttpTaskAsyncHandler
{
    public override async Task ProcessRequestAsync(HttpContext context)
    {
        string idString = context.Request.Url.Segments[2];
        int id;
        Int32.TryParse(idString, out id);
        User user;
        string result = "";
    }
}
```

```

        using(UserContext db = new UserContext())
        {
            user = await db.Users.FindAsync(id);
        }
        if(user!=null)
        {
            result += "<p>Id=" + user.Id.ToString() + " Name: " + user.Name+"</p>";
        }
        context.Response.Write(result);
    }
}

```

В даному випадку ми отримуємо третій сегмент рядка запиту, який повинен містити id потрібного нам запису, і отримавши id, виводимо з бази даних відповідні йому дані.

Також зареєструємо новий маршрут, запити по якому будуть оброблятися даним оброблювачем, в файлі RouteConfig.cs:

```

public class RouteConfig
{
    public static void RegisterRoutes(RouteCollection routes)
    {
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

        routes.Add(new Route("User/{id}", new CustomRouteHandler()));

        routes.MapRoute(
            name: "Default",
            url: "{controller}/{action}/{id}",
            defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }
        );
    }
}

class CustomRouteHandler : IRouteHandler
{
    public IHttpHandler GetHttpHandler(RequestContext requestContext)
    {
        return new UserInfoAsyncHandler();
    }
}

```

Новий маршрут буде відповідати запитам типу <http://localhost:8989/user/1>, де останній сегмент буде вилучатись оброблювачем і використовуватися для отримання інформації з бази даних.

4.2.12 Перевизначення HTTP-обробника

Раніше ми дивилися, як можна призначити обробник для обробки різних маршрутів. Але ми також можемо перевизначити базовий обробник запиту при наявності деяких умов.

За допомогою модулів ми можемо перевизначити дії http-обробника, не вносячи великих змін в код програми. При перевизначенні обробника нам треба враховувати, що обробник запиту вибирається при виникненні події `MapRequestHandler`. Тому перевизначення обробника має відбуватися на більш ранній стадії.

Отже, додамо в проект наступний модуль:

```
public class HandlerSelectionModule : IHttpModule
{
    public void Init(HttpApplication app)
    {
        app.PostResolveRequestCache += (src, args) =>
        {
            if (string.Equals((string)app.Context.Request.RequestContext.RouteData.Values["controller"],
                "Home", StringComparison.OrdinalIgnoreCase) &&
                string.Equals((string)app.Context.Request.RequestContext.RouteData.Values["action"],
                "Index", StringComparison.OrdinalIgnoreCase))
            {
                app.Context.RemapHandler(new UserInfoHandler());
            }
        };
    }
    public void Dispose()
    {}
}
```

Оскільки нам треба перевизначити обробник запиту до події `MapRequestHandler`, то ми це робимо при обробці іншої події - `PostResolveRequestCache`, яке виникає до `MapRequestHandler`. Модуль порівнює дані, отримані з контексту запиту, зокрема, порівнюємо контролер і метод. І якщо виявиться, що запит призначався до методу `Index` контролера `HomeController`, то викликається http-обробник `UserInfoHandler`, який я взяв з однієї з попередніх тем.

Для перевизначення http-обробника використовується метод `RemapHandler()`, який є у об'єкта контексту `HttpContext`, і після визначення модуля зареєструємо його в `web.config`:

```
<system.webServer>
  <modules>
    <add name="Selector" type="LifeCycleApp.Modules.HandlerSelectionModule"/>
  </modules>
  <handlers>
    <add name="MyHttpHandler" path="*" verb="GET"
      type="LifeCycleApp.Handlers.UserInfoHandler" />
  </handlers>
</system.webServer>
```

```
</handlers>
</system.webServer>
```

І якщо ми перейдемо на запит Home/Index, то даний запит буде перехоплений модулем, і його обробник буде змінений на UserInfoHandler.

Ми можемо проконтролювати вибір обробника, додавши у відповідь клієнту назва http-обробника. Для цього додамо в метод Init() модуля оброблювач події EndRequest:

```
app.EndRequest += (src, args) =>
{
    app.Context.Response.Write("<p>Handler: " + app.Context.Handler.ToString() + "</p>");
};
```

4.2.13 Виконання дій по таймеру

Однією з найбільш поширених завдань при створенні модулів і обробників є виконання деяких дій, що повторюються по таймеру, наприклад, поштових розсилок, записів в логи і т.д. Створимо свій модуль, який буде нести подібну функціональність. Наприклад, визначимо модуль, який буде відправляти лист на пошту в певний час:

```
public class TimerModule : IHttpModule
{
    static Timer timer;
    long interval = 30000; //30 секунд
    static object synclock = new object();
    static bool sent=false;

    public void Init(HttpApplication app)
    {
        timer = new Timer(new TimerCallback(SendEmail), null, 0, interval);
    }

    private void SendEmail(object obj)
    {
        lock (synclock)
        {
            DateTime dd = DateTime.Now;
            if (dd.Hour == 1 && dd.Minute == 30 && sent == false)
            {
                // настройки smtp-сервера, з якого ми будемо відправити листа
                SmtpClient smtp = new System.Net.Mail.SmtpClient("smtp.gmail.com", 587);
                smtp.EnableSsl = true;
                smtp.Credentials = new System.Net.NetworkCredential("somemail@gmail.com",
                    "password");

                // наш email з заголовком листа
```



```

MailAddress from = new MailAddress("somemail@gmail.com", "Test");
// кому відправляємо
MailAddress to = new MailAddress("othermail@yandex.ru");
// створюємо об'єкт повідомлення
MailMessage m = new MailMessage(from, to);
// тема листа
m.Subject = "Test mail";
// текст письма
m.Body = "Розсилка сайту";
smtp.Send(m);
sent = true;
}
else if (dd.Hour != 1 && dd.Minute !=30)
{
    sent = false;
}
}
}
public void Dispose()
{
}
}

```

Даний модуль налаштований на відправку листа в 1:30. У методі Init() запускається таймер, який кожні 30000 мілісекунд (30 секунд) викликає метод SendEmail(). Метод SendEmail() за допомогою оператора lock визначає критичну секцію, доступ до якої одночасно можливий тільки для одного потоку. Для управління відправленням ми перевіряємо поточний час і значення змінної sent і в залежності від результатів перевірки відправляємо лист.

Подібним чином можна визначити і інші дії, що виконуються з певною періодичністю, і після визначення модуля зареєструємо його в web.config:

```

<system.webServer>
  <modules>
    <add name="Timer" type="MvcTimerApp.Modules.TimerModule"/>
  </modules>
</system.webServer>

```

Контрольні питання.

1. Принципи TDD.
2. Переваги і недоліки використання фреймворку Moq.
3. Принципи юніт-тестування.
4. Принципи управління життєвим циклом додатку.
5. Спосіб перевизначення HTTP-обробника.

4.3 Кешування в ASP.NET MVC

4.3.1 Атрибут OutputCache

Кешування контенту може відігравати важливу роль в роботі програми. За допомогою кешування ми можемо зменшити кількість звернень до бази даних, оптимізувати обробку запитів і тим самим зменшити навантаження на сервер і підвищити продуктивність.

Найбільш простий спосіб кешування даних представляє використання атрибута OutputCache. Даний атрибут може застосовуватися як до всього контролера, так і до окремих його методів. Застосування атрибута до контролера дозволяє задати єдину політику кешування до всіх методів даного контролера.

Дія атрибута OutputCache можна налаштувати за допомогою його властивостей:

- **CacheProfile**: визначає конфігурацію кешування
- **Duration**: тривалість кешування контенту в секундах. Важливо оптимально підібрати часовий відрізок, так як якщо задати велике значення, то користувачі будуть працювати практично зі статичними даними, а при дуже маленькому значенні буде занадто часто оновлення кешу, що нівелює переваги його використання
- **Location**: місце, де розміщується кеширований контент
- **NoStore**: якщо дане значення дорівнює true, у відповіді в заголовку Cache-Control встановлюється флаг no-store, що вказує браузеру, що контент не треба зберігати
- **SqlDependency**: визначає залежність між кешем і таблицею в БД
- **VaryByCustom**: вказує довільне значення, за яким будуть визначатися різні версії кешованих даних
- **VaryByHeader**: вказує на набір заголовків, за якими визначатимуться різні версії кешованих даних
- **VaryByParam**: вказує на параметри рядка запиту або параметри переданих форм, за якими визначатимуться різні версії кешованих даних

За допомогою властивості Location можна визначити як сховище кешованого контенту як сервер, так і браузер клієнта. Зокрема, ця властивість може приймати одне із значень перерахування System.Web.UI.OutputCacheLocation:

- **Any**: контент кеширується як на клієнті, так і на проксі-сервері і в вихідному кеші сервера
- **Client**: контент кеширується на клієнті
- **Downstream**: контент кешується як на клієнті, так і на проксі-сервері. Вихідний кеш сервера не використовується
- **None**: для заголовка Cache-Control встановлюється значення no-cache, що означає, що контент не буде показаний кешуватися
- **Server**: контент кеширується тільки в вихідному кеші сервера

- **ServerAndClient:** контент кеширується на клієнті і в вихідному кеші сервера

Розглянемо на прикладі:

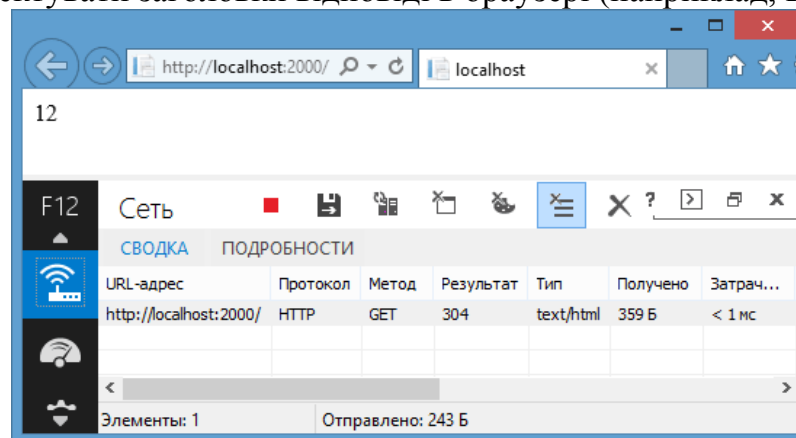
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.Mvc;

namespace CachingApp.Controllers
{
    public class HomeController : Controller
    {
        static int x = 9;

        [OutputCache(Duration=30, Location= OutputCacheLocation.Downstream)]
        public string Index()
        {
            x++;
            return x.ToString();
        }
    }
}
```

Для простоти прикладу в даному випадку виводиться в браузері значення статичної змінної. При звичайній роботі без кешування при кожному зверненні до даного методу значення цієї змінної збільшувалася б на одиницю. Але тут використовується атрибут `OutputCache`, який задає, по-перше, час кешування – 30 секунд, по-друге, місце кеша - клієнт і можливе проксі-сервери, через які може йти запит.

В результаті незважаючи на нові звернення до ресурсу в межах 30 секунд ми будемо отримувати одне і те ж значення змінної `x`, тому що її значення буде братися з кеша, і тільки через 30 секунд ця змінна може інкрементуватись. Ми можемо проінспектувати заголовки відповіді в браузері (наприклад, в IE):

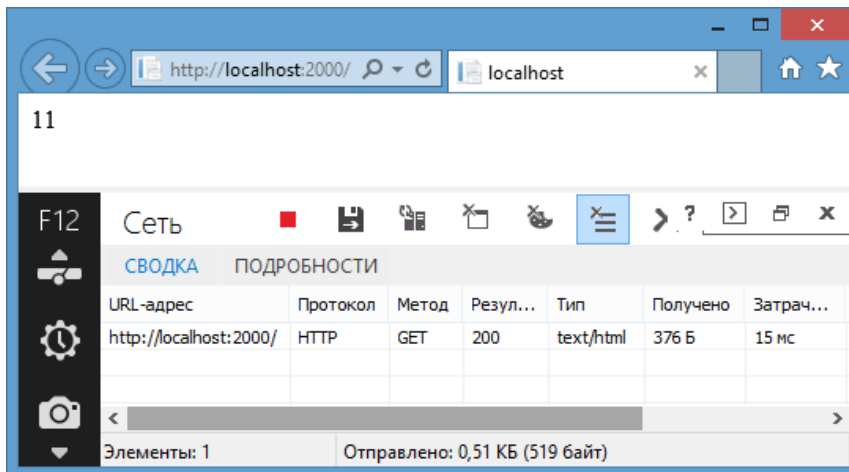


При отриманні кешованих даних ми отримаємо статусний код 304, а витрачений на запит час буде дуже невеликим (в даному випадку не більше 1

мс), так як дані будуть братися з кеша браузера. Подібним чином ми можемо обмежитися кешуванням на сервері:

```
public class HomeController : Controller
{
    static int x = 9;
    [OutputCache(Duration=30, Location= OutputCacheLocation.Server)]
    public string Index()
    {
        x++;
        return x.ToString();
    }
}
```

Той же метод, тільки кешування обмежена кешем сервера. На відміну від попереднього прикладу тут вже запит йде від браузера до сервера, але після кешування протягом наступних 30 секунд при повторних зверненнях до методу даний метод не буде спрацьовувати: він просто буде віддавати клієнту дані, які є в кеші:



При повторних зверненнях знову ж часу досить мало - в даному випадку всього 15 мілісекунд. Хоча часу виконання даного методу дуже мало в силу його простоти - простий інкремент змінної, але в реальності методи, природно, будуть набагато складніші, з вибіркою з бази даних, зверненням до різних ресурсів і складними обчисленнями. І тут кешування дійсно зможе реально оптимізувати роботу сервера.

4.3.2 Параметри кешування

Використовуючи параметри `VaryByHeader`, `VaryByCustom` і `VaryByParam` атрибута `OutputCache`, ми можемо керувати версіями кеша і в залежності від умов надавати одну або іншу версію кешованих даних. Застосування варіантів кешування підвищує гнучкість програми.

Диференціація кеша по заголовкам

При отриманні запиту ми отримуємо різні заголовки і можемо визначити вид заголовка, за яким буде йти диференціація кеша. Наприклад:

```
public class HomeController : Controller
{
    static int x = 9;
    [OutputCache(Duration=30, Location= OutputCacheLocation.Any, VaryByHeader="user-agent")]
    public string Index()
    {
        x++;
        return x.ToString();
    }
}
```

І якщо ми звернемося до додатка одночасно з різних браузерів, які будуть посилати різні заголовки для параметра user-agent, то ми побачимо різні результати.

Диференціація кеша за параметрами запиту

Параметр VaryByParam задає набір параметрів, для яких будуть генеруватися різні версії кешованих даних:

```
public class HomeController : Controller
{
    static int x = 9;
    [OutputCache(Duration=30, Location= OutputCacheLocation.Any, VaryByParam="title;id")]
    public string Index(string title, int? id)
    {
        x++;
        return x.ToString();
    }
}
```

Якщо ми хочемо генерувати різні версії кешу для декількох параметрів, то ці параметри вказуються через крапку з комою. Також можна вказати за допомогою знака *, щоб генерувалися версії кешу для всіх можливих наборів параметрів. В даному випадку для різних значень параметрів title і id будуть створюватися різні версії кешованих даних.

І якщо в двох різних браузерах ми, наприклад, наберемо в адресному рядку `http://localhost:2000/Home/Index/title=mvc`, то вони відобразять одне і те ж значення з кешу.

Диференціація кеша по призначених для користувача параметрах

За допомогою властивості VaryByCustom ми можемо самі визначити значення, за якими будуть генеруватися різні версії кешу. Для цього перейдемо до файлу Global.asax і перевизначимо в класі MvcApplication метод GetVaryByCustomString:

```

public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        AreaRegistration.RegisterAllAreas();
        FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
        RouteConfig.RegisterRoutes(RouteTable.Routes);
        BundleConfig.RegisterBundles(BundleTable.Bundles);
    }

    public override string GetVaryByCustomString(HttpContext context, string custom)
    {
        if (custom == "browser")
        {
            return context.Request.Browser.Browser;
        }
        else
        {
            return base.GetVaryByCustomString(context, custom);
        }
    }
}

```

Метод простий: якщо переданий параметр має значення "browser", то повертається назва браузера, якщо немає, то йде звернення до виклику методу з батьківського класу.

Тепер встановимо відповідним чином атрибут OutputCache:

```

public class HomeController : Controller
{
    static int x = 9;
    [OutputCache(Duration=30, Location=OutputCacheLocation.Any, VaryByCustom="browser")]
    public string Index(string title, int? id)
    {
        x++;
        return x.ToString();
    }
}

```

Значення атрибута якраз і представляє те значення, яке в якості параметра буде передаватися в метод GetVaryByCustomString(). В результаті при зверненні до ресурсу з різних браузерів ми будемо отримувати різні версії кешу.

4.3.3 Профілі кешування

Альтернативу визначення параметрів в атрибуті OutputCache представляє використання профілів кешування. Визначимо свій профіль. Для цього перейдемо до файлу конфігурації web.config і додамо в межах вузла system.web наступний вузол caching:

```
<?xml version="1.0" encoding="utf-8"?>
```

```

<configuration>
  <appSettings>
    <add key="webpages:Version" value="3.0.0.0"/>
    <add key="webpages:Enabled" value="false"/>
    <add key="ClientValidationEnabled" value="true"/>
    <add key="UnobtrusiveJavaScriptEnabled" value="true"/>
  </appSettings>
  <system.web>
    <compilation debug="true" targetFramework="4.5"/>
    <httpRuntime targetFramework="4.5"/>
    <caching>
      <outputCacheSettings>
        <outputCacheProfiles>
          <add name="cacheProfile1" duration="30" location="Any" varyByCustom="browser"/>
        </outputCacheProfiles>
      </outputCacheSettings>
    </caching>
  </system.web>
  <!--решта вмісту файлу-->
</configuration>

```

Вузол `caching` визначає конфігурації профілів кешування. У вкладеному вузлі `outputCacheProfiles` можна задати набір профілів. В даному випадку заданий тільки один профіль, який через атрибути визначає все ті ж параметри кешування, які застосовуються в атрибуті `OutputCache`. І щоб застосувати профіль, змінимо атрибут `OutputCache` наступним чином:

```

public class HomeController : Controller
{
    static int x = 9;
    [OutputCache(CacheProfile="cacheProfile1")]
    public string Index(string title, int? id)
    {
        x++;
        return x.ToString();
    }
}

```

Параметр `CacheProfile` вказує на профіль, який визначений в файлі `web.config`.

4.3.4 Динамічне управління кешем

Крім застосування атрибута `OutputCache` можна управляти кешем динамічно за допомогою властивості `Response.Cache`. Ця властивість представляє об'єкт класу `HttpCachePolicy`, в якому визначені наступні методи і властивості:

- **SetCacheability(policy):** встановлює у відповіді заголовок Cache-Control за допомогою перечислення HttpCacheability. Встановлюючи цей параметр, ми можемо визначити місце зберігання кешованих даних - клієнт або сервер
- **SetExpires(date):** встановлює за допомогою об'єкта DateTime у відповіді заголовок Expires, який визначає термін закінчення кешування
- **SetLastModified(date):** встановлює за допомогою об'єкта DateTime у відповіді заголовок Last-Modified
- **SetMaxAge(period):** встановлює за допомогою об'єкта TimeSpan у заголовка Cache-Control флаг max-age
- **SetNoServerCaching():** Вимикає для даної відповіді кешування на стороні сервера
- **SetNoStore():** додає до заголовку Cache-Control флаг no-store
- **SetNoTransforms():** додає до заголовку Cache-Control флаг no-trans
- **SetProxyMaxAge(period):** встановлює за допомогою об'єкта TimeSpan у заголовка Cache-Control флаг s-maxage
- **SetVaryByCustom(name):** встановлює параметр, в залежності від якого буде вибиратися версія кеш даних. Аналогічно застосування параметра VaryByCustom в атрибуті OutputCache
- **VaryByHeaders:** визначає заголовки, які використовуються для вибору версії кешу
- **VaryByParams:** визначає параметри рядка запиту або форм, які використовуються для вибору версії кешу

Використовуємо деякі з цих методів:

```
public class HomeController : Controller
{
    static int x = 9;

    public string Index()
    {
        Response.Cache.SetExpires(DateTime.Now.AddSeconds(30));
        Response.Cache.SetCacheability(HttpCacheability.Public);

        x++;
        return x.ToString();
    }
}
```

Для кешування в даному випадку не потрібний атрибут OutputCache, так як все робиться через властивість Response.Cache.

Вираз: **Response.Cache.SetExpires(DateTime.Now.AddSeconds(30))** визначає 30-ти секундний період кешування. А вираз:

Response.Cache.SetCacheability(HttpCacheability.Public) встановлює у відповіді для заголовка Cache-Control значення public, яке вказує, що контент

кешується як на клієнті, так і на проксі-сервері. Його дія аналогічна до застосування значення **OutputCacheLocation.Downstream** для параметра Location атрибута OutputCache.

Крім значення **HttpCacheability.Public** перерахування **HttpCacheability** може мати ще кілька значень:

- **NoCache**: встановлює у відповіді для заголовка **Cache-Control** значення no-cache, яке вказує, що контент НЕ буде кешуватися. При використанні атрибута OutputCache аналог - **OutputCacheLocation.None**
- **Private**: встановлює у відповіді для заголовка **Cache-Control** значення private, яке вказує, що контент буде кешуватися тільки на клієнті. Аналог - **OutputCacheLocation.Client**
- **Server**: встановлює у відповіді для заголовка Cache-Control значення no-cache. Контент буде кешуватися тільки в вихідному кеші ASP.NET. Аналог - **OutputCacheLocation.Server**
- **ServerAndNoCache**: об'єднує дію значень **Server** і **NoCache**
- **ServerAndPrivate**: об'єднує дію значень **Server** і **Private** (кешування на клієнті і сервері)

Наприклад, використовуємо тільки вихідний кеш сервера без кешування на клієнті:

```
public string Index()
{
    Response.Cache.SetExpires(DateTime.Now.AddSeconds(30));
    Response.Cache.SetCacheability(HttpCacheability.Server);

    x++;
    return x.ToString();
}
```

4.3.5 Кешування статичного контенту

Крім кешування результатів дій велике значення для оптимізації роботи додатка має кешування статичного контекста. В даному випадку під статичним контентом маються на увазі файли зображень, скрипти javascript, файли стилів css, якісь інші файли, наприклад, відео або аудіо-файли, які використовуються на сторінці.






При використанні зображень можна їх додатково оптимізувати за допомогою різних сервісів, наприклад, tinypng.org, або скласти спрайт, що зменшить обсяг даних при завантаженні і знизить кількість запитів до сервера.

При роботі зі скриптами javascript і стилями css їх можна мінімізувати за допомогою спеціальних мініфікаторів, наприклад, Microsoft Ajax Minifier. Крім того, з стилів і скриптів можна скласти бандли, що також оптимізує продуктивність і зменшує кількість запитів до сервера.






Хоча ці методи і оптимізують продуктивність, але їх також може не вистачити, особливо коли ми маємо справу з високонавантажених додатками. До

того ж ще може бути купа інших типів статичних файлів. У цьому випадку ми можемо ще більше збільшити їх продуктивність, додавши кешування.

Наприклад, в додатку використовується стилі, скрипти і зображення. При першому зверненні до ресурсу сервер відправить всі необхідні файли разом з заголовком OK 200:

<div> ⏏ 🔍 📄 <input type="checkbox"/> Preserve log <input checked="" type="checkbox"/> Disable cache </div>							
Name Path	Method	Status Text	Type	Initiator	Size Content	Time Latency	
 modernizr-2.6.2.js /Scripts	GET	200 OK	application/j...	localhost:10 Parser	50.7 KB 50.3 KB	45 ms 43 ms	
 150541.jpg /Content/img	GET	200 OK	image/jpeg	localhost:37 Parser	3.4 MB 3.4 MB	709 ms 77 ms	
 160504.jpg /Content/img	GET	200 OK	image/jpeg	localhost:38 Parser	1.9 MB 1.9 MB	506 ms 75 ms	
 183255.jpg /Content/img	GET	200 OK	image/jpeg	localhost:39 Parser	1.5 MB 1.5 MB	454 ms 50 ms	
 jquery-1.10.2.js /Scripts	GET	200 OK	application/j...	localhost:47 Parser	268 KB 268 KB	59 ms 39 ms	

Як правило, браузери зберігають всі завантажені статичні файли і при повторному запиті перевіряють, чи змінився файл на сервері. Якщо сервер віддає заголовок 304 Not Modified, що значить, що файли не змінювалися, браузер використовує раніше збережені файли на локальному комп'ютері. Якщо файли на сервері були змінені, то вони повторно завантажуються на локальний комп'ютер:

 modernizr... /Scripts	GET	304 Not M...	applic...	Index:10 Parser	363 B 50.3 KB	99 ms 97 ms	
 site.css /Content	GET	304 Not M...	text/css	Index:8 Parser	346 B 596 B	40 ms 40 ms	
 150541.jpg /Content/i...	GET	304 Not M...	image...	Index:37 Parser	354 B 3.4 MB	30 ms 30 ms	
 jquery-1.1... /Scripts	GET	304 Not M...	applic...	Index:47 Parser	359 B 268 KB	96 ms 95 ms	
 bootstrap.js /Scripts	GET	304 Not M...	applic...	Index:49 Parser	354 B 57.9 KB	94 ms 93 ms	

При великому навантаженні подібні запити також можуть впливати на роботу сервера, і щоб браузер кешував статичні файли і використовував їх за замовчуванням, нам треба налаштувати файл web.config.

У файлі конфігурації в межах вузла <configuration> додамо новий елемент:

```
<system.webServer>
  <staticContent>
    <clientCache httpExpires="Fri, 08 May 2015 00:00:00 GMT" cacheControlMode="UseExpires" />
  </staticContent>
</system.webServer>
```

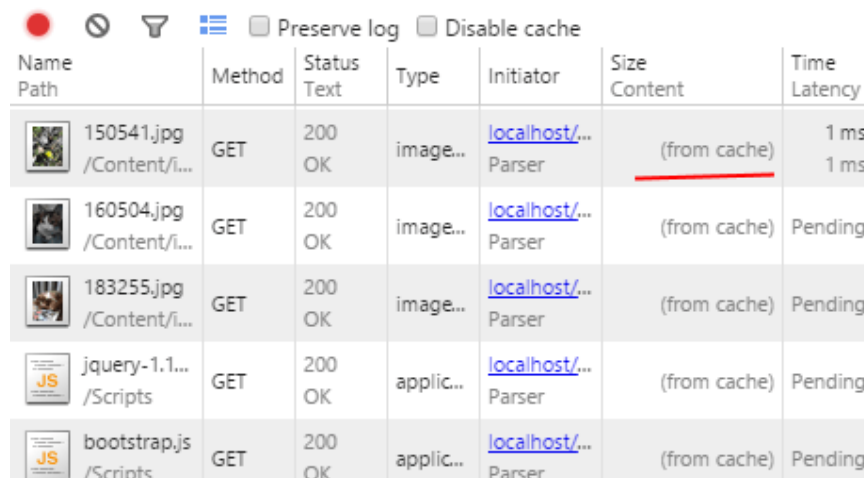
Елемент `staticContent` визначає секцію, що відноситься до статичних файлів. Зокрема, за допомогою елемента `<clientCache>` можна задати параметри кешування на клієнті.






Атрибут `httpExpires` задає максимальну дату, до якої контент в браузері буде кешуватися. В даному випадку це П'ятниця 8 травня 2015 року. При визначенні дати ми повинні дотримуватись певного формату.

Атрибут `cacheControlMode` приймає одне із значень, які вказують на режим кешування:

- `NoControl`: використовується за замовчуванням, не додає у відповідь клієнту ніяких заголовків кешування
- `DisableCache`: додає у відповідь клієнту заголовок `Cache-Control: no-cache` - заборона кешування
- `UseMaxAge`: додає у відповідь клієнту заголовок `Cache-Control: max-age = "часовий відрізок"` - в якості тимчасового відрізка використовується значення атрибута `CacheControlMaxAge`
- `UseExpires`: додає у відповідь клієнту заголовок `Expires`: дата, де в якості дати застосовується значення атрибута **`httpExpires`**

Тепер при повторному зверненні програма не буде посилати серверу запит, чи змінилися файли чи ні, а буде витягувати ці файли з кешу.



Name Path	Method	Status Text	Type	Initiator	Size Content	Time Latency
 150541.jpg /Content/i...	GET	200 OK	image...	localhost/... Parser	(from cache)	1 ms
 160504.jpg /Content/i...	GET	200 OK	image...	localhost/... Parser	(from cache)	Pending
 183255.jpg /Content/i...	GET	200 OK	image...	localhost/... Parser	(from cache)	Pending
 jquery-1.1... /Scripts	GET	200 OK	applic...	localhost/... Parser	(from cache)	Pending
 bootstrap.js /Scripts	GET	200 OK	applic...	localhost/... Parser	(from cache)	Pending

Однак параметр **`httpExpires`** не завжди зручний, особливо коли ми хочемо задати певний час кешування, наприклад, на місяць або на день, незалежно від поточної дати. Для цього змінимо елемент `clientCache` наступним чином:

```
<clientCache cacheControlCustom="public;max-age"
  cacheControlMaxAge="10.00:00:00"
  cacheControlMode="UseMaxAge" />
```

Атрибут **`cacheControlMaxAge`** задає період кешування у вигляді об'єкта **`TimeSpan`**. А параметр **`cacheControlCustom`** вказує на додаткові вказівки, яка буде використовуватися у відповіді клієнту. Знову ж в даному випадку файли будуть братися з кеша без відправки додаткових запитів серверу.

Http-компресія

Застосування стиснення дозволить додатково оптимізувати навантаження на сайт. Для цього в змінімо вишеопределенний елемент `system.webServer`:

```
<system.webServer>
  <urlCompression doDynamicCompression="true"
    doStaticCompression="true"
    dynamicCompressionBeforeCache="true" />

  <staticContent>
    <clientCache cacheControlCustom="public;max-age"
      cacheControlMaxAge="10.00:00:00"
      cacheControlMode="UseMaxAge" />
  </staticContent>
</system.webServer>
```

Елемент `<urlCompression>` дозволяє налаштувати стиснення за допомогою наступних атрибутів:

- **doDynamicCompression**: якщо має значення `true`, то застосовується динамічне стиснення. За замовчуванням `true`
- **doStaticCompression**: якщо має значення `true`, то застосовується статичне стиснення. За замовчуванням `true`
- **dynamicCompressionBeforeCache**: якщо має значення `true`, то динамічне стиснення проводиться до приміщення відповіді в вихідний кеш. За замовчуванням `false`

В результаті стиснення браузер отримає такий заголовок `Content-Encoding: gzip`. Відповідно якщо ми хочемо відключити компресію, то всіх атрибутів треба поставити значення `false`.

4.3.6 MemoryCache

Починаючи з версії .NET 4.0 у фреймворк був доданий клас `MemoryCache`, який призначений для кешування контенту. Даний клас не призначений тільки для створення веб-додатків в ASP.NET, його можна використовувати і при написанні графічних і навіть консольних додатків.

При використанні `MemoryCache` слід враховувати, що об'єкт цього класу прив'язаний до домену додатку, і також треба враховувати, що цей об'єкт буде знищуватися веб-сервером IIS при перезапуску пулу додатків в наступних випадках:

- Якщо протягом 20 хвилин не було запитів до додатка
- Якщо з моменту запуску (останнього перезапуску) додатку пройшло 1740 хвилин - часовий інтервал для перезапуску додатків в IIS за замовчуванням
- Якщо в папку проекту на IIS була додана нова копія додатку (йде автоматичний перезапуск)

Для використання MemoryCache треба додати в проект бібліотеку System.Runtime.Caching.dll. Нехай у нас є деяка модель Phone і клас контексту даних PhoneContext:

```
public class Phone
{
    public int Id { get; set; }
    public string Name { get; set; }
}
public class PhoneContext : DbContext
{
    public DbSet<Phone> Phones { get; set; }
}
```

Додамо новий клас, який буде використовувати функціонал MemoryCache:

```
using System.Runtime.Caching;
//.....
public class AppCache
{
    public Phone GetValue(int id)
    {
        MemoryCache memoryCache = MemoryCache.Default;
        return memoryCache.Get(id.ToString()) as Phone;
    }

    public bool Add(Phone value)
    {
        MemoryCache memoryCache = MemoryCache.Default;
        return memoryCache.Add(value.Id.ToString(), value, DateTime.Now.AddMinutes(10));
    }

    public void Update(Phone value)
    {
        MemoryCache memoryCache = MemoryCache.Default;
        memoryCache.Set(value.Id.ToString(), value, DateTime.Now.AddMinutes(10));
    }

    public void Delete(int id)
    {
        MemoryCache memoryCache = MemoryCache.Default;
        if (memoryCache.Contains(id.ToString()))
        {
            memoryCache.Remove(id.ToString());
        }
    }
}
```

Багато в чому даний клас нагадує стандартний репозиторій. Тепер застосуємо його для отримання об'єкта Phone в методі контролера:

```
public class PhonesController : Controller
```

```

{
    private PhoneContext db = new PhoneContext();
    private AppCache appCache;

    public PhonesController()
    {
        appCache = new AppCache();
    }
    public ActionResult Index(int? id)
    {
        if (id == null)
        {
            return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
        }
        var result = appCache.GetValue(id.Value);
        // якщо нема в кеші
        if(result==null)
        {
            // берем з БД
            result = db.Phones.Find(id);
            // додаємо в кеш
            appCache.Add(result);
        }
        if (result == null)
        { return HttpNotFound(); }
        return View(result);
    }
}

```

Контрольні питання.

1. Призначення механізмів кешування при розробці Web-додатків.
2. Особливості динамічного управління кешем.
3. Особливості кешування статичних даних.
4. Застосування профілів кешування.

4.4 Впровадження залежностей

4.4.1 Інверсія управління:

У невеликих додатках на ASP.NET MVC ми відносно легко можемо замінити один клас на інший, замість одного контексту даних використовувати інший. Однак у великих додатках це вже буде проблематично зробити, особливо якщо у нас десятки контролерів з сотнею методів. У цій ситуації нам на допомогу може прийти такий механізм як впровадження залежностей.

Розглянемо простий приклад. Припустимо, у нас є деяка модель Book:

```

public class Book
{
    public int Id { get; set; }
    public string Name { get; set; }
}

```

І нехай у нас визначений контекст даних і репозиторій для управління об'єктами даної моделі:

```

public class BookContext : DbContext
{
    public BookContext() :base("DefaultConnection"){ }

    public DbSet<Book> Books { get; set; }
}
public class BookRepository : IDisposable
{
    private BookContext db = new BookContext();

    public void Save(Book b)
    {
        db.Books.Add(b);
        db.SaveChanges();
    }
    public IEnumerable<Book> List()
    {
        return db.Books;
    }
    public Book Get(int id)
    {
        return db.Books.Find(id);
    }

    protected void Dispose(bool disposing)
    {
        if (disposing)
        {
            if (db != null)
            {
                db.Dispose();
                db = null;
            }
        }
    }
    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }
}

```

Репозиторій містить посилання на контекст даних і з його допомогою управляє моделями, наприклад, додає в БД і т.д. Тепер розглянемо використання сховища в контролері:

```
public class HomeController : Controller
{
    BookRepository repo;
    public HomeController()
    {
        repo = new BookRepository();
    }
    public ActionResult Index()
    {
        return View(repo.List());
    }
}
```

Контролер містить посилання на BookRepository і в даному випадку він залежить від BookRepository. А сам об'єкт BookRepository є залежністю. Контролер посилається на конкретну реалізацію сховища, тому між ними існує жорсткий зв'язок (tight coupling). На перший погляд нічого поганого тут немає, але, наприклад, у нас змінилася схема підключення до бази даних: замість MS SQL Servera ми вирішили використовувати MongoDB або MySql. Крім того, може знадобитися динамічно змінювати один клас на інший.

В даному випадку жорсткий зв'язок прив'язує контролер до конкретної реалізації сховища. Такий код з розширенням додатку складніше підтримувати і складніше тестувати. Тому рекомендується йти від використання жорсткозв'язаних компонентів до слабозв'язаних. Для цього визначимо новий інтерфейс IRepository і змінимо клас сховища:

```
public interface IRepository
{
    void Save(Book b);
    IEnumerable<Book> List();
    Book Get(int id);
}

public class BookRepository : IDisposable, IRepository
{
    private BookContext db = new BookContext();
    // методи
}
```

Паралельно з цим ми можемо визначити і інший клас сховища, наприклад, для використання MySql: визначимо новий інтерфейс IRepository і змінимо клас сховища:

```
public class MySqlRepository : IDisposable, IRepository
{
```



```
// методи
}
```

І тепер ми можемо позбавити контролер від жорсткої залежності від компонента:

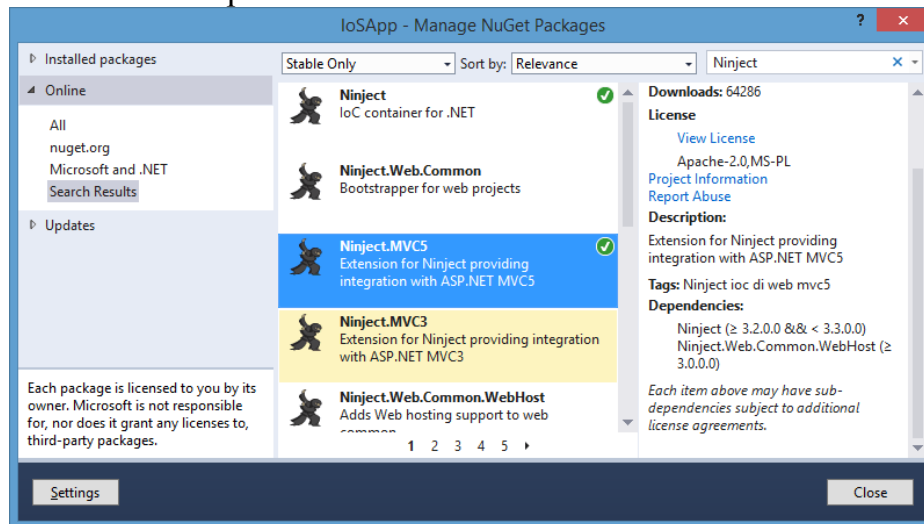
```
public class HomeController : Controller
{
    IRepository repo;
    public HomeController(IRepository r)
    {
        repo = r;
    }
}
```

Тепер контролер нічого не знає і не залежить від конкретної реалізації сховища. Тепер його створення виноситься з класу контролера назовні. Подібний процес називається інверсією управління (inversion of control (IoC)).

4.4.2 IoC-контейнер Ninject

Отже, ми оголосили залежність контролера від об'єкта IRepository. Тепер необхідно впровадити залежність в контролер. Для впровадження залежностей, як правило, використовуються IoC-контейнери. Існує безліч IoC-контейнерів: Castle Windsor, Spring.Net, Autofac, Ninject, Unity. Використовуємо Ninject.

Для цього спочатку всі необхідні бібліотеки в проект. Встановити необхідний пакет можна через NuGet:



Потрібний нам пакет: Ninject.MVC5, разом з яким встановлюється ряд залежних пакетів. Після установки пакета ми вже можемо використовувати Ninject в проекті. Наприклад, змінимо конструктор контролера наступним чином:

```
using Ninject;
//.....
public class HomeController : Controller
{
```

```

IRepository repo;
public HomeController()
{
    IKernel ninjectKernel = new StandardKernel();
    ninjectKernel.Bind<IRepository>().To<BookRepository>();
    repo = ninjectKernel.Get<IRepository>();

}
public ActionResult Index()
{
    return View(repo.List());
}
}

```

Щоб керувати залежностями через Ninject, спочатку треба створити об'єкт Ninject.IKernel за допомогою вбудованої реалізації цього інтерфейсу - класу StandardKernel:

```

IKernel ninjectKernel = new StandardKernel();

```

Далі потрібно встановити відносини між інтерфейсами і їх реалізаціями:

```

ninjectKernel.Bind<IRepository>().To<BookRepository>();

```

Цей вираз показує, що об'єкти IRepository повинні будуть розглядатися як BookRepository. І в кінці створюється об'єкт інтерфейсу через метод Get:

```

repo = ninjectKernel.Get<IRepository>();

```

Оскільки вище ми встановили зіставлення між IRepository і BookRepository, то метод ninjectKernel.Get<IRepository>() буде створювати екземпляр класу BookRepository.

4.4.3 Інтерфейс IDependencyResolver

Для установки залежностей в додатку MVC надає інтерфейс IDependencyResolver. Його реалізація служить обгорткою навколо IoC-контейнера. Створимо в проєкті нову папку Util і помістимо в неї новий клас NinjectDependencyResolver:

```

using IoApp.Models;
using Ninject;
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using System.Web.Mvc;

namespace IoApp.Util
{
    public class NinjectDependencyResolver : IDependencyResolver
    {

```

```

private IKernel kernel;
public NinjectDependencyResolver(IKernel kernelParam)
{
    kernel = kernelParam;
    AddBindings();
}
public object GetService(Type serviceType)
{
    return kernel.TryGet(serviceType);
}
public IEnumerable<object> GetServices(Type serviceType)
{
    return kernel.GetAll(serviceType);
}
private void AddBindings()
{
    kernel.Bind<IRepository>().To<BookRepository>();
}
}
}

```

Клас Ninject DependencyResolver реалізує інтерфейс DependencyResolver, визначаючи ряд методів. Методи GetService і GetServices викликаються фреймворком MVC, коли необхідно створити об'єкт типу, який передається через параметр, для обробки вхідних запитів.

В методі AddBindings власне встановлюється співставлення між інтерфейсом-залежністю і конкретним класом цього інтерфейсу, і в кінці необхідно ініціалізувати зіставлення залежностей відбувалося під час запуску програми. При додаванні пакетів Ninject за замовчуванням в папку App_Start додається файл NinjectWebCommon.cs, який запускається при старті програми. Змінимо його вміст в такий спосіб:

```

[assembly:
WebActivatorEx.PreApplicationStartMethod(typeof(LoSApp.App_Start.NinjectWebCommon),
"Start")]
[assembly:
WebActivatorEx.ApplicationShutdownMethodAttribute(typeof(LoSApp.App_Start.NinjectWebComm
on), "Stop")]

namespace LoSApp.App_Start
{
    using System;
    using System.Web;

    using Microsoft.Web.Infrastructure.DynamicModuleHelper;

    using Ninject;
    using Ninject.Web.Common;
    using System.Web.Mvc;
}

```

```

public static class NinjectWebCommon
{
    private static readonly Bootstrapper bootstrapper = new Bootstrapper();

    public static void Start()
    {
        DynamicModuleUtility.RegisterModule(typeof(OnePerRequestHttpModule));
        DynamicModuleUtility.RegisterModule(typeof(NinjectHttpModule));
        bootstrapper.Initialize(CreateKernel);
    }

    public static void Stop()
    {
        bootstrapper.ShutDown();
    }

    private static IKernel CreateKernel()
    {
        {
            var kernel = new StandardKernel();
            try
            {
                kernel.Bind<Func<IKernel>>().ToMethod(ctx => () => new Bootstrapper().Kernel);
                kernel.Bind<IHttpModule>().To<HttpApplicationInitializationHttpModule>();

                RegisterServices(kernel);
                return kernel;
            }
            catch
            {
                kernel.Dispose();
                throw;
            }
        }
        private static void RegisterServices(IKernel kernel)
        {
            DependencyResolver.SetResolver(new IoApp.Util.NinjectDependencyResolver(kernel));
        }
    }
}

```

Цей клас викликається при запуску програми. У його методі Start() реєструються модулі для обробки і створюється об'єкт Ninject.IKernel за допомогою виклику CreateKernel(). У методі RegisterServices реєструємо раніше створений співставник залежностей.

І тепер змінимо контролер:

```

public class HomeController : Controller
{
    IRepository repo;
    public HomeController(IRepository r)
    {

```

```

        repo = r;
    }
    public ActionResult Index()
    {
        return View(repo.List());
    }
}

```

Незважаючи на те, що в конструктор контролера не передається ніякої конкретної реалізації, але все буде працювати. Розглянемо поетапно, як відбувається в даному випадку впровадження залежностей:

Фреймворк MVC отримує запит і звертається до контролера HomeController.

Фреймворк MVC звертається до класу зіставлення залежностей (в даному випадку клас `NinjectDependencyResolver`), щоб той створив новий об'єкт HomeController, передаючи параметр `Type` в метод `GetService` (в клас `NinjectDependencyResolver`).

Співставник залежностей викликає інфраструктуру `Ninject` для створення нового об'єкта HomeController, передаючи тип створюваного об'єкта в метод `TryGet`. `Ninject` дивиться на конструктор HomeController і бачить, що там використовується залежність від інтерфейсу `IRepository`, для якого він встановлює зіставлення з конкретною реалізацією. `Ninject` створює екземпляр класу `BookRepository` і потім використовує його для створення контролера HomeController.

`Ninject` передає створений об'єкт HomeController співставника залежностей, який, в свою чергу, передає його фреймворку MVC, і далі відбувається обробка запиту. Таким чином, ми уникаємо використання конкретних реалізацій і працюємо з об'єктами на рівні інтерфейсів. А зіставлення інтерфейсів з конкретними реалізаціями перекладається на клас `NinjectDependencyResolver`.

4.4.4 Впровадження залежностей з параметрами

Використовуваний при впровадженні залежностей об'єкт `BookRepository` був досить простим. Трохи ускладнимо його, додавши конструктор з параметром:

```

public class BookRepository : IDisposable, IRepository
{
    private BookContext db;

    public BookRepository(BookContext context)
    {
        db = context;
    }
    // визначення методів
}

```

Тепер при зіставленні типів треба встановлювати аргумент, який передається в конструктор. Для цього використовується метод `WithConstructorArgument`. Змінимо в класі `NinjectDependencyResolver` метод `AddBindings`:

```
private void AddBindings()
{
    kernel.Bind<IRepository>().To<BookRepository>()
        .WithConstructorArgument("context", new BookContext());
}
```

Метод **`WithConstructorArgument`** приймає два параметри: назва параметра конструктора і значення. Подібним чином ми могли б винести контекст даних в репозиторій в окрему властивість:

```
public class BookRepository : IDisposable, IRepository
{
    private BookContext db;
    public BookContext Context
    {
        get { return db; }
        set { db = value; }
    }

    public BookRepository()
    {
    }
    // визначення методів
}
```

Тоді встановити потрібне значення для властивості можна за допомогою методу `WithPropertyValue`:

```
private void AddBindings()
{
    kernel.Bind<IRepository>().To<BookRepository>()
        .WithPropertyValue("Context", new BookContext());
}
```

Метод `With Property Value` також приймає два параметри: назва властивості та значення для нього.

4.4.5 IoC-контейнер Autofac

IoC-контейнер Autofac надає аналогічну функціональність, що і Ninject, пропонуючи альтернативний спосіб впровадження залежностей. Візьмемо той же приклад з репозиторієм:

```
interface IRepository
```

```

{
    // визначення методів
}
public class BookRepository : IDisposable, IRepository
{
    private BookContext db = new BookContext();

    public BookRepository()
    {
    }
    // визначення методів
}

```

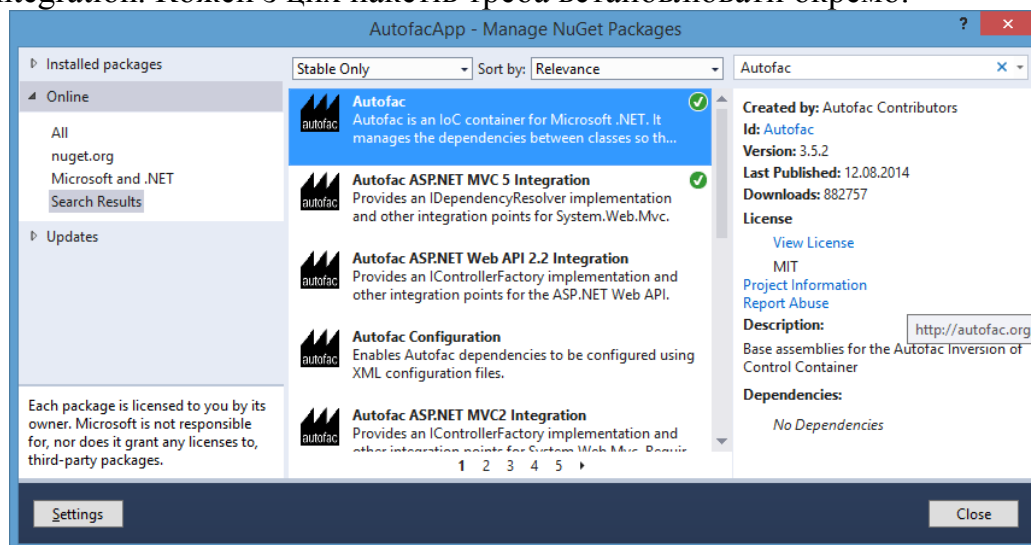
І нехай у нас буде той же контролер:

```

public class HomeController : Controller
{
    IRepository repo;
    public HomeController(IRepository r)
    {
        repo = r;
    }
    public ActionResult Index()
    {
        return View(repo.List());
    }
}

```

Використовуємо для установки залежностей IoC-контейнер Autofac. Для цього додамо в проект через NuGet два пакети: Autofac і Autofac ASP.NET MVC 5 Integration. Кожен з цих пакетів треба встановлювати окремо.



Після додавання Autofac визначимо в проекті, наприклад, в якій-небудь додаткової папці Util новий клас AutofacConfig:

```
using Autofac;
```

```

using Autofac.Core;
using Autofac.Integration.Mvc;
using System.Web.Mvc;

public class AutofacConfig
{
    public static void ConfigureContainer()
    {
        // одержуємо екземпляр контейнера
        var builder = new ContainerBuilder();

        // реєструємо контролер в поточній збірці
        builder.RegisterControllers(typeof(MvcApplication).Assembly);

        // реєструємо співставлення типів
        builder.RegisterType<BookRepository>().As<IRepository>();

        // створюємо новий контейнер з тими залежностями, які визначені вище
        var container = builder.Build();

        // установка співставника залежностей
        DependencyResolver.SetResolver(new AutofacDependencyResolver(container));
    }
}

```

Клас `Autofac Config` визначає один статичний метод `ConfigureContainer ()`, який і виконує всю настройку.

Після цього залишається викликати даний метод в `global.asax`:

```

protected void Application_Start()
{
    AutofacConfig.ConfigureContainer();
    AreaRegistration.RegisterAllAreas();
    FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
    RouteConfig.RegisterRoutes(RouteTable.Routes);
    BundleConfig.RegisterBundles(BundleTable.Bundles);
}

```

Установка параметрів і властивостей

Як і `Ninject`, `Autofac` надає можливість встановлювати параметри і властивості у залежностей. Наприклад, клас сховища приймає параметр:

```

public class BookRepository : IDisposable, IRepository
{
    private BookContext db;
    public BookRepository(BookContext context)
    {
        db = context;
    }
}

```


Для установки параметрів використовується метод `With Parameter`:

```
builder.RegisterType<BookRepository>()  
    .As<IRepository>()  
    .WithParameter("context", new BookContext());
```

Першим параметром в метод передається назва параметра, а другим - значення для нього. Якщо у нас використовується кілька параметрів (наприклад, `public BookRepository (BookContext context, string connectionStr)`), то застосовується метод `WithParameters()`:

```
builder.RegisterType<BookRepository>()  
    .As<IRepository>()  
    .WithParameters(new List<Parameter> { new NamedParameter("context", new BookContext()),  
        new NamedParameter("connectionStr", "fvdsfffgf") });
```

Для установки залежностей у властивостях застосовуються методи `WithProperty/WithProperties`, дія яких аналогічно. Наприклад, в репозиторії є властивість:

```
public class BookRepository : IDisposable, IRepository  
{ private BookContext db;  
    public BookContext Context  
    {  
        get { return db; }  
        set { db = value; }  
    }  
}
```

Встановимо значення для цієї властивості:

```
builder.RegisterType<BookRepository>()  
    .As<IRepository>()  
    .WithProperty("Context", new BookContext());
```

4.4.6 IoC-контейнер Castle Windsor

Ще одним поширеним IoC-контейнером є Castle Windsor. Візьмемо звичний приклад з репозиторієм:

```
interface IRepository  
{  
    // визначення методів  
}  
public class BookRepository : IDisposable, IRepository  
{  
    private BookContext db = new BookContext();
```

```

public BookRepository()
{
}
// визначення методів
}

```

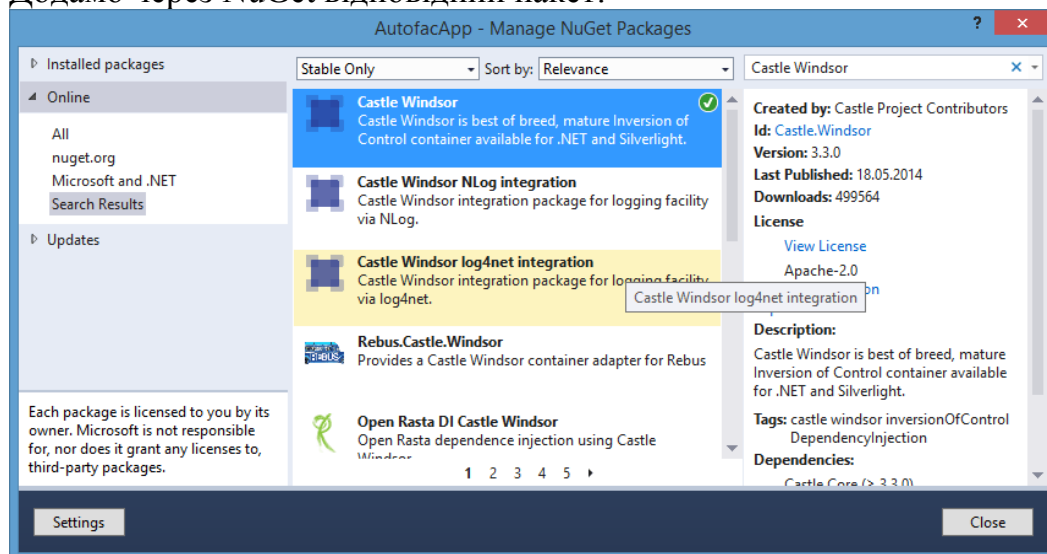
І той же контролер:

```

public class HomeController : Controller
{
    BookRepository repo;
    public HomeController()
    {
        repo = new BookRepository();
    }
    public ActionResult Index()
    {
        return View(repo.List());
    }
}

```

Додамо через NuGet відповідний пакет:



Додаємо в проект в якусь спеціальну папку новий клас CastleControllerFactory:

```

using AutofacApp.Models;
using Castle.MicroKernel.Registration;
using Castle.Windsor;
using System;
using System.Web.Mvc;
using System.Web.Routing;

namespace CastleApp.Util
{
    public class CastleControllerFactory : DefaultControllerFactory
    {

```

```

//контейнер
public IWindsorContainer Container { get; protected set; }

public CastleControllerFactory(IWindsorContainer container)
{
    if (container == null)
    {
        throw new ArgumentNullException("container");
    }

    this.Container = container;
}

//одержання контролера для обробки запиту
protected override IController GetControllerInstance(RequestContext requestContext, Type
controllerType)
{
    if (controllerType == null)
    {
        return null;
    }
    // одержуємо запрошений контролер від Castle
    return Container.Resolve(controllerType) as IController;
}
// одержуємо контроллер
public override void ReleaseController(IController controller)
{
    var disposableController = controller as IDisposable;
    if (disposableController != null)
    {
        disposableController.Dispose();
    }
    // інформуємо іос-контейнер, що контроллер нам більше не потрібний
    Container.Release(controller);
}
}
}

```

Клас Castle Controller Factory представляє реалізацію фабрики контролерів. Його мета - встановлення контейнера Castle Windsor і отримання контролера для обробки запиту. Також додамо новий клас ApplicationCastleInstaller:

```

using CastleApp.Models;
using Castle.MicroKernel.Registration;
using Castle.MicroKernel.SubSystems.Configuration;
using Castle.Windsor;
using System.Linq;
using System.Reflection;
using System.Web.Mvc;

namespace CastleApp.Util

```

```

{
    public class ApplicationCastleInstaller : IWindsorInstaller
    {
        public void Install(IWindsorContainer container, IConfigurationStore store)
        {
            // реєструємо компоненти програми
            container.Register(Component.For<IRepository>().ImplementedBy<BookRepository>());
            // реєструємо кожний контролер по окремістї
            var controllers = Assembly.GetExecutingAssembly()
                .GetTypes().Where(x => x.BaseType == typeof(Controller)).ToList();
            foreach (var controller in controllers)
            {
                container.Register(Component.For(controller).LifestylePerWebRequest());
            }
        }
    }
}

```

Даний клас реалізує інтерфейс IWindsorInstaller, що надається IoC-контейнером і реєструє залежності компонентів. В даному випадку реєструється зіставлення об'єктів IRepository з типом BookRepository, а також реєструються всі контролери, які входять в збірку. І щоб впровадження залежностей запрацювало, змінимо файл Global.asax:

```

protected void Application_Start()
{
    // створюємо контейнер
    var container = new WindsorContainer();
    // реєструємо компоненти з допомогою об'єкта ApplicationCastleInstaller
    container.Install(new ApplicationCastleInstaller());

    // викликаємо свою фабрику контролерів
    var castleControllerFactory = new CastleControllerFactory(container);

    // додаємо фабрику контролерів для обробки запитів
    ControllerBuilder.Current.SetControllerFactory(castleControllerFactory);

    AreaRegistration.RegisterAllAreas();
    FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
    RouteConfig.RegisterRoutes(RouteTable.Routes);
    BundleConfig.RegisterBundles(BundleTable.Bundles);
}

```

Суть проведених змін полягає в створенні контейнера і установки фабрики контролерів для обробки запитів.

Установка параметрів і властивостей

Якщо залежність у вигляді сховища приймає в конструкторі параметр:

```

public class BookRepository : IDisposable, IRepository

```

```

{
    private BookContext db;
    public BookRepository(BookContext context)
    {
        db = context;
    }
}

```

То ми можемо встановити значення для цього параметра в такий спосіб:

```

container.Register(Component.For<IRepository>().ImplementedBy<BookRepository>()
    .DynamicParameters((r, k) =>{ k["context"] = new BookContext(); }));

```

Метод Dynamic Parameters приймає делегат, через який встановлюється значення параметра наступним чином: k["назва_параметра"]=значення

Установка залежностей у властивостях буде трохи складнішою. По-перше, змінимо інтерфейс сховища, щоб він приймав контекст у вигляді властивості:

```

public interface IRepository
{
    BookContext Context { get; set; }
    void Save(Book b);
    IEnumerable<Book> List();
    Book Get(int id);
}
public class BookRepository : IDisposable, IRepository
{
    private BookContext db;
    public BookContext Context
    {
        get { return db; }
        set { db = value; }
    }
    // решту методів
}

```

Встановимо значення для цієї властивості:

```

public class ApplicationCastleInstaller : IWindsorInstaller
{
    public void Install(IWindsorContainer container, IConfigurationStore store)
    {
        container.Register(Component.For<IRepository>().ImplementedBy<BookRepository>());
        // одержуємо об'єкт IRepository
        var repo = container.Resolve<IRepository>();
        // встановлюємо в нього властивість
        repo.Context = new BookContext();

        var controllers = Assembly.GetExecutingAssembly().GetTypes()

```

```

        .Where(x => x.BaseType == typeof(Controller)).ToList();
foreach (var controller in controllers)
{
    container.Register(Component.For(controller).LifestylePerWebRequest());
}
}
}

```

Щоб встановити властивість, нам треба створити потрібний об'єкт `var repo=container.Resolve<IRepository>()`; Потім ми можемо звертатися до нього так само, як в звичайній програмі.

Контрольні питання.

1. Призначення механізмів впровадження залежностей.
2. Особливості залежностей з параметрами.
3. Використання IoC-контейнерів.

4.5 URL Rewriting

4.5.1 Вступ в URL Rewriting

Крім визначення маршрутів в додатку ASP.NET MVC5 нам може знадобитися такий механізм як URL Rewriting або перевизначення URL-адрес, які використовуються для доступу до ресурсів додатку. Наприклад, додаток використовував один домен, потім переїхав на інший - треба зробити редирект. Або стандартна проблема: перенаправлення з домену `www` на домен без `www` і навпаки. Для цього робиться перевизначення URL-адрес.

За перевизначення URL відповідає спеціальний модуль URL Rewrite Module. Саме перевизначення проводиться в файлі конфігурації `web.config` в елементі `<system.webServer>`, який визначається всередині елемента `<configuration>`. Наприклад, визначимо автоматичне перенаправлення з адрес без `www` на адреси без `www`:

```

<?xml version="1.0" encoding="utf-8"?>
<configuration>
<!--решту вмісту-->
<system.webServer>
    <rewrite>
        <rules>
            <rule name="Remove WWW" patternSyntax="Wildcard" stopProcessing="true">
                <match url="*" />
                <conditions>
                    <add input="{CACHE_URL}" pattern="*://www.*" />
                </conditions>
                <action type="Redirect" url="{C:1}://{C:2}" redirectType="Permanent" />
            </rule>
        </rules>
    </rewrite>

```

```
</system.webServer>  
<!-->  
</configuration>
```

Всі правила перевизначення URL визначаються в елементі `<rewrite>`. У ньому ми можемо використовувати два типи наборів правил:

- `<GlobalRules>`: глобальні правила, які задаються на стороні сервера і по відношенню до всього сервера. Вони визначаються у файлі `ApplicationHost.config` і не можуть бути перевизначені або відключені на більш низькому рівні (на рівні окремих сайтів і веб-додатків). Ці правила спрацьовують під час події `PreBeginRequest` на етапі обробки запиту.
- `<Rules>`: це ті правила, які задаються на більш низьких рівнях конфігураційної ієрархії, зокрема, на рівні додатку в файлі `web.config` і які були використані в прикладі вище. Ці правила спрацьовують під час події `BeginRequest`

Кожне правило, представлене елементом `<rule>`, складається з трьох частин:

- `Pattern` - вираз, яким повинна відповідати рядок запиту і яке задається в елементі `<match>`
- `Conditions` - різні додаткові умови, яким повинен відповідати URL-адресу. Наприклад, значення HTTP-заголовків, шляхи до файлів і т.д.
- `Action` - дія, яка повинна виконуватися, якщо рядок URL відповідає регулярному виразу в `Pattern` і умов `Conditions`

Ми можемо використовувати кілька правил одночасно, але їх виконання не завжди обов'язкове. Тому у кожного елемента `rule` визначено атрибут `StopProcessing`. Якщо він має значення `true`, то після виконання дії в елементі `<action>` адреса URL, створювана цим правилом, передається в конвеєр IIS, а інші правила не будуть оброблятися.

Перевизначення URL має такий порядок дій:

Рядок запиту порівнюється з виразом в елементі `match`. Якщо виявиться, що запитувана адреса не відповідає виразу, то модуль `URL Rewrite Module` припиняє обробляти поточне правило і переходить до наступного (якщо задано кілька правил). Якщо рядок запиту відповідає виразу в елементі `match` і при цьому не задано жодних додаткових умов за допомогою елемента `<conditions>`, то `URL Rewrite Module` виконує дію, яке визначено в правилі за допомогою елемента `<action>`.

Якщо рядок запиту відповідає виразу в елементі `match` і також визначені додаткові умови, то `URL Rewrite Module` перевіряє ці умови. І якщо URL відповідає цим умовам, то виконується дія `action`.

Pattern

За допомогою атрибута `url` визначається вираз, з яким порівнюється рядок

запиту. Тип синтаксису цього виразу задається атрибутом **patternSyntax** елемента **rule**, який може набувати таких значень:

- **ECMAScript**: стандарт синтаксису регулярних виразів ECMAScript. Якщо ми не використовуємо атрибут **patternSyntax**, то дане значення використовується для правила за замовчуванням.
- **Wildcard**: для пошуку відповідності використовуються метасимволи

Крім атрибута **url** елемент **match** має ще ряд атрибутів:

- **negate**: якщо має значення **true**, то правило застосовується, якщо URL НЕ відповідає виразу в атрибуті **url** - тобто зворотня дія. За замовчуванням має значення **false**.
- **ignoreCase**: якщо має значення **false**, то правило враховує регістр рядки URL. За замовчуванням має значення **true**, тобто регістр не враховується.

В вище використаному прикладі як **Pattern** використовувався наступний елемент:

```
<rule name="Remove WWW" patternSyntax="Wildcard" stopProcessing="true">  
  <match url="*" />
```

Зірочка означає, що рядок запиту може складатися з будь-яких символів.

Визначення умов

Умови, що задаються елементом **<conditions>**, визначають додаткову логіку оцінки URL на відповідність правилу. Правило може і не містити елемента **<conditions>**, якщо він не потрібен. У цьому елементі визначається атрибут **logicalGrouping**, який може мати одне з двох значень:

- **MatchAll**: при такому значенні для застосування правила до URL повинні бути дотримані всі умови
- **MatchAny**: для застосування правила до URL має бути дотримана хоча б одна умова

Кожна окрема умова задається за допомогою елемента **<add>** і налаштовується за допомогою наступних атрибутів:

- **input**: визначає об'єкт, який буде використовуватися умовою для оцінки. Зокрема, в прикладі вище використовується **input="{CACHE_URL}"**, де **"CACHE_URL"** представляє змінну сервера, що зберігає адресу **url**. Тут також можуть використовуватися і інші змінні сервера.
- **pattern**: визначає регулярний вираз, якому повинен відповідати об'єкт
- **matchType**: набуває таких значень:
 - *Pattern*: в цьому випадку об'єкт (в даному випадку адреса URL) зіставляється з виразом в атрибуті **pattern**. При інших значеннях атрибут **pattern** не враховується

- *IsFile*: визначає, чи є об'єкт (адреса URL) файлом в файлової системі
- *IsDirectory*: визначає, чи є об'єкт (адреса URL) каталогом в файлової системі
- *ignoreCase*: вказує, чи треба ігнорувати регістр адреси URL. За замовчуванням дорівнює true, тому регістр не враховується
- *negate*: якщо дорівнює true, то правило застосовується, якщо умова НЕ враховується. За замовчуванням дорівнює false

Визначення дій

Якщо вираз і умови, які визначаються правилом, відповідають об'єкту (наприклад, адресу URL), то виконується певна дія, задана елементом <action>. Дії можуть бути декількох типів. Тип задається за допомогою атрибута type, який набуває таких значень:

- Rewrite: замінює поточний рядок запиту URL інший рядком
- Redirect: виконує редирект, посилаючи клієнтові статусний код 3xx.
- CustomResponse: відправляє клієнту певний статусний код, а також може відправляти специфічне повідомлення
- AbortRequest: скидає підключення для поточного клієнта

Інші атрибути елемента action:

- url: рядок, який буде замінювати поточний рядок запиту URL
- appendQueryString: визначає, чи повинна зберігатися та частина рядка запиту, яка йде після назви домену та порту. За замовчуванням має значення true, що означає, що рядок запиту з усіма параметрами за винятком назви домену буде зберігатися.
- redirectType: статусний код переадресації при використанні типу Redirect (301 - Permanent, 302 - Found, 303 - See other, 307 - Temporary)
- statusCode: визначає статусний код в якості відповіді клієнту при використанні типу CustomResponse
- subStatusCode: визначає допоміжний статусний код при використанні типу CustomResponse
- statusReason: визначає повідомлення, що відправляється клієнтові разом зі статусним кодом при використанні типу CustomResponse
- statusDescription: визначає повідомлення, що відправляється клієнтові в тілі відповіді при використанні типу CustomResponse
- statusCode: статусний код в якості відповіді клієнту при використанні типу CustomResponse

У нашому випадку відбувається переадресація:

```
<conditions>
  <add input="{CACHE_URL}" pattern="*://www.*" />
</conditions>
<action type="Redirect" url="{C:1}://{C:2}" redirectType="Permanent" />
```

При створенні умови в атрибуті `pattern` використовуються дві зірочки, які інкапсулюють два сегмента до і після "www". Ці два сегменти передаються в дію {C: 1} і {C: 2}. З їх допомогою ми можемо побудувати вихідну адресу URL.

4.5.2 Змінні сервера і зворотні посилання

Використання змінних сервера

При зміні URL ми можемо використовувати такі змінні сервера:

- `QUERY_STRING`: параметри запиту
- `HTTP_HOST`: домен
- `SERVER_PORT`: номер порту
- `SERVER_PORT_SECURE` і `HTTPS`: вказують, чи використовує клієнт захищене підключення
- `REQUEST_URI`: повний рядок запиту

Для IIS URL представляється в наступному вигляді: `http(s)://<host>:<port> /<path>? <Querystring>`. Припустимо, користувач звертається до URL `http://www.somesite.com/home/index?id=2&name=3`. Тоді IIS сегментує її наступним чином:

- `path`: представляє сегмент `home/index`. Ця частина потім порівнюється правилом з виразом, визначеним в елементі `<match>`
- `QUERY_STRING`: в даному випадку сегмент параметрів `id=2 & name=3`
- `HTTP_HOST`: сегмент www.somesite.com
- `SERVER_PORT`: якщо номер порту невідомий, то за замовчуванням дорівнює 80.
- `SERVER_PORT_SECURE` дорівнює 0, а `HTTPS` містить OFF
- `REQUEST_URI`: сегмент `home / index? Id = 2 & name = 3`

При створенні умов для правил ми можемо посилатися на ці змінні через вираз виду `"{НАЗВА_ЗМІННОЇ}"`. Наприклад, нам потрібно умова, згідно з якою в рядку параметрів повинен бути числовий параметр `id`:

```
<add input="{QUERY_STRING}" pattern="id=([0-9]+)" />
```

Крім того, нам доступні заголовки HTTP-запиту, наприклад, рядок юзер-агента ми можемо отримати за допомогою виразу `"{HTTP_USER_AGENT}"`. При використанні заголовків запиту треба враховувати, що всі дефіси в назві заголовків (наприклад, `User-Agent`) замінюються символами підкреслення. Всі малі літери замінюються великими, а до назви змінних додається префікс `"HTTP_"`. Як наприклад, з заголовка `User-Agent` створюється змінна `HTTP_USER_AGENT`.

Зворотні посилання

Зворотні посилання представляють окремі сегменти виразів, які використовуються в умовах, наприклад:

```
<rule name="Remove WWW" patternSyntax="Wildcard" stopProcessing="true">
  <match url="*" />
  <conditions>
    <add input="{CACHE_URL}" pattern="*://www.*" />
  </conditions>
  <action type="Redirect" url="{C:1}://{C:2}" redirectType="Permanent" />
</rule>
```

{C1} і {C2} - це зворотні посилання, які представляють сегменти виразу `pattern = "*/ //www.*"`.

Принцип створення зворотних посилань залежить від типу синтаксису виразу - ECMAScript або Wildcard. Так, в прикладі вище використовувався тип Wildcard, і кожна зірочка в виразі буде створювати з відповідного сегмента зворотнє посилання.

При використанні типу ECMAScript і стандартного синтаксису регулярних виразів, якби у нас було наступна умова -

```
<add input="{CACHE_URL}" pattern="([a-z]+)/([a-z]+)/([0-9]+)" />
```

- то з рядка запити `"home/index/2"` генерувалися б три зворотних посилання: `"home"`, `"index"` і `"2"`. Всі зворотні посилання представляють вирази на кшталт {C:N}, де N - число від 0 до 9. При цьому значення {C: 0} представляє весь рядок, який потрапляє під таку умову.

Крім умов для створення зворотних посилань можуть застосовуватися вирази в елементі match. Всі зворотні посилання з виразу match доступні через висловлювання на кшталт {R:N}, де N - число від 0 до 9. При цьому значення {R:0} представляє весь рядок, який потрапляє під таку умову.

Розглянемо ще один приклад. Припустимо, у нас є правило:

```
<rule name="Rewrite query" stopProcessing="true">
  <match url="^home/index/([0-9]+)/([_0-9a-z-]+)" />
  <action type="Rewrite" url="home/index?id={R:1}&name={R:2}" />
</rule>
```

Наприклад, при запиті `http://www.somesite.com/home/index/2/article` ми отримаємо такі сегменти:

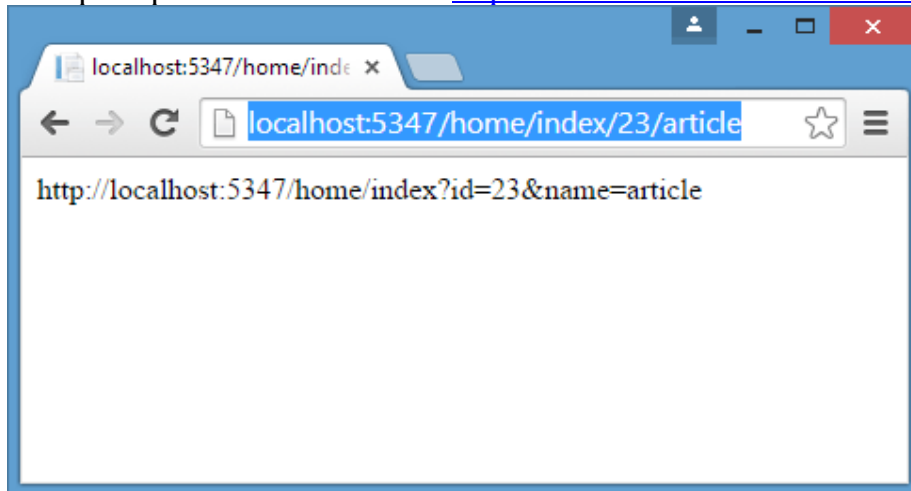
- {R: 0} = "home / index / 2/3"
- {R: 1} = "2"
- {R: 2} = "article"

В результаті буде формуватися наступний рядок URL:
<http://www.somesite.com/home/index?id=2&name=article>

Для тестування визначимо наступний метод, який виводить поточний рядок запиту:

```
public class HomeController : Controller
{
    public ActionResult Index(int id, string name)
    {
        return Content(Request.Url.AbsoluteUri);
    }
}
```

Тепер звернемося за запитом <http://localhost:5347/home/index/23/article>:



4.5.3 Приклади правил URL Rewriting

Розглянемо деякі приклади правил, які можуть часто використовуватися. Переадресація з домена без www на домен з www:

```
<rule name="Enforce WWW" stopProcessing="true">
  <match url="*" />
  <conditions>
    <add input="{CACHE_URL}" pattern="^(.+):(?!http)://(?!www)(.*)" />
  </conditions>
  <action type="Redirect" url="{C:1}://www.{C:2}" redirectType="Permanent" />
</rule>
```

Видаляємо з адреси www:

```
<rule name="Remove WWW" patternSyntax="Wildcard" stopProcessing="true">
  <match url="*" />
  <conditions>
    <add input="{CACHE_URL}" pattern="*://www.*" />
  </conditions>
  <action type="Redirect" url="{C:1}://{C:2}" redirectType="Permanent" />
</rule>
```

Наприклад, ми хочемо перенаправляти з home на home/index, так як по суті ці два url ведуть на один ресурс і тим самим уникнути дублів:

```
<rule name="avoid articles directory" stopProcessing="true">
  <match url="home[/]?$" />
  <action type="Redirect" url="home/index" redirectType="Permanent" />
</rule>
```

Видалення завершаючого рядок запиту слеша (наприклад, http://localhost:1234/home/index/1/)):

```
<rule name="Remove trailing slash" stopProcessing="true">
  <match url="(.*)/$" />
  <conditions>
    <add input="{REQUEST_FILENAME}" matchType="IsFile" negate="true" />
    <add input="{REQUEST_FILENAME}" matchType="IsDirectory" negate="true" />
  </conditions>
  <action type="Redirect" redirectType="Permanent" url="{R:1}" />
</rule>
```

Якщо у нас змінилася адреса, то ми можемо використовувати переадресацію зі старої адреси на нову:

```
<rule name="Canonical Host Name" stopProcessing="true">
  <match url="(.*)" />
  <conditions>
    <add input="{HTTP_HOST}" negate="true" pattern="^mysite\.com$" />
  </conditions>
  <action type="Redirect" url="http://mysite.com/{R:1}" redirectType="Permanent" />
</rule>
```

Запобігання Image Hotlinking - посилок сторонніх сайтів на зображення на нашому сайті:

```
<rule name="Prevent image hotlinking">
  <match url=".*\.(gif|jpg|png)$" />
  <conditions>
    <add input="{HTTP_REFERER}" pattern="^$" negate="true" />
    <add input="{HTTP_REFERER}" pattern="^http://mysite\.com/.*$" negate="true" />
  </conditions>
  <action type="Rewrite" url="/images/say_no_to_hotlinking.jpg" />
</rule>
```

Якщо сайт доступний за протоколом https, то при зверненні до тих же ресурсів сайту, але за протоколом http, сервер буде відправляти помилку 403. Щоб уникнути цього і просто перенаправити користувача на той же запитаний їм ресурс, але тільки по протоколу https, можна використовувати наступне правило:

```
<rule name="Redirect to HTTPS" stopProcessing="true">
  <match url="(.*)" />
  <conditions>
    <add input="{HTTPS}" pattern="^OFF$" />
  </conditions>
  <action type="Redirect" url="https://{HTTP_HOST}/{R:1}" redirectType="Permanent" />
</rule>
```

Переклад рядка запиту в нижній регістр:

```
<rule name="Convert to lower case" stopProcessing="true">
  <match url=".*[A-Z].*" ignoreCase="false" />
  <action type="Redirect" url="{ToLower:{R:0}}" redirectType="Permanent" />
</rule>
```

Блокування запиту по IP:

```
<rule name="Restrict URL" stopProcessing="true">
  <match url="(.*)" />
  <conditions>
    <add input="{REMOTE_ADDR}" pattern="^127.0.0.1$" />
  </conditions>
  <action type="AbortRequest" />
</rule>
```

Контрольні питання.

1. Призначення механізмів адресних перенаправлень.
2. Способи використання серверних змінних.
3. Приклад перенаправлення з www на доменне ім'я.

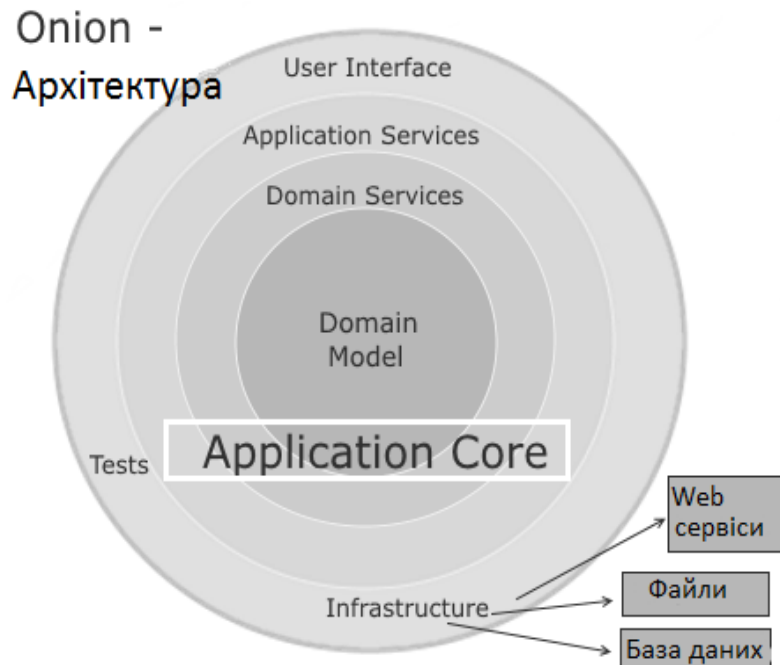
4.6 Архітектура програмного забезпечення

4.6.1 Onion-архітектура.

Термін "Onion Architecture" був запропонований Джеффри Палермо (Jeffrey Palermo) ще в 2008 році. Через роки ця концепція стала досить популярною і є однією з найбільш вживаних типів архітектури при побудові програми на ASP.NET.

Onion-архітектура являє собою поділ програми на рівні. При чому є один незалежний рівень, який знаходиться в центрі архітектури. Від цього рівня залежить другий рівень, від другого - третій і так далі. Тобто виходить, що навколо першого незалежного рівня нашаровується другий-залежний. Навколо другого нашаровується третій, який також може залежати і від першого. Образно це може бути виражено у вигляді цибулини, в якому також є серцевина, навколо якого нашаровуються всі інші шари, аж до лушпиння.

Кількість рівнів може відрізнятися, але в центрі завжди знаходиться модель домену (Domain Model), тобто ті класи моделей, які використовуються в додатку і об'єкти яких зберігаються в базі даних:

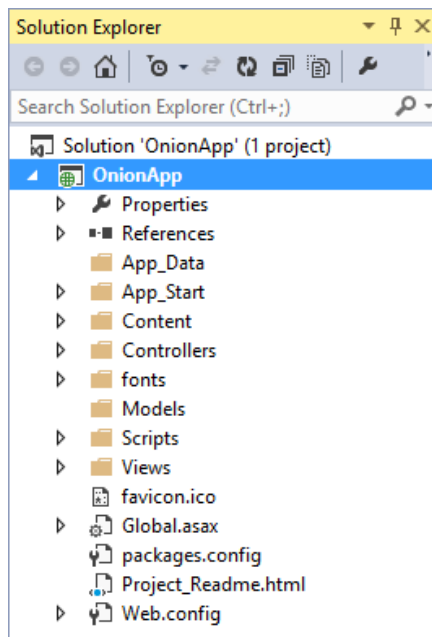


Перший рівень навколо моделі домену утворюють інтерфейси, які керують роботою з моделлю домену. Зазвичай це інтерфейси репозиторіїв, через які ми взаємодіємо з базою даних.

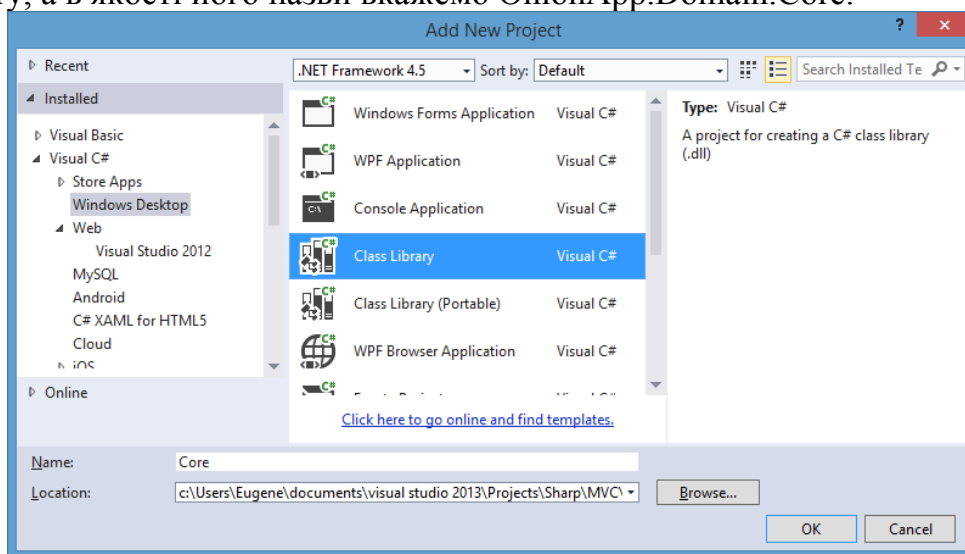
Зовнішній рівень представляє такі компоненти, які дуже часто змінюються. Зазвичай зовнішній рівень утворюють користувальницький інтерфейс, тести, якісь допоміжні класи інфраструктури додатку. До цього рівня також належать конкретні реалізації інтерфейсів, оголошених на нижчих рівнях. Наприклад, реалізація інтерфейсу сховища, який оголошений на рівні Domain Services. Взагалі все внутрішньому рівні, які можна об'єднати в Application Core, визначають тільки інтерфейси, а конкретна реалізація цих інтерфейсів розташовується на зовнішньому рівні.

Також варто відзначити, що всі зовнішні сховища, як бази даних, файли, зовнішні веб-сервіси, від яких ми можемо отримувати дані, - все це є зовнішнім по відношенню до архітектури.

Для більш детального розгляду даного типу архітектури створимо звичайний проект ASP.NET MVC 5, який буде називатися OnionApp.



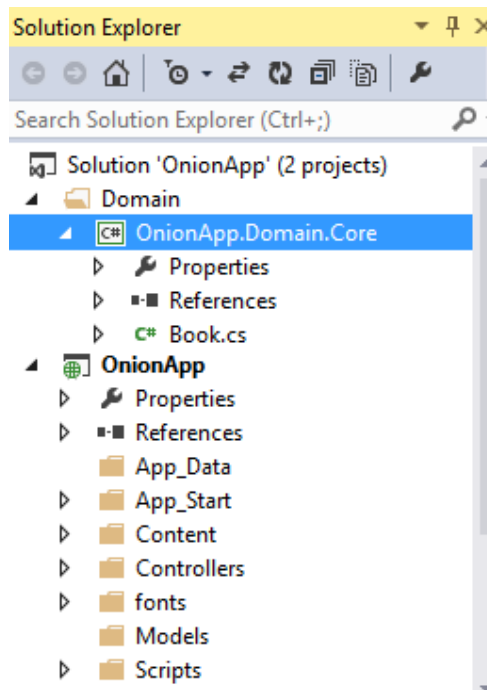
Поки це монолітний додаток, в якому весь код розміщений в одному проєкті. Тепер додамо в солюшн (не в проєкт) нову папку і назвемо її Domain. Потім додамо в папку новий проєкт і в якості типу проєкту виберемо тип Class Library, а в якості його назви вкажемо OnionApp.Domain.Core:



Додамо в новий проєкт клас, що представляє книгу, і це клас буде представляти Domain Model:

```
namespace OnionApp.Domain.Core
{
    public class Book
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public decimal Price { get; set; }
    }
}
```


}

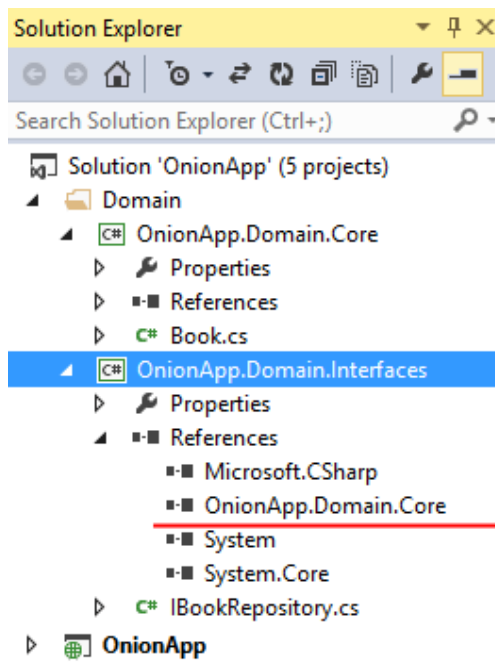


Після цього додамо в папку Domain новий проект також за типом Class Library, а в якості його назви вкажемо OnionApp.Domain.Interfaces. Потім додамо в цей проект посилання на вищевизначений проект OnionApp.Domain.Core і також додамо новий інтерфейс:

```
using System;
using System.Collections.Generic;
using OnionApp.Domain.Core;

namespace OnionApp.Domain.Interfaces
{
    public interface IBookRepository: IDisposable
    {
        IEnumerable<Book> GetBookList();
        Book GetBook(int id);
        void Create(Book item);
        void Update(Book item);
        void Delete(int id);
        void Save();
    }
}
```

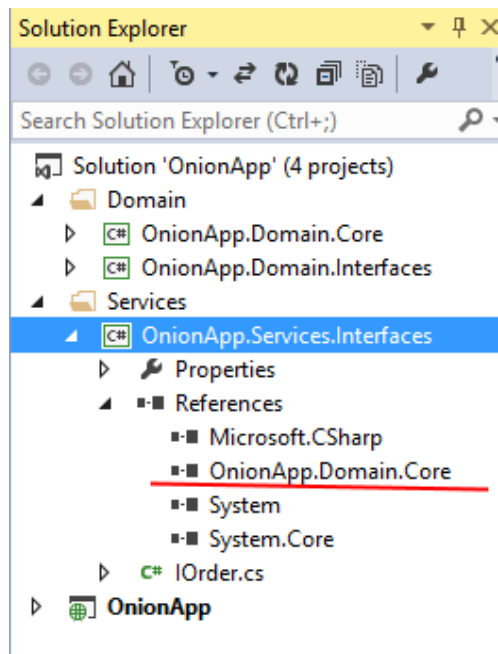
Цей інтерфейс і становить рівень Domain Services та залежить від рівня Domain Model. На даний момент проекти мають виглядати так:



При створенні архітектури додатку треба розуміти, що реальна кількість рівнів тут вельми умовна. Залежно від масштабу завдань рівнів може бути більше, або й менше. Однак важливо розуміти сам принцип, що в центрі у нас моделі домену, а все інше залежить від них. Кожен зовнішній рівень може залежати від внутрішнього, але не навпаки.

На вищепредставленій схемі між зовнішнім рівнем і рівнем Domain Services є ще рівень API або інтерфейсів бізнес-логіки додатка - рівень Application Services. Цей рівень може включати інтерфейси допоміжних класів. Наприклад, покупка книги може являти собою об'єкт, який в залежності від способу оплати (готівкою, через кредитну карту, через електронні гроші) може включати той чи інший функціонал. І, можливо, було б непогано визначити загальний інтерфейс покупки, а в залежності від типу магазину використовувати його конкретну реалізацію. Тому додамо в рішення нову папку, яку назвемо Services. У цю папку додамо новий проект по типу Class Library, який назвемо OnionApp.Services.Interfaces, і додамо в цей проект інтерфейс IOrder:

```
using OnionApp.Domain.Core;
namespace OnionApp.Services.Interfaces
{
    public interface IOrder
    {
        void MakeOrder(Book book);
    }
}
```



Даний проект також має залежність від класів проекту Onion App.Domain.Core. А інтерфейс IOrder, що представляє процес покупки і оформлення замовлення, використовує ці класи в методі MakeOrder(). Передбачається, що в метод передається об'єкт купленої книги.

Тепер перейдемо до створення зовнішнього рівня, який і буде реалізовувати дані інтерфейси. Для цього додамо в рішення папку Infrastructure і потім в неї додамо новий проект типу Class Library, який назовемо OnionApp.Infrastructure.Data.

Даний проект буде реалізовувати інтерфейси, оголошені на нижніх рівнях, і пов'язувати їх з сховищем даних. Як сховище даних буде використовуватися БД MS SQL Server, з якою ми будемо взаємодіяти через Entity Framework. Тому додамо в цей проект через nuGet всі пакети Entity Framework. Також додамо в проект посилання на проекти OnionApp.Domain.Core і OnionApp.Domain.Interfaces, після чого додамо в проект новий клас OrderContext:

```
using System;
using System.Collections.Generic;
using System.Data.Entity;
using OnionApp.Domain.Core;

namespace OnionApp.Infrastructure.Data
{
    public class OrderContext : DbContext
    {
        public DbSet<Book> Books { get; set; }
    }
}
```

Також додамо клас сховища BookDepository:

```

using System;
using System.Collections.Generic;
using System.Linq;
using OnionApp.Domain.Core;
using OnionApp.Domain.Interfaces;
using System.Data.Entity;

namespace OnionApp.Infrastructure.Data
{
    public class BookRepository : IBookRepository
    {
        private OrderContext db;

        public BookRepository()
        {
            this.db = new OrderContext();
        }

        public IEnumerable<Book> GetBookList()
        {
            return db.Books.ToList();
        }

        public Book GetBook(int id)
        {
            return db.Books.Find(id);
        }

        public void Create(Book book)
        {
            db.Books.Add(book);
        }

        public void Update(Book book)
        {
            db.Entry(book).State = EntityState.Modified;
        }

        public void Delete(int id)
        {
            Book book = db.Books.Find(id);
            if (book != null)
                db.Books.Remove(book);
        }

        public void Save()
        {
            db.SaveChanges();
        }

        private bool disposed = false;
    }
}

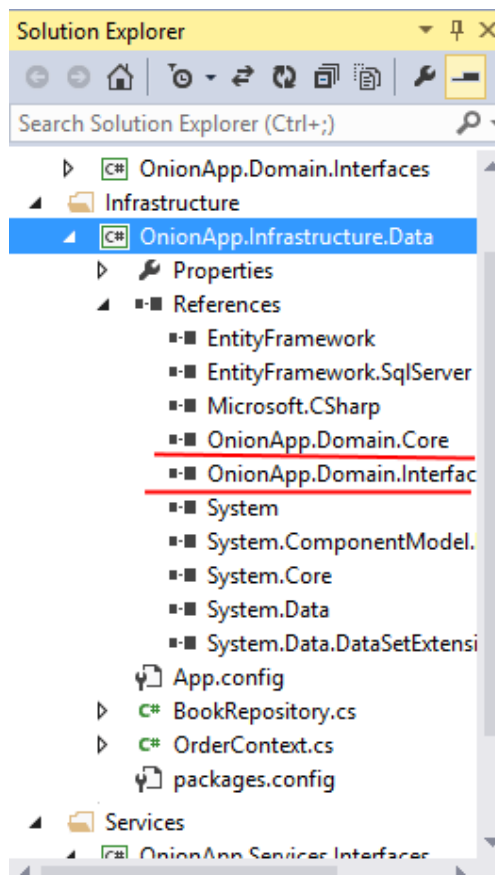
```

```

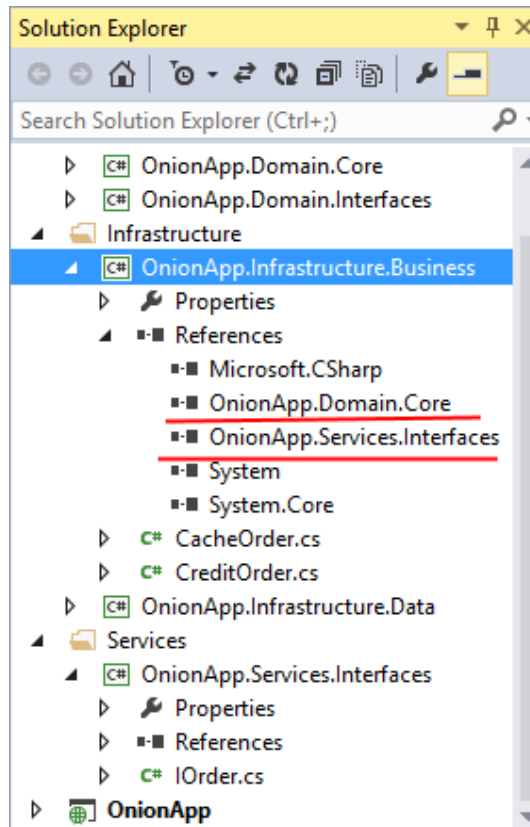
public virtual void Dispose(bool disposing)
{
    if (!this.disposed)
    {
        if (disposing)
        {
            db.Dispose();
        }
    }
    this.disposed = true;
}

public void Dispose()
{
    Dispose(true);
    GC.SuppressFinalize(this);
}
}
}

```



Також у нас є інтерфейс IOrder, і для його реалізації визначимо в папці Infrastructure додатковий проект по типу Class Library, який назовемо OnionApp.Infrastructure.Business. Додамо до нього посилання на проекти OnionApp.Domain.Core і OnionApp.Services.Interfaces:



Тут ми можемо визначити різні реалізації інтерфейсу Order. наприклад:

```
using OnionApp.Domain.Core;
using OnionApp.Services.Interfaces;

namespace OnionApp.Infrastructure.Business
{
    public class CacheOrder : IOrder
    {
        public void MakeOrder(Book book)
        {
            // код покупки книги при оплаті готівкою
        }
    }
}
```

Для простоти прикладу не наводимо код. Подібним чином можна створити також інші можливі реалізації:

```
using OnionApp.Domain.Core;
using OnionApp.Services.Interfaces;
namespace OnionApp.Infrastructure.Business
{
    public class CreditOrder : IOrder
    {
        public void MakeOrder(Book book)
        {
            // код покупки книги з допомогою кредитної карти
        }
    }
}
```

```

    }
}
}

```

Тепер, коли всі необхідні проекти готові, ми можемо перейти до створення веб-інтерфейсу для управління функціоналом. Для цього будемо використовувати наявний за замовчуванням проект OnionApp. У цьому проекті вже є стандартний HomeController. Змінимо його таким чином, щоб він використовував залежності з раніше створених проектів:

```

using System.Web;
using System.Web.Mvc;
using OnionApp.Domain.Core;
using OnionApp.Domain.Interfaces;
using OnionApp.Services.Interfaces;

namespace OnionApp.Controllers
{
    public class HomeController : Controller
    {
        IBookRepository repo;
        IOrder order;
        public HomeController(IBookRepository r, IOrder o)
        {
            repo = r;
            order = o;
        }
        public ActionResult Index()
        {
            var books = repo.GetBookList();
            return View();
        }

        public ActionResult Buy(int id)
        {
            Book book = repo.GetBook(id);
            order.MakeOrder(book);
            return View();
        }
        protected override void Dispose(bool disposing)
        {
            repo.Dispose();
            base.Dispose(disposing);
        }
    }
}

```

Але щоб зв'язати залежності, скористаємося контейнером залежностей Ninject. Додамо в проект OnionApp через NuGet пакет Ninject.MVC5 також. Щоб зареєструвати залежності, створимо в проекті OnionApp каталог Util, в який додамо новий клас NinjectDependencyResolver:

```

using Ninject;
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using System.Web.Mvc;
using OnionApp.Domain.Core;
using OnionApp.Domain.Interfaces;
using OnionApp.Infrastructure.Business;
using OnionApp.Infrastructure.Data;
using OnionApp.Services.Interfaces;

namespace OnionApp.Util
{
    public class NinjectDependencyResolver : IDependencyResolver
    {
        private IKernel kernel;
        public NinjectDependencyResolver(IKernel kernelParam)
        {
            kernel = kernelParam;
            AddBindings();
        }
        public object GetService(Type serviceType)
        {
            return kernel.TryGet(serviceType);
        }
        public IEnumerable<object> GetServices(Type serviceType)
        {
            return kernel.GetAll(serviceType);
        }
        private void AddBindings()
        {
            kernel.Bind<IBookRepository>().To<BookRepository>();
            kernel.Bind<IOrder>().To<CacheOrder>();
        }
    }
}

```

Даний клас пов'язує IBook Repository з BookRepository і IOrder з CacheOrder. Після цього, перейдемо до папки App_Start. Після додавання Ninject в цій папці повинен бути файл NinjectWebCommon.cs. Змінимо в цьому файлі метод RegisterServices():

```

private static void RegisterServices(IKernel kernel)
{
    System.Web.Mvc.DependencyResolver.SetResolver(new
    OnionApp.Util.NinjectDependencyResolver(kernel));
}

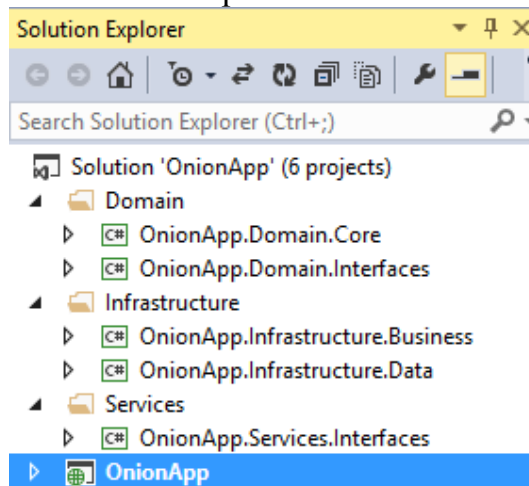
```


І, оскільки наш веб-додаток використовує Entity Framework, то додамо в проект OnionApp пакет Entity Framework і в файлі web.config пропишемо рядок підключення:

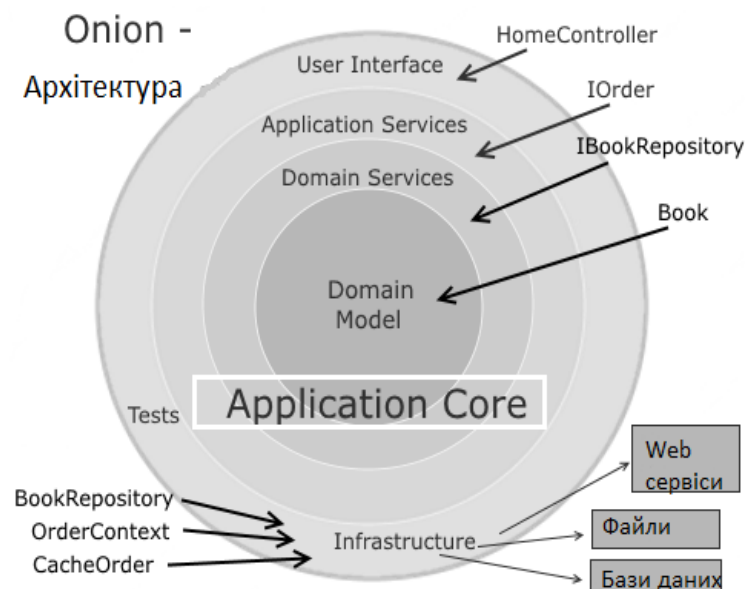
```
<connectionStrings>
  <add name="ObjectContext" providerName="System.Data.SqlClient"
    connectionString="Data
Source=(LocalDB)\v11.0;AttachDbFilename='|DataDirectory|\Bookstore.mdf';Integrated
Security=True" />
</connectionStrings>
```

Тобто незважаючи на те, що клас контексту даних Order Context визначено в проекті OnionApp.Infrastructure.Data, звернення до контексту даних через репозиторій буде відбуватися в проекті OnionApp, тому в ньому і прописуємо рядок підключення.

В результаті у нас вийшло шість проектів:



Таким чином, ми створили додаток з onion-архітектурою, яку схематично можна виразити таким чином:



4.6.2 Патерн Unit of Work

У додатках ASP.NET MVC нерідко використовується патерн репозиторій для інкапсулювання логіки роботи з джерелами даних, і нерідко ми оперуємо безліччю сутностей і моделей, для управління якими створюється безліч класів-репозиторіїв. Патерн Unit of Work дозволяє спростити роботу з різними репозиторіями і дає впевненість, що всі репозиторії використовуватимуть один і той же контекст даних. Розглянемо приклад. Припустимо, у нас є наступна пара моделей:

```
public class Book
{
    public int Id { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
}

public class Order
{
    public int Id { get; set; }
    public string Number { get; set; }
    public int BookId { get; set; }
    public Book Book { get; set; }
}
```

І також існує наступний контекст даних:

```
public class OrderContext : DbContext
{
    public DbSet<Book> Books { get; set; }
    public DbSet<Order> Orders { get; set; }
}
```

Якщо ми будемо використовувати патерн Репозиторій для роботи з базою даних, то ми можемо визначити інтерфейс сховища та створити дві його окремих реалізації:

```
interface IRepository<T> where T : class
{
    IEnumerable<T> GetAll();
    T Get(int id);
    void Create(T item);
    void Update(T item);
    void Delete(int id);
}

public class BookRepository : IRepository<Book>
{
    private OrderContext db;
```

```

public BookRepository(OrderContext context)
{
    this.db = context;
}

public IEnumerable<Book> GetAll()
{
    return db.Books;
}

public Book Get(int id)
{
    return db.Books.Find(id);
}

public void Create(Book book)
{
    db.Books.Add(book);
}

public void Update(Book book)
{
    db.Entry(book).State = EntityState.Modified;
}

public void Delete(int id)
{
    Book book = db.Books.Find(id);
    if (book != null)
        db.Books.Remove(book);
}
}

public class OrderRepository : IRepository<Order>
{
    private OrderContext db;

    public OrderRepository(OrderContext context)
    {
        this.db = context;
    }

    public IEnumerable<Order> GetAll()
    {
        return db.Orders.Include(o=>o.Book);
    }

    public Order Get(int id)
    {
        return db.Orders.Find(id);
    }
}

```

```

public void Create(Order order)
{
    db.Orders.Add(order);
}

public void Update(Order order)
{
    db.Entry(order).State = EntityState.Modified;
}

public void Delete(int id)
{
    Order order = db.Orders.Find(id);
    if (order != null)
        db.Orders.Remove(order);
}
}

```

Для того, щоб використовувати патерн Unit of Work, створимо новий клас:

```

public class UnitOfWork : IDisposable
{
    private OrderContext db = new OrderContext();
    private BookRepository bookRepository;
    private OrderRepository orderRepository;

    public BookRepository Books
    {
        get
        {
            if (bookRepository == null)
                bookRepository = new BookRepository(db);
            return bookRepository;
        }
    }

    public OrderRepository Orders
    {
        get
        {
            if (orderRepository == null)
                orderRepository = new OrderRepository(db);
            return orderRepository;
        }
    }

    public void Save()
    {
        db.SaveChanges();
    }
}

```

```

private bool disposed = false;

public virtual void Dispose(bool disposing)
{
    if (!this.disposed)
    {
        if (disposing)
        {
            db.Dispose();
        }
        this.disposed = true;
    }
}

public void Dispose()
{
    Dispose(true);
    GC.SuppressFinalize(this);
}
}

```

Клас Unit Of Work надає доступ до репозиторіїв через окремі властивості і визначає загальний контекст для обох репозиторіїв. Крім того, даний клас містить додаткові методи Save() і Dispose(), які в іншій ситуації ми мусіли б визначити в репозиторіях. Але так як цей функціонал буде загальним для обох репозиторіїв, то його краще винести в клас UnitOfWork.

У цьому випадку контролер, який здійснює операції над моделлю Book, може виглядати наступним чином:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using UoWMvcApp.Models;

namespace UoWMvcApp.Controllers
{
    public class HomeController : Controller
    {
        UnitOfWork unitOfWork;
        public HomeController()
        {
            unitOfWork = new UnitOfWork();
        }
        public ActionResult Index()
        {
            var books = unitOfWork.Books.GetAll();
            return View();
        }
    }
}

```

```

    }

    public ActionResult Create()
    {
        return View();
    }

    [HttpPost]
    public ActionResult Create(Book b)
    {
        if(ModelState.IsValid)
        {
            unitOfWork.Books.Create(b);
            unitOfWork.Save();
            return RedirectToAction("Index");
        }
        return View(b);
    }

    public ActionResult Edit(int id)
    {
        Book b = unitOfWork.Books.Get(id);
        if (b == null)
            return HttpNotFound();
        return View(b);
    }

    [HttpPost]
    public ActionResult Edit(Book b)
    {
        if (ModelState.IsValid)
        {
            unitOfWork.Books.Update(b);
            unitOfWork.Save();
            return RedirectToAction("Index");
        }
        return View(b);
    }

    public ActionResult Delete(int id)
    {
        unitOfWork.Books.Delete(id);
        unitOfWork.Save();
        return RedirectToAction("Index");
    }

    protected override void Dispose(bool disposing)
    {
        unitOfWork.Dispose();
        base.Dispose(disposing);
    }

```

```
}  
}
```

4.6.3 AutoMapper

Automapper дозволяє проектувати одну модель на іншу, що дозволяє скоротити обсяги коду і спростити програму. Розглянемо на прикладі. Нехай у нас є наступна модель домену User, об'єкти якої зберігаються в базі даних:

```
public class User  
{  
    public int Id { get; set; }  
    public string Name { get; set; }  
    public int Age { get; set; }  
    public string Email { get; set; }  
}
```

Для роботи з моделлю визначені наступні класи контексту даних і сховища:

```
public class UserContext : DbContext  
{  
    public DbSet<User> Users { get; set; }  
}
```

```
public class UserRepository : IRepository<User>  
{  
    UserContext db;  
    public UserRepository()  
    {  
        db = new UserContext();  
    }  
    public IEnumerable<User> GetAll()  
    {  
        return db.Users.ToList();  
    }  
  
    public User Get(int id)  
    {  
        return db.Users.Find(id);  
    }  
  
    public void Create(User item)  
    {  
        db.Users.Add(item);  
    }  
  
    public void Update(User item)  
    {  
        db.Entry(item).State = EntityState.Modified;  
    }  
}
```

```

    }

    public void Delete(int id)
    {
        User user = db.Users.Find(id);
        if (user != null)
            db.Users.Remove(user);
    }

    public void Save()
    {
        db.SaveChanges();
    }

    private bool disposed = false;

    public virtual void Dispose(bool disposing)
    {
        if (!this.disposed)
        {
            if (disposing)
            {
                db.Dispose();
            }
        }
        this.disposed = true;
    }

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }
}

interface IRepository<T> : IDisposable
    where T : class
{
    IEnumerable<T> GetAll();
    T Get(int id);
    void Create(T item);
    void Update(T item);
    void Delete(int id);
    void Save();
}

```

У контролері передбачена базова CRUD-функціональність, і для кожного представлення створені свої моделі view model:

```

public class IndexUserViewModel
{
    public int Id { get; set; }
}

```



```

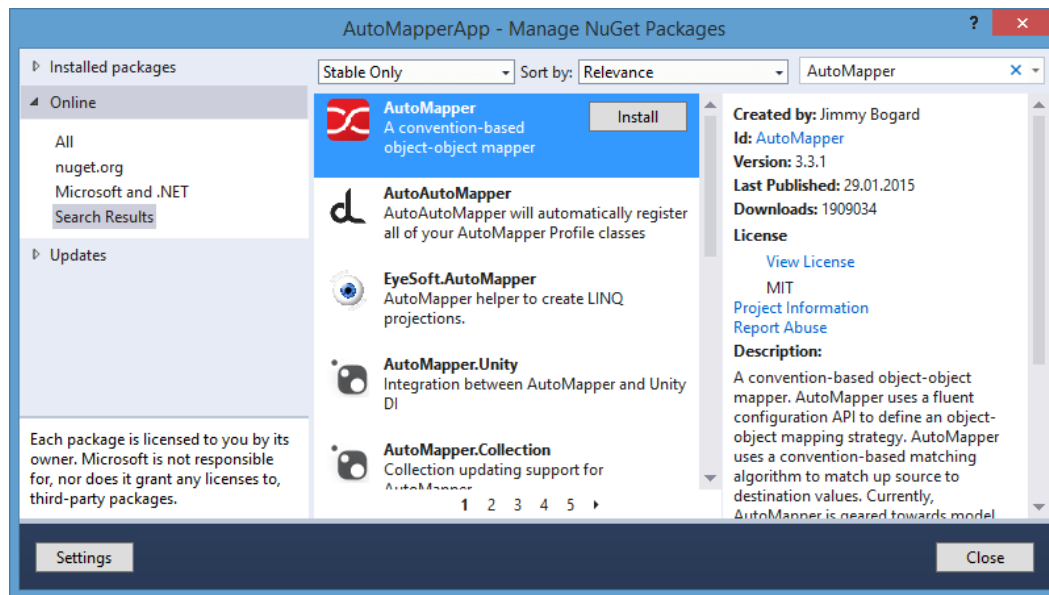
    public string Name { get; set; }
    public int Age { get; set; }
}

public class CreateUserViewModel
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int Age { get; set; }
    public string Login { get; set; }
}

public class EditUserViewModel
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }
    public string Login { get; set; }
}

```

Таким чином, є одна модель домену і три моделі представлення, які відрізняються по набору властивостей. Тепер додамо в проект через NuGet пакет AutoMapper:



Після додавання застосуємо AutoMapper в контролері:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using AutoMapperApp.Models;
using AutoMapper;

```

```

namespace AutoMapperApp.Controllers
{
    public class HomeController : Controller
    {
        IRepository<User> repo;
        public HomeController()
        {
            repo = new UserRepository();
        }
        public ActionResult Index()
        {
            // Налаштування AutoMapper
            Mapper.CreateMap<User, IndexUserViewModel>();
            // сопоставление
            var users =
                Mapper.Map<IEnumerable<User>, List<IndexUserViewModel>>(repo.GetAll());
            return View(users);
        }

        public ActionResult Create()
        {
            return View();
        }
        [HttpPost]
        public ActionResult Create(CreateUserViewModel model)
        {
            if(ModelState.IsValid)
            {
                // Налаштування AutoMapper
                Mapper.CreateMap<CreateUserViewModel, User>()
                    .ForMember("Name", opt => opt.MapFrom(c => c.FirstName + " " + c.LastName))
                    .ForMember("Email", opt => opt.MapFrom(src => src.Login));
                // Виконуємо співставлення
                User user = Mapper.Map<CreateUserViewModel, User>(model);
                repo.Create(user);
                repo.Save();
                return RedirectToAction("Index");
            }
            return View(model);
        }

        public ActionResult Edit(int? id)
        {
            if (id == null)
                return HttpNotFound();
            // Налаштування AutoMapper
            Mapper.CreateMap<User, EditUserViewModel>()
                .ForMember("Login", opt => opt.MapFrom(src => src.Email));
            // Виконуємо співставлення
            EditUserViewModel user = Mapper.Map<User, EditUserViewModel>(repo.Get(id.Value));

```

```

        return View(user);
    }
    [HttpPost]
    public ActionResult Edit(EditUserViewModel model)
    {
        if (ModelState.IsValid)
        {
            // Налаштування AutoMapper
            Mapper.CreateMap<EditUserViewModel, User>()
                .ForMember("Email", opt => opt.MapFrom(src => src.Login));
            // Виконуємо співставлення
            User user = Mapper.Map<EditUserViewModel, User>(model);
            repo.Update(user);
            repo.Save();
            return RedirectToAction("Index");
        }
        return View(model);
    }
}

```

Спочатку створюється об'єкт сховища IRepository. Потім в кожному методі проводиться конфігурація Automapper і зіставлення об'єктів. Для настройки конфігурації використовується метод Mapper.CreateMap():

```
Mapper.CreateMap<User, IndexUserViewModel>()
```

В даному випадку об'єкт User буде зіставлятися з об'єктом IndexUserViewModel. В даному випадку велике значення має порядок: так як в методі Index отримуємо з БД об'єкти User і з них створюємо об'єкти IndexUserViewModel, то першим в

```
Mapper.CreateMap<User, IndexUserViewModel> йде User.
```

При конфігурації зіставляються однойменні властивості User і IndexUserViewModel, однак при створенні, наприклад, набір властивостей моделей CreateUserViewModel і User відрізняється, тому нам треба використовувати додатковий метод ForMember() для проектування окремих властивостей одного класу на властивості іншого класу:

```
Mapper.CreateMap<CreateUserViewModel, User>()
    .ForMember("Name", opt => opt.MapFrom(c => c.FirstName + " " + c.LastName))
    .ForMember("Email", opt => opt.MapFrom(src => src.Login));
```

Те ж саме і з редагуванням. При зіставленні об'єктів використовується метод Mapper.Map():

```
User user = Mapper.Map<EditUserViewModel, User>(model);
```

Причому також можна створювати колекції об'єктів:

```
var users =
    Mapper.Map<IEnumerable<User>, List<IndexUserViewModel>>(repo.GetAll());
```

Таким чином, ми можемо використовувати AutoMapper для проектування одних класів на інші.

У висновку можна відзначити, що всю настройку AutoMapper ми можемо винести в окремий клас, щоб зробити код контролерів простішим. Для цього додамо в папку App_Start новий клас, який назовемо AutoMapperConfig і який буде мати наступний код:

```
using AutoMapper;
using AutoMapperApp.Models;

namespace AutoMapperApp
{
    public class AutoMapperConfig
    {
        public static void RegisterMappings()
        {
            Mapper.CreateMap<User, IndexUserViewModel>();

            Mapper.CreateMap<CreateUserViewModel, User>()
                .ForMember("Name", opt => opt.MapFrom(c => c.FirstName + " " + c.LastName))
                .ForMember("Email", opt => opt.MapFrom(src => src.Login));

            Mapper.CreateMap<User, EditUserViewModel>()
                .ForMember("Login", opt => opt.MapFrom(src => src.Email));

            Mapper.CreateMap<EditUserViewModel, User>()
                .ForMember("Email", opt => opt.MapFrom(src => src.Login));
        }
    }
}
```

Даний клас зроблений за аналогією з іншими класами, що зберігаються в папці App_Start. У методі RegisterMappings встановлюється конфігурація AutoMapper. І щоб застосувати цей клас, нам треба викликати метод RegisterMappings в файлі Global.asax в методі Application_Start:

```
protected void Application_Start()
{
    AutoMapperConfig.RegisterMappings();
    // решту кода
}
```

І після цього відповідні виклики в методах контролера нам вже не потрібні, і ми їх можемо видалити.

4.6.4 Багаторівнева архітектура

У цій темі і ряді наступних матеріалів ми розглянемо трирівневу архітектуру програми ASP.NET MVC і її можливу реалізацію.

Взагалі існує багато різних видів і типів архітектур, які успішно застосовуються. Однієї з найбільш використовуваних є класична трирівнева система, яка передбачає поділ додатка на три рівні.

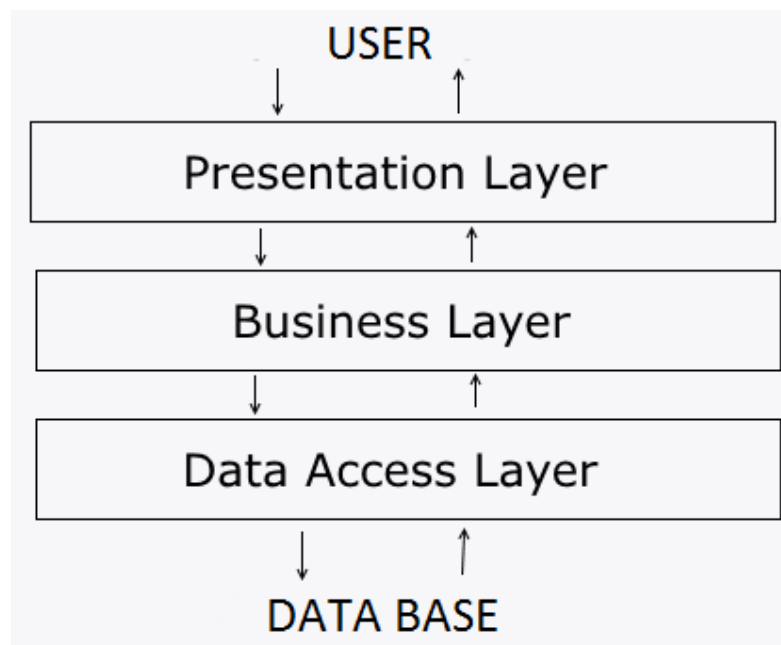
Тут відразу треба сказати, що багаторівневої архітектурою часто позначають два не зовсім пов'язаних поняття: n-layer і n-tier. І layer, і tier, як правило, позначаються словом "рівень", іноді по відношенню до "layer" ще вживається слово "шар". Однак в обох випадках рівні будуть різного порядку.

Tier представляє собою фізичний рівень. Тобто якщо ми говоримо про трирівневу архітектуру, то n-tier додаток могло бути розділене на такі рівні: сервер бази даних, веб-додаток на веб-сервері і браузер користувача. Тобто кожен рівень представляв би особливий окремий фізичний процес, навіть якщо б і сервер баз даних, і веб-сервер, і браузер користувача знаходилися б на одному комп'ютері. Якби в якості клієнта альтернативно використовувалося мобільний додаток, то це був би ще один фізичний рівень.

Layer представляє собою логічний рівень. Тобто у нас може бути рівень доступу до даних, рівень бізнес-логіки, рівень представлення, рівень сервісів і так далі. При цьому логічні рівні не збігаються з фізичними. Як правило рівень представлення в додатку ASP.NET містить і контролери, які обробляють ввід, і представлення, які відображаються в веб-браузері, тобто розділяється на два фізичних рівня.

В даному випадку ми будемо говорити саме про логічні рівні, тобто про n-layer архітектуру.

Класична трирівнева система складається з наступних рівнів:



Presentation layer (рівень представлення): це той рівень, з яким безпосередньо взаємодіє користувач. Цей рівень включає компоненти для користувача інтерфейсу, механізм отримання вводу від користувача. Стосовно до asp.net mvc на даному рівні розташовані представлення і всі ті компоненти, який складають призначений для користувача інтерфейс (стили, статичні

сторінки html, javascript), а також моделі представлень, контролери, об'єкти контексту запиту.

Business layer (рівень бізнес-логіки): містить набір компонентів, які відповідають за обробку отриманих від рівня представлень даних, реалізує всю необхідну логіку додатка, всі обчислення, взаємодіє з базою даних і передає рівнем подання результат обробки.

Data Access layer (рівень доступу до даних): зберігає моделі, що описують використовувані сутності. Також тут розміщуються специфічні класи для роботи з різними технологіями доступу до даних, наприклад, клас контексту даних Entity Framework. Тут також зберігаються репозиторії, через які рівень бізнес-логіки взаємодіє з базою даних.

При цьому треба зазначити, що крайні рівні не можуть взаємодіяти між собою, тобто рівень представлення (стосовно ASP.NET MVC, контролери) не можуть безпосередньо звертатися до бази даних і навіть до рівня доступу до даних, а тільки через рівень бізнес-логіки.

Рівень доступу до даних не залежить від інших рівнів, рівень бізнес-логіки залежить від рівня доступу до даних, а рівень представлення - від рівня бізнес-логіки. Компоненти, як правило, повинні бути слабо зв'язані (loose coupling), тому невід'ємною ланкою багаторівневих додатків є впровадження залежностей.

При чому про ASP.NET MVC ми говоримо перш за все стосовно до рівня подання, інші ж рівні можуть бути реалізовані незалежно і можуть використовуватися в додатках на інших технологіях, як Windows Forms, WPF і т.д. і, як правило, всі додатки в цілому будуть представляти рішення (solution) в Visual Studio, а окремі рівні - проекти. У той же час невірно думати, що рівень обов'язково повинен відповідати окремому проекту. При необхідності ми можемо розділити один рівень на кілька проектів, головне, щоб його функціонал представляв єдину логічну ланку.

4.6.5 Монолітна архітектура

Перш ніж безпосередньо приступити до створення багаторівневого додатку, побудуємо простий стандартний додаток ASP.NET MVC з монолітною архітектурою, яку потім переробимо в трирівневу.

Ця програма буде імітувати роботу магазину смартфонів. Нехай у нас буде визначено наступні моделі Phone і Order:

```
public class Phone
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Company { get; set; }
    public decimal Price { get; set; }

    public ICollection<Order> Orders { get; set; }
}
public class Order
{
```

```

public int Id { get; set; }
public decimal Sum { get; set; }
public string PhoneNumber { get; set; }
public string Address { get; set; }
public DateTime Date { get; set; }

public int PhoneId { get; set; }
public Phone Phone { get; set; }
}

```

Модель Phone описує смартфон, модель Order - замовлення на покупку, і об'єкти цих моделей будуть зберігатися в базі даних. Для взаємодії з базою даних визначено наступний клас контексту:

```

public class MobileContext : DbContext
{
    public DbSet<Phone> Phones { get; set; }
    public DbSet<Order> Orders { get; set; }
}

public class StoreDbInitializer : DropCreateDatabaseAlways<MobileContext>
{
    protected override void Seed(MobileContext db)
    {
        db.Phones.Add(new Phone { Name = "Nokia Lumia 630", Company = "Nokia", Price = 220 });
        db.Phones.Add(new Phone { Name = "iPhone 6", Company = "Apple", Price = 320 });
        db.Phones.Add(new Phone { Name = "LG G4", Company = "LG", Price = 260 });
        db.Phones.Add(new Phone { Name = "Samsung Galaxy S 6", Company = "Samsung", Price = 300 });
    };
    db.SaveChanges();
}

```

Крім того, тут також визначено ініціалізатор, який додає в БД початкові значення. Для оформлення замовлення визначена додаткова модель OrderViewModel:

```

public class OrderViewModel
{
    public int PhoneId { get; set; } // id телефону
    public string Address { get; set; } // адреса
    public string PhoneNumber { get; set; } // номер телефону покупця
}

```

За логіку відповідає HomeController:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

```

```

using System.Web.Mvc;
using MonolitMvcApp.Models;

namespace MonolitMvcApp.Controllers
{
    public class HomeController : Controller
    {
        MobileContext db = new MobileContext();
        public ActionResult Index()
        {
            return View(db.Phones.ToList());
        }

        public ActionResult MakeOrder(int? id)
        {
            if (id == null)
                return HttpNotFound();
            Phone phone = db.Phones.Find(id);
            if (phone == null)
                return HttpNotFound();
            OrderViewModel orderModel = new OrderViewModel { PhoneId=phone.Id };
            return View(orderModel);
        }
        [HttpPost]
        public ActionResult MakeOrder(OrderViewModel orderModel)
        {
            if (ModelState.IsValid)
            {
                Phone phone = db.Phones.Find(orderModel.PhoneId);
                if (phone == null)
                    return HttpNotFound();
                decimal sum = phone.Price;

                // якщо зараз перше число місяця, тоді скидка - 10%
                if (DateTime.Now.Day == 1)
                    sum = sum - sum * 0.1m;

                Order order = new Order
                {
                    PhoneId=phone.Id,
                    PhoneNumber = orderModel.PhoneNumber,
                    Address=orderModel.Address,
                    Date = DateTime.Now,
                    Sum = sum
                };
                db.Orders.Add(order);
                db.SaveChanges();
                return Content("<h2>Ваш заказ успішно оформлено</h2>");
            }
            return View(orderModel);
        }
    }
}

```



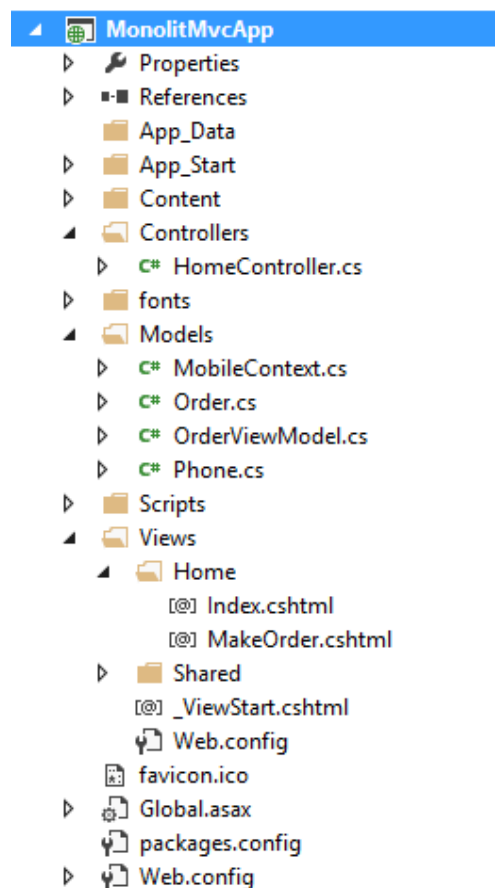
```

protected override void Dispose(bool disposing)
{
    db.Dispose();
    base.Dispose(disposing);
}
}
}

```

Контролер має три методи. Метод Index виводить в представлення список смартфонів. GET-версія методу MakeOrder генерує представлення для оформлення замовлення. POST-версія методу MakeOrder отримує введені дані, і, якщо все вірно, додає об'єкт в базу даних. При цьому, якщо поточне число - перше число будь-якого місяця, то застосовується знижка в 10%.

Два представлення виглядають стандартно. У підсумку виходить наступна структура проекту:



Таким чином, проект містить досить простий і невеликий функціонал, і обраний спосіб реалізації - монолітна архітектура, (тобто цілісний додаток без поділу на рівні) в даному випадку цілком підходить, також як він і підходить для простих реалізацій CRUD-інтерфейсів.

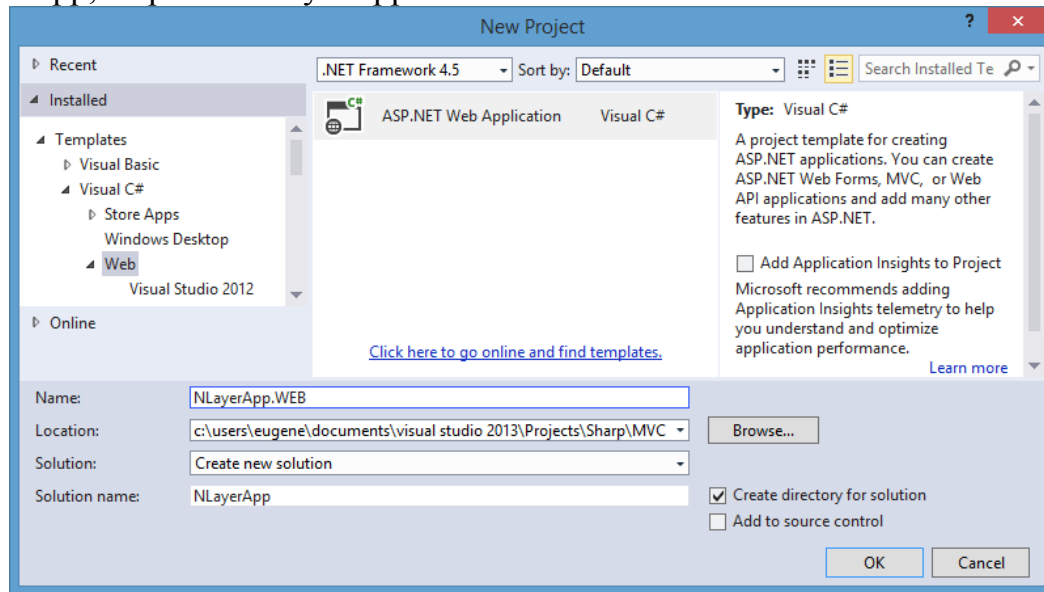
Однак якщо ми працюємо над великим додатком, тим більше, якщо над ним працюємо не тільки ми, а ціла команда, і якщо додаток в майбутньому буде динамічно розвиватися і розширюватися, то монолітна архітектура навряд чи буде в багатьох ситуаціях найкращим вибором.

В цьому випадку логічно було б розділити додаток на окремі компоненти, які окремо простіше розробляти і розширювати.

Тепер створимо той самий додаток, але вже із застосуванням трирівневої архітектури.

4.6.6 Data Access Layer

Першим рівнем, який ми реалізуємо, буде Data Access Layer або рівень доступу до даних. Цей рівень зазвичай містить всі моделі даних, що зберігаються в БД, а також класи, через які йде взаємодія з БД. Спочатку створимо новий проект ASP.NET MVC без аутентифікації. При цьому солюшн назовемо NLayerApp, а проект NLayerApp.WEB:



Проект NLayer App.WEB представлятиме рівень представлення. Після створення солюшена і проекту додамо в солюшн новий проект по типу Class Library, який назвемо NLayerApp.DAL. Цей проект буде представляти рівень доступу до даних.

Насамперед визначимо моделі, об'єкти яких будуть зберігатися в БД. Для цього в проект додамо нову папку Entities, а в цій папці визначимо наступний клас моделі Phone:

```
using System.Collections.Generic;
namespace NLayerApp.DAL.Entities
{
    public class Phone
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string Company { get; set; }
        public decimal Price { get; set; }
        public ICollection<Order> Orders { get; set; }
    }
}
```

```
}
```

І моделі Order:

```
using System;
namespace NLayerApp.DAL.Entities
{
    public class Order
    {
        public int Id { get; set; }
        public decimal Sum { get; set; }
        public string PhoneNumber { get; set; }
        public string Address { get; set; }

        public int PhoneId { get; set; }
        public Phone Phone { get; set; }

        public DateTime Date { get; set; }
    }
}
```

Тепер додамо в проект через NuGet пакет Entity Framework і також визначимо в проєкті нову папку, яка буде називатися EF. У цю папку додамо клас контексту даних:

```
using System.Collections.Generic;
using System.Data.Entity;
using System.Threading.Tasks;
using NLayerApp.DAL.Entities;

namespace NLayerApp.DAL.EF
{
    public class MobileContext : DbContext
    {
        public DbSet<Phone> Phones { get; set; }
        public DbSet<Order> Orders { get; set; }

        static MobileContext()
        {
            Database.SetInitializer<MobileContext>(new StoreDbInitializer());
        }
        public MobileContext(string connectionString)
            : base(connectionString)
        {
        }
    }

    public class StoreDbInitializer : DropCreateDatabaseIfModelChanges<MobileContext>
    {
        protected override void Seed(MobileContext db)
        {
        }
    }
}
```

```

        db.Phones.Add(new Phone { Name = "Nokia Lumia 630", Company = "Nokia", Price = 220 });
        db.Phones.Add(new Phone { Name = "iPhone 6", Company = "Apple", Price = 320 });
        db.Phones.Add(new Phone { Name = "LG G4", Company = "LG", Price = 260 });
        db.Phones.Add(new Phone { Name = "Samsung Galaxy S 6", Company = "Samsung", Price =
300 });
        db.SaveChanges();
    }
}
}

```

Як і в проєкті з монолітної архітектурою тут також використовується ініціалізатор БД, який тепер виконується в статичному конструкторі контексту. Крім того, для рядка підключення конструктор контексту приймає рядковий параметр `connectionString`.

Для збільшення гнучкості підключення до БД використовуються репозиторії. Тому спочатку визначимо в проєкті ще одну папку `Interfaces`, і в неї додамо інтерфейс репозиторіїв `IRepository`:

```

using System;
using System.Collections.Generic;

namespace NLayerApp.DAL.Interfaces
{
    public interface IRepository<T> where T : class
    {
        IEnumerable<T> GetAll();
        T Get(int id);
        IEnumerable<T> Find(Func<T, Boolean> predicate);
        void Create(T item);
        void Update(T item);
        void Delete(int id);
    }
}

```

Оскільки ми будемо використовувати кілька сховищ для кожної сутності, то для спрощення доступу до БД будемо використовувати патерн `Unit Of Work`. І для цього також в папку `Interfaces` додамо новий інтерфейс `IUnitOfWork`:

```

using NLayerApp.DAL.Entities;
using System;

namespace NLayerApp.DAL.Interfaces
{
    public interface IUnitOfWork : IDisposable
    {
        IRepository<Phone> Phones { get;}
        IRepository<Order> Orders { get;}
        void Save();
    }
}

```

Для зберігання реалізацій даних інтерфейсів визначимо в проєкті ще одну папку `Repositories`. Додамо в неї клас сховища для смартфонів `PhoneRepository`:

```
using System;
using System.Collections.Generic;
using System.Linq;
using NLayerApp.DAL.Entities;
using NLayerApp.DAL.EF;
using NLayerApp.DAL.Interfaces;
using System.Data.Entity;

namespace NLayerApp.DAL.Repositories
{
    public class PhoneRepository : IRepository<Phone>
    {
        private MobileContext db;

        public PhoneRepository(MobileContext context)
        {
            this.db = context;
        }

        public IEnumerable<Phone> GetAll()
        {
            return db.Phones;
        }

        public Phone Get(int id)
        {
            return db.Phones.Find(id);
        }

        public void Create(Phone book)
        {
            db.Phones.Add(book);
        }

        public void Update(Phone book)
        {
            db.Entry(book).State = EntityState.Modified;
        }

        public IEnumerable<Phone> Find(Func<Phone, Boolean> predicate)
        {
            return db.Phones.Where(predicate).ToList();
        }

        public void Delete(int id)
        {
            Phone book = db.Phones.Find(id);
            if (book != null)
                db.Phones.Remove(book);
        }
    }
}
```

```

    }
}
}

```

Також додамо в цю ж папку клас сховища для замовлень OrderRepository:

```

using System;
using System.Collections.Generic;
using System.Data.Entity;
using System.Linq;
using NLayerApp.DAL.Entities;
using NLayerApp.DAL.EF;
using NLayerApp.DAL.Interfaces;

namespace NLayerApp.DAL.Repositories
{
    public class OrderRepository : IRepository<Order>
    {
        private MobileContext db;
        public OrderRepository(MobileContext context)
        {
            this.db = context;
        }
        public IEnumerable<Order> GetAll()
        {
            return db.Orders.Include(o => o.Phone);
        }

        public Order Get(int id)
        {
            return db.Orders.Find(id);
        }

        public void Create(Order order)
        {
            db.Orders.Add(order);
        }

        public void Update(Order order)
        {
            db.Entry(order).State = EntityState.Modified;
        }
        public IEnumerable<Order> Find(Func<Order, Boolean> predicate)
        {
            return db.Orders.Include(o => o.Phone).Where(predicate).ToList();
        }
        public void Delete(int id)
        {
            Order order = db.Orders.Find(id);
            if (order != null)
                db.Orders.Remove(order);
        }
    }
}

```

```

    }
}
}

```

I також в папку Repositories додамо клас EF Unit Of Work:

```

using System;
using NLayerApp.DAL.EF;
using NLayerApp.DAL.Interfaces;
using NLayerApp.DAL.Entities;

namespace NLayerApp.DAL.Repositories
{
    public class EFUnitOfWork : IUnitOfWork
    {
        private MobileContext db;
        private PhoneRepository phoneRepository;
        private OrderRepository orderRepository;

        public EFUnitOfWork(string connectionString)
        {
            db = new MobileContext(connectionString);
        }

        public IRepository<Phone> Phones
        {
            get
            {
                if (phoneRepository == null)
                    phoneRepository = new PhoneRepository(db);
                return phoneRepository;
            }
        }

        public IRepository<Order> Orders
        {
            get
            {
                if (orderRepository == null)
                    orderRepository = new OrderRepository(db);
                return orderRepository;
            }
        }

        public void Save()
        {
            db.SaveChanges();
        }

        private bool disposed = false;

        public virtual void Dispose(bool disposing)
        {

```

```

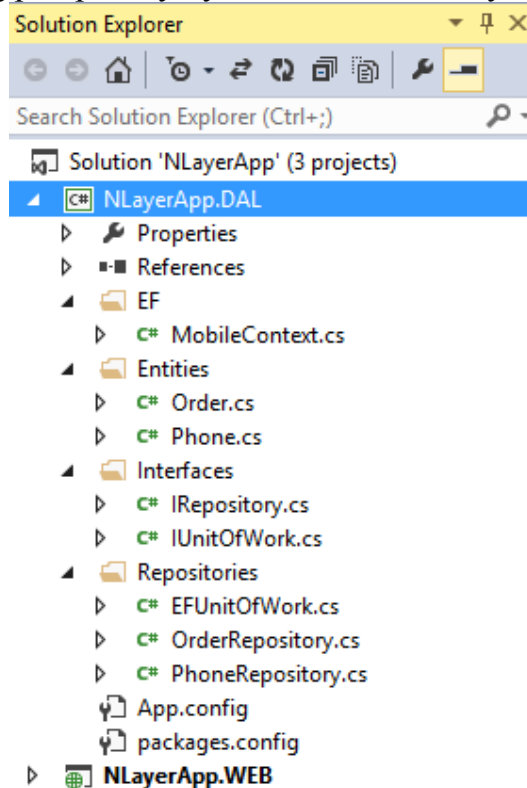
        if (!this.disposed)
        {
            if (disposing)
            {
                db.Dispose();
            }
            this.disposed = true;
        }
    }

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }
}
}

```

Клас EF UnitOfWork в конструкторі приймає рядок - назву підключення, яка потім буде передаватися в конструктор контексту даних. Власне через EFUnitOfWork ми і будемо взаємодіяти з базою даних.

В результаті структура проекту буде виглядати наступним чином:



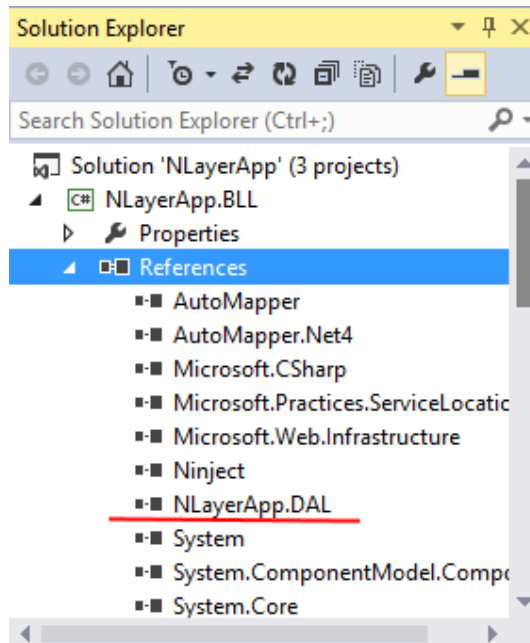
Власне, це і весь рівень доступу до даних, який ми могли б зробити для проекту з минулого теми, тому тепер перейдемо до побудови бізнес-логіки.

4.6.7 Business Logic Layer

Business Logic Layer або бізнес-рівень інкапсулює всю бізнес-логіку, всі необхідні обчислення, отримує об'єкти з рівня доступу до даних і передає їх на

рівень представлення, або, навпаки, отримує дані з рівня представлення і передає їх на рівень даних.

Отже, додамо в рішення новий проект по типу Class Library, який назовемо NLayerApp.BLL. Оскільки бізнес-рівень буде використовувати класи з рівня доступу до даних, то нам треба додати на нього посилання:



Рівень представлення не може безпосередньо отримувати дані з бази даних. В даному випадку BLL виступатиме в ролі посередника між двома рівнями. Але також треба враховувати, що безпосередньо він не може передавати в контролери об'єкти Phone і Order, так як рівень представлення не повинен мати доступ до функціональності рівня DAL. Тому нам потрібні проміжні сутності. Отже, додамо в проект BLL папку, яку назовемо DTO. Визначимо в ній новий клас PhoneDTO:

```
namespace NLayerApp.BLL.DTO
{
    public class PhoneDTO
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string Company { get; set; }
        public decimal Price { get; set; }
    }
}
```

Через цей клас ми будемо передавати об'єкти смартфонів між рівнями. Але хоча даний клас багато в чому схожий з визначення на клас Phone, це не обов'язкова умова. Клас PhoneDTO повинен містити тільки ті дані, які ми збираємося передати на рівень представлення або, навпаки, отримати з цього

рівня. Тобто це те, що називається Data Transfer Object - спеціальна модель для передачі даних.

Подібним чином визначимо в тій же папці клас OrderDTO:

```
using System;
using NLayerApp.DAL.Entities;

namespace NLayerApp.BLL.DTO
{
    public class OrderDTO
    {
        public int Id { get; set; }
        public string PhoneNumber { get; set; }
        public string Address { get; set; }
        public int PhoneId { get; set; }
        public DateTime? Date { get; set; }
    }
}
```

Для спрощення зіставлення класів моделей додамо в проект через NuGet бібліотеку AutoMapper.

Крім простих класів типу DTO BLL може містити класи, які описують бізнес-логіку. Зокрема, якщо ми повернемося до проекту з монолітної архітектурою, то там був невеликий функціонал знижки. І в принципі знижку можна виділити в окремий клас. Але для його зберігання додамо в BLL нову папку BusinessModels. І в ній визначимо клас знижки Discount:

```
using System;

namespace NLayerApp.BLL.BusinessModels
{
    public class Discount
    {
        public Discount(decimal val)
        {
            _value = val;
        }
        private decimal _value = 0;
        public decimal Value { get { return _value; }}
        public decimal GetDiscountedPrice(decimal sum)
        {
            if(DateTime.Now.Day==1)
                return sum - sum * _value;
            return sum;
        }
    }
}
```

Це бізнес-модель, яка не тільки зберігає стан, як класи DTO, але і може описувати деякі дії. Виділення бізнес-моделей в проєкті цілком індивідуально і залежить від конкретної ситуації.

Велику роль в додатку грає валідація даних. Здебільшого за валідацію відповідає саме BLL. У контролері ми легко можемо провалідувати модель через об'єкт ModelState і при необхідності повернути в представлення повідомлення про помилки. Але на рівні BLL ModelState недоступний. Однак ми все ж можемо використовувати валідацію з передачею помилок в рівень представлення.

Отже, визначимо в BLL новий каталог Infrastructure і в нього додамо новий клас ValidationException:

```
using System;

namespace NLayerApp.BLL.Infrastructure
{
    public class ValidationException : Exception
    {
        public string Property { get; protected set; }
        public ValidationException(string message, string prop) : base(message)
        {
            Property = prop;
        }
    }
}
```

Клас ValidationException успадковується від базового класу винятків Exception і визначає властивість Property. Ця властивість дозволяє зберегти назву властивості моделі, яке некоректна і не проходить валідацію. І також передаючи в конструктор базового класу параметр message, ми визначаємо повідомлення, яке буде виводитися для некоректного властивості в Property.

Взаємодіяти між іншими двома рівнями ми будемо через спеціальний сервіс. Знову ж для більшої гнучкості спочатку визначимо його інтерфейс. Для зберігання інтерфейсу додамо в BLL папку Interfaces і в неї покладемо інтерфейс IOrderService:

```
using NLayerApp.BLL.DTO;
using System.Collections.Generic;
namespace NLayerApp.BLL.Interfaces
{
    public interface IOrderService
    {
        void MakeOrder(OrderDTO orderDto);
        PhoneDTO GetPhone(int? id);
        IEnumerable<PhoneDTO> GetPhones();
        void Dispose();
    }
}
```

Інтерфейс визначає 4 методи: отримання всіх смартфонів для вибору товару, вибір смартфона для замовлення, оформлення замовлення і метод Dispose. Для зберігання реалізацій інтерфейсу визначимо в проекті ще одну папку Послуги. Додамо в неї клас сервісу OrderService:

```
using System;
using NLayerApp.BLL.DTO;
using NLayerApp.DAL.Repositories;
using NLayerApp.DAL.Entities;
using NLayerApp.BLL.BusinessModels;
using NLayerApp.DAL.Interfaces;
using NLayerApp.BLL.Infrastructure;
using NLayerApp.BLL.Interfaces;
using System.Collections.Generic;
using AutoMapper;

namespace NLayerApp.BLL.Services
{
    public class OrderService : IOrderService
    {
        IUnitOfWork Database { get; set; }

        public OrderService(IUnitOfWork uow)
        {
            Database = uow;
        }

        public void MakeOrder(OrderDTO orderDto)
        {
            Phone phone = Database.Phones.Get(orderDto.PhoneId);

            // валідація
            if (phone == null)
                throw new ValidationException("Телефон не знайдено", "");
            // застосовуємо скидку
            decimal sum = new Discount(0.1m).GetDiscountedPrice(phone.Price);
            Order order = new Order
            {
                Date = DateTime.Now,
                Address = orderDto.Address,
                PhoneId = phone.Id,
                Sum = sum,
                PhoneNumber = orderDto.PhoneNumber
            };
            Database.Orders.Create(order);
            Database.Save();
        }

        public IEnumerable<PhoneDTO> GetPhones()
        {
            // застосовуємо автомAPPER для проєкції одної колекції на іншу
        }
    }
}
```

```

        Mapper.CreateMap<Phone, PhoneDTO>();
        return Mapper.Map<IEnumerable<Phone>, List<PhoneDTO>>>(Database.Phones.GetAll());
    }

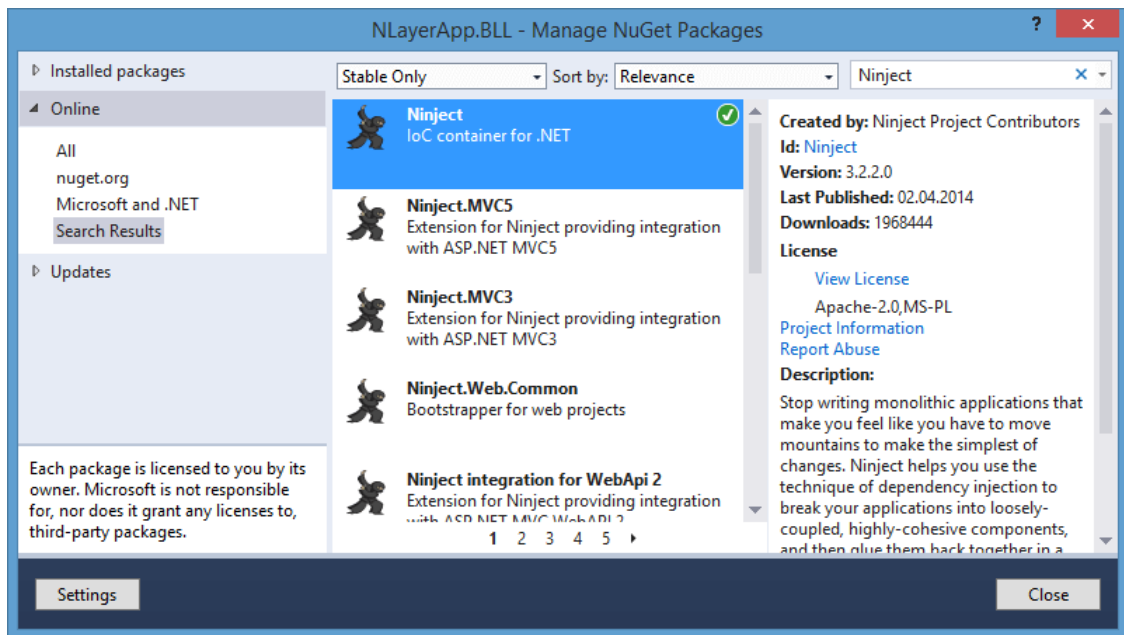
    public PhoneDTO GetPhone(int? id)
    {
        if (id == null)
            throw new ValidationException("Не встановлений id телефону", "");
        var phone = Database.Phones.Get(id.Value);
        if (phone == null)
            throw new ValidationException("Телефон не знайдений", "");
        // застосовуємо автомAPPER для проєкції Phone на PhoneDTO
        Mapper.CreateMap<Phone, PhoneDTO>();
        return Mapper.Map<Phone, PhoneDTO>(phone);
    }

    public void Dispose()
    {
        Database.Dispose();
    }
}
}

```

Order Service в конструкторі приймає об'єкт UnitOfWork, через який йде взаємодія з рівнем DAL.

Метод MakeOrder() отримує об'єкт для збереження з рівня представлення і створює по ньому об'єкт Order і зберігає його в базу даних. Метод GetPhones() отримує всі смартфони і за допомогою автомAPPERа перетворює їх в PhoneDTO і передає на рівень представлення. І метод GetPhone() передає окремий смартфон на рівень представлення. Оскільки в даному випадку ми не задаємо в конструкторі явно об'єкт IUnitOfWork, то нам треба використовувати впровадження залежностей для передачі конкретної реалізації даного інтерфейсу в OrderService. Для цього додамо в проект через NuGet пакет Ninject:



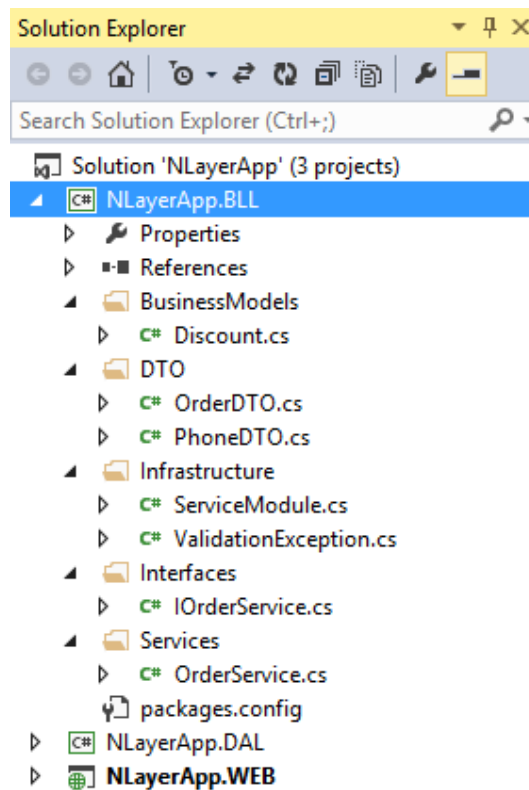
Потім додаємо до проекту в раніше створену папку Infrastructure наступний клас:

```
using Ninject.Modules;
using NLayerApp.DAL.Interfaces;
using NLayerApp.DAL.Repositories;
```

```
namespace NLayerApp.BLL.Infrastructure
{
    public class ServiceModule : NinjectModule
    {
        private string connectionString;
        public ServiceModule(string connection)
        {
            connectionString = connection;
        }
        public override void Load()
        {
            Bind<IUnitOfWork>().To<EFUnitOfWork>().WithConstructorArgument(connectionString);
        }
    }
}
```

Service Module представляє спеціальний модуль Ninject, який служить для організації зіставлення залежностей. Зокрема, він встановлює використання EFUnitOfWork як об'єкт IUnitOfWork. Крім того, тут через конструктор передається назва підключення, яка в підсумку буде визначатися в файлі web.config проекту, що представляє рівень представлення.

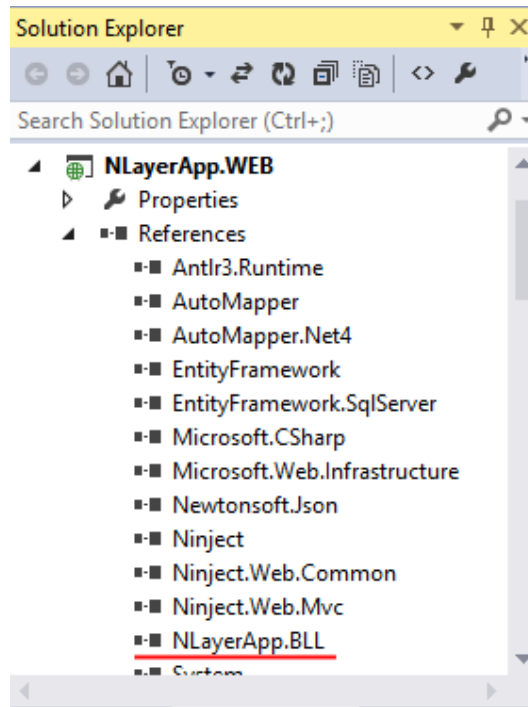
В результаті структура проекту буде виглядати наступним чином:



Тепер створимо рівень представлення.

4.6.8 Presentation Layer

Presentation Layer або рівень представлення відповідає за взаємодію з користувачем. Для цього в нашому солюшені вже є проект NLayerApp.WEB. Правда, фактично всю основну роботу і весь основний функціонал ми вже створили, залишилося тільки його застосувати на рівні представлення. Для цього спочатку додамо в проект посилання на проект BLL:



Додамо в проект через NuGet пакети AutoMapper, Entity Framework і Ninject.MVC5, які нам потім будуть потрібні. Для представлення даних в представленнях визначимо дві моделі представлення в папці Models. Клас PhoneViewModel, що представляє смартфони:

```
namespace NLayerApp.WEB.Models
{
    public class PhoneViewModel
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string Company { get; set; }
        public decimal Price { get; set; }
    }
}
```

І також клас Order ViewModel, що представляє замовлення:

```
namespace NLayerApp.WEB.Models
{
    public class OrderViewModel
    {
        public int PhoneId { get; set; }
        public string Address { get; set; }
        public string PhoneNumber { get; set; }
    }
}
```

Тоді контролер HomeController буде виглядати наступним чином:


```

using NLayerApp.BLL.Interfaces;
using NLayerApp.BLL.DTO;
using NLayerApp.WEB.Models;
using AutoMapper;
using NLayerApp.BLL.Infrastructure;

namespace NLayerApp.WEB.Controllers
{
    public class HomeController : Controller
    {
        IOrderService orderService;
        public HomeController(IOrderService serv)
        {
            orderService = serv;
        }
        public ActionResult Index()
        {
            Mapper.CreateMap<PhoneDTO, PhoneViewModel>();
            var phones = Mapper.Map<IEnumerable<PhoneDTO>,
List<PhoneViewModel>>(orderService.GetPhones());
            return View(phones);
        }

        public ActionResult MakeOrder(int? id)
        {
            try
            {
                Mapper.CreateMap<PhoneDTO, OrderViewModel>()
                    .ForMember("PhoneId", opt => opt.MapFrom(src => src.Id));
                var order = Mapper.Map<PhoneDTO, OrderViewModel>(orderService.GetPhone(id));
                return View(order);
            }
            catch (ValidationException ex)
            {
                return Content(ex.Message);
            }
        }
        [HttpPost]
        public ActionResult MakeOrder(OrderViewModel order)
        {
            try
            {
                Mapper.CreateMap<OrderViewModel, OrderDTO>();
                var orderDto = Mapper.Map<OrderViewModel, OrderDTO>(order);
                orderService.MakeOrder(orderDto);
                return Content("<h2>Ваше замовлення оформлене</h2>");
            }
            catch (ValidationException ex)
            {

```

```

        ModelState.AddModelError(ex.Property, ex.Message);
    }
    return View(order);
}
protected override void Dispose(bool disposing)
{
    orderService.Dispose();
    base.Dispose(disposing);
}
}
}

```

Контролер виробляє всі ті ж дії, що виконував аналогічний контролер в проєкті з монолітної архітектурою. Щоб перехоплювати помилки валідації, які виникають на рівні BLL, тут використовується конструкція `try..catch`, в якій перехоплюються винятки `ValidationException`, а відповідні повідомлення про помилки передаються в представлення.

І фінальний момент - впровадження залежностей. Для установки залежностей додамо в проєкт папку `Util`, а в неї клас співствника залежностей:

```

using NLayerApp.BLL.Services;
using System.Web.Mvc;
using Ninject;
using System.Collections.Generic;
using System;
using NLayerApp.BLL.Interfaces;

namespace NLayerApp.WEB.Util
{
    public class NinjectDependencyResolver : IDependencyResolver
    {
        private IKernel kernel;
        public NinjectDependencyResolver(IKernel kernelParam)
        {
            kernel = kernelParam;
            AddBindings();
        }
        public object GetService(Type serviceType)
        {
            return kernel.TryGet(serviceType);
        }
        public IEnumerable<object> GetServices(Type serviceType)
        {
            return kernel.GetAll(serviceType);
        }
        private void AddBindings()
        {
            kernel.Bind<IOrderService>().To<OrderService>();
        }
    }
}

```

```
}
```

В даному випадку інтерфейс `IOrderService` зіставляється з класом `OrderService`. Тепер перейдемо до папки `App_Start`. Після додавання пакета `Ninject.MVC5` в цю папку також додається файл `NinjectWebCommon.cs`. Знайдемо в ньому метод `RegisterServices` і змінимо його наступним чином:

```
private static void RegisterServices(IKernel kernel)
{
    System.Web.Mvc.DependencyResolver.SetResolver(new
    NLayerApp.WEB.Util.NinjectDependencyResolver(kernel));
}
```

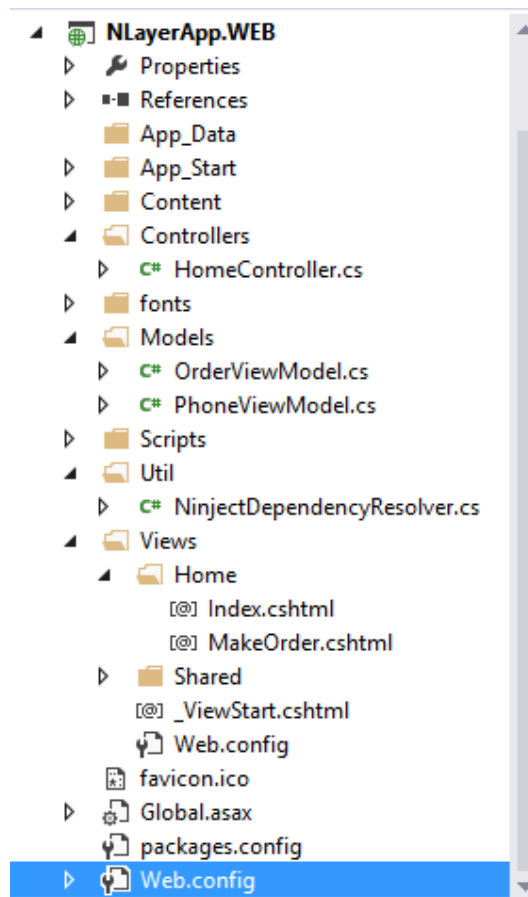
В даному випадку ми реєструємо свій співставник залежностей. Але в проєкті `BLL` також був створений модуль, який зіставляв `IUnitOfWork` з `EFUnitOfWork` і передавав їм назву підключення. Цей модуль нам теж потрібно задіяти, для цього в цьому ж файлі перейдемо до методу `CreateKernel()` і змінимо його наступним чином:

```
private static IKernel CreateKernel()
{
    // встановлюємо рядок підключення
    var modules = new INinjectModule[] { new ServiceModule("DefaultConnection") };
    var kernel = new StandardKernel(modules);
    try
    {
        kernel.Bind<Func<IKernel>>().ToMethod(ctx => () => new Bootstrapper().Kernel);
        kernel.Bind<IHttpModule>().To<HttpApplicationInitializationHttpModule>();

        RegisterServices(kernel);
        return kernel;
    }
    catch
    {
        kernel.Dispose();
        throw;
    }
}
```

В даному випадку встановлюється рядок підключення `DefaultConnection`. Відповідно це підключення має бути визначено у файлі `web.config`. При бажанні потім ми зможемо змінити рядок підключення.

Проєкт рівня представлення матиме наступну структуру:



Таким чином, ми створили програму, що реалізує трирівневу архітектуру, яка виконує всі ті ж дії, що і проект з монолітною архітектурою. Однак тепер нам легше його тестувати, розподілено розробляти в команді, розвивати і розширювати.

Контрольні питання.

1. Особливості побудови архітектури програмного забезпечення для Web-додатків.
2. Випадки застосування багаторівневої архітектури.
3. Випадки застосування монолітної архітектури.

Список використаних джерел

1. Оператори C# - [Електронний ресурс] – Електрон. дані (64 файлів). – 2012–2017 – Режим доступу: <https://msdn.microsoft.com/ru-ru/library/6a71f45d.aspx>
2. Справочник по C# - [Електронний ресурс] – Електрон. дані (11 файлів). – 2012–2017 – Режим доступу: <https://msdn.microsoft.com/ru-ru/library/6a71f45d.aspx>
3. Введение в ASP.NET MVC 5 - [Електронний ресурс] – Електрон. дані (159 файлів). – 2015–2017 – Режим доступу: <http://metanit.com/sharp/mvc5/1.1.php>
4. Руководство по C# - Часть 1 [Електронний ресурс] – Електрон. дані (78 файлів). – 2015–2017 – Режим доступу: https://professorweb.ru/my/csharp/charp_theory/level1/index.php
5. Руководство по C# - Часть 2 [Електронний ресурс] – Електрон. дані (67 файлів). – 2015–2017 – Режим доступу: https://professorweb.ru/my/csharp/charp_theory/level1/index1.php
6. Н. Секунов “Самоучитель C#”. – СПб.: БХВ-Петербург, 2001. – 576с.
7. Гуриков С. Р. Введение в программирование на языке Visual C#; Форум, Инфра-М, 2013. - 448 с.
8. Мартин Р. С., Мартин М. Принципы, паттерны и методики гибкой разработки на языке C#; Символ-Плюс, 2011. - 768 с.
9. Пугачев С., Шериев А., Кичинский К. Разработка приложений для Windows 8 на языке C#; БХВ-Петербург, 2013. - 416 с.
10. Фленов Михаил Библия C#; БХВ-Петербург, 2009. - 560 с