МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

МЕТОДИЧНІ ВКАЗІВКИ

до виконання лабораторних робіт

ЧАСТИНА ІІ

«Стек технології»

з дисципліни «Стек технології»
для студентів 122 спеціальності «Комп’ютерні науки»

Затверджено
редакційно-видавничою
радою університету,
протокол № 3 від 16.10.2023 р.

Харків
НТУ «ХПІ»
2023

Методичні вказівки до виконання лабораторних робіт з дисципліни «Стек технології» для студентів 122 спеціальності «Комп’ютерні науки» / уклад. А.О. Лисенко, О.О. Лисенко – Харків : НТУ «ХПІ». 2023 – 149с.

Укладачі: А.О. ЛИСЕНКО
О.О. ЛИСЕНКО

Рецензент: М.А. ГРИНЧЕНКО

Кафедра стратегічного управління

ЗМІСТ

ВСТУП

Після того, як ми розглянули базові можливості платформи .NET, такі як робота з колекціями, потоками та LINQ у першій частині нашого посібника, настав час перейти до більш спеціалізованої та витонченої теми – розробки веб-додатків. У цій частині методичного посібника ми зосередимося на вивченні та застосуванні шаблонів проектування MVC (Model-View-Controller) та REST (Representational State Transfer), які є фундаментальними для сучасної веб-розробки на платформі .NET.

Цей розділ має на меті не лише ознайомити вас із теоретичними основами MVC та REST, але й демонструвати, як ці концепції можуть бути втілені в життя за допомогою інструментів та бібліотек .NET, таких як ASP.NET та Entity Framework. Ви навчитеся створювати гнучкі, масштабовані та безпечні веб-додатки, що відповідають сучасним вимогам і стандартам індустрії.

Лабораторні роботи, представлені у цій частині, допоможуть Вам закріпити навчальний матеріал на практиці. Кожна робота включає детальний опис завдань, які дозволять вам крок за кроком освоїти процес розробки веб-додатків, від проектування архітектури до впровадження функціональності та оптимізації продуктивності.

Ми переконані, що знання та навички, отримані у ході вивчення цієї частини, стануть в нагоді не тільки для академічних цілей, але й для вашого подальшого професійного зростання у світі веб-розробки. Заохочуємо вас активно застосовувати отримані знання на практиці, експериментувати та розвивати свої творчі здібності у створенні веб-рішень.

МЕТА, КОМПЕТЕНТНОСТІ, РЕЗУЛЬТАТИ НАВЧАННЯ

Мета викладання курсу полягає в наданні майбутнім спеціалістам у галузі інформаційних технологій компетенцій та навичок, необхідних для розробки додатків на мові програмування C#. Курс орієнтований на ознайомлення з ключовими принципами та сучасними тенденціями у сфері програмування та технологій, акцентуючи увагу на використанні можливостей платформи .NET. Це дасть студентам необхідну базу для ефективної роботи у динамічно розвиваючійся IT-індустрії, а також забезпечить їх готовністю до адаптації та інновацій у майбутньому.

Курс навчання спрямований на те, щоб забезпечити студентів необхідними компетентностями для розвитку та успішної кар'єри у галузі інформаційних технологій. Основна увага приділяється формуванню глибоких теоретичних знань та практичних навичок у сфері сучасних технологій і інструментів для розробки складних програмних систем. Курс сприяє розвитку вміння застосовувати отримані знання на різних етапах життєвого циклу розробки.

У рамках цього курсу ми визначаємо наступні загальні та фахові компетентності:

Загальні компетентності:

ЗК1. Здатність до абстрактного мислення, аналізу та синтезу.

ЗК2. Здатність застосовувати знання у практичних ситуаціях.

ЗК3. Знання та розуміння предметної області та розуміння професійної діяльності.

ЗК6. Здатність вчитися й оволодівати сучасними знаннями.

ЗК7. Здатність до пошуку, оброблення та аналізу інформації з різних джерел.

ЗК8. Здатність генерувати нові ідеї (креативність).

ЗК11. Здатність приймати обґрунтовані рішення.

Спеціальні (фахові, предметні) компетентності:

СК8. Здатність проектувати та розробляти програмне забезпечення із застосуванням різних парадигм програмування: узагальненого, об’єктно-орієнтованого, функціонального, логічного, з відповідними моделями, методами й алгоритмами обчислень, структурами даних і механізмами управління.

СК10. Здатність застосовувати методології, технології та інструментальні засоби для управління процесами життєвого циклу інформаційних і програмних систем, продуктів і сервісів інформаційних технологій відповідно до вимог замовника.

СК12. Здатність забезпечити організацію обчислювальних процесів в інформаційних системах різного призначення з урахуванням архітектури, конфігурування, показників результативності функціонування операційних систем і системного програмного забезпечення.

Результати навчання: В ході вивчання матеріалів курсу та виконання практичних (лабораторних) завдань студенти отримають теоретичні та практичні навички розробки програмних застосувань мовою C# з використанням сучасних технологій, фреймфорків та інструментальніх засобів. Впродовж курсу розглядаються питання взаємодії додатків із зовнішними сховищами даних, деякі аспекти використаня технологій контейнерної віртуалізації, автоматизованого складання та тестування програмного продукту.

Результатом навчання буде набуття наступних компетентностей:

РН1. Застосовувати знання основних форм і законів абстрактно-логічного мислення, основ методології наукового пізнання, форм і методів вилучення, аналізу, обробки та синтезу інформації в предметній області комп'ютерних наук.

РН9. Розробляти програмні моделі предметних середовищ, вибирати парадигму програмування з позицій зручності та якості застосування для реалізації методів та алгоритмів розв’язання задач в галузі комп’ютерних наук.

РНП 1.3. Володіти навичками проєктування та розробки систем, вирішення практичних задач, пов’язаних з розробкою складних систем, методами.

ОПИС НАВЧАЛЬНОЇ ДИСЦИПЛІНИ

(розподіл навчального часу за семестрами та видами навчальних занять)

Семестр	Загальний обсяг (годин) / кредитів ECTS	З них		За видами аудиторних занять (годин)			Індивідуальні завдання студентів (КП, КР, РГ, Р, РЕ)	Поточний контроль	Семестровий контроль	
		Аудиторні заняття (годин)	Самостійна робота (годин)	Лекції	Лабораторні заняття	Практичні заняття, семінари		Контрольні роботи (кількість робіт)	Залік	Іспит
1	2	3	4	5	6	7	8	9	10	11
7	**150 /5**	**64**	**86**	**32**	**32**			**2**		**V**

Співвідношення кількості годин аудиторних занять до загального обсягу складає 43%.

ЛАБОРАТОРНА РОБОТА №5

Тема: розробка веб-додатків з використанням платформи ASP.NET Core та СУБД Microsoft SQL Server

Мета та завдання:

1. Згадати основи веб-програмування: мови HTML, CSS, JavaScript
2. Познайомитись з Active Server Pages (ASP) – технологією від Microsoft для розробки веб-додатків
3. Дізнатися, що таке структура проекту Model-View-Controller. Навчитися будувати логіку веб-додатку, ґрунтуючись на ній.
4. Розробити завершений веб-додаток.

Підготовка

Переконайтеся, що на ПК встановлено:

1. Microsoft SQL Server + Microsoft SQL Server Management Studio (SSMS).
2. Microsoft Visual Studio із засобами веб-розробки.

(буде описано в розділі «Хід Роботи»)

1. ТЕОРЕТИЧНА ЧАСТИНА

У цій лабораторній роботі ми розглядатимемо різні аспекти розробки веб-додатків з використанням платформи ASP.NET Core. ASP.NET Core є сучасною, кросплатфор-менною інкарнацією .NET, відомою як .NET Core. Завдяки своїй кросплатформенності, застосування, створені на .NET Core, можуть бути розгорнуті на серверах, що працюють під різними операційними системами, зокрема Linux та OSX/macOS.

ASP.NET є інтегральною частиною технологічного стеку .NET, що використовується для створення потужних клієнт-серверних інтернет-додатків. Технологія ASP.NET виникла як результат інтеграції попередньої технології ASP (Active Server Pages) та .NET Framework.

Важливо відмітити, що ASP.NET не є простим продовженням ASP. Це повністю нова технологія, розроблена Microsoft в рамках концепції .NET. ASP.NET включає в себе все необхідне для того, щоб зробити процес розробки веб-додатків більш швидким і ефективним, а також спростити їхню підтримку у майбутньому.

1.1. Історія розвитку ASP.NET

У сфері веб-розробки, розуміння історії та еволюції технологій має ключове значення. Цей розділ присвячений історії розвитку ASP.NET - однієї з найпопулярніших та впливових технологій у світі для створення веб-додатків. Розуміння еволюції ASP.NET не лише дає змогу краще зрозуміти, як виникли та розвивались ключові концепції та компоненти цієї платформи, але й надає цінний контекст для сучасних рішень і практик у веб-розробці.

Осмислення історії ASP.NET допомагає усвідомити, чому саме певні функції були впроваджені, які проблеми вони вирішували та як це вплинуло на сучасний стан веб-розробки. Також, це дає можливість оцінити прогрес, який був досягнутий у цій галузі, і зрозуміти, в якому напрямку вона може розвиватися в майбутньому. Таким чином, цей розділ не лише інформативний, але й має практичне значення для розробників, які прагнуть глибше зануритися у світ ASP.NET та веб-розробки в цілому.

Веб-розробка вже давно перестала бути екзотичною новацією. На зорі інтернету веб-сторінки були статичними, а вся навігація забезпечувалася за допомогою гіперпосилань. Недоліки такого підходу швидко стали очевидними. Розробнику було нелегко витримати єдину тему оформлення на всіх сторінках, а перегляд однакових веб-сторінок викликав у користувачів нудьгу.

Веб-сервери почали підтримувати різноманітні технології, завдяки яким вигляд веб-сторінки міг змінюватися залежно від вхідних даних. Одні технології приходили та зникали (наприклад, SSI — Server Side Includes), інші — такі, як CGI (Common Gateway Interface) — у тій чи іншій формі продовжували існування. ASP (Active Server Pages — «активні серверні сторінки») —технологія, запропонована компанією Microsoft у 1996 році для створення Web-додатків. Назва Active Server Pages використовується вже близько двадцяти років, тоді як сама технологія зазнала радикальних змін.

У 1996 році третю версію сервера IIS (Internet Information Services) було випущено з підтримкою ASP першої версії. Технологія ASP будується з урахуванням технології Active Scripting. При цьому застосування Active Scripting не обмежується ASP, оскільки ця технологія також стала частиною Internet Explorer та Windows Script Host.

По суті, ASP — це технологія динамічного створення сторінок на боці сервера, що наблизила проектування та реалізацію Web-додатків до тієї моделі, за якою проектуються та реалізуються звичайні програми.

Технологія ASP дозволяє вбудовувати у веб-сторінки сценарії, написані іншою мовою, та виконувати їх. Теоретично, підтримка більшості мов забезпечувалась за допомогою інтеграції Active Scripting із технологією COM (Component Object Model). У ті часи основна конкуренція в цій галузі точилася між JScript та VBScript — двома мовами, які мали гарну підтримку з боку Microsoft. Втім, у середині 1990-х інші мови, такі як Perl, теж були популярними. Мова Perl до певної міри забезпечувала кросплатформенну сумісність, оскільки він підтримувався в Linux і міг працювати через CGI-шлюзи Apache. Perl була однією з найпопулярніших мов для створення інтерактивних веб-застосунків у ранніх версіях веб-сервера Apache, будучи попередником NCSA HTTPd.

На відміну від PHP, технологія Active Server Pages дозволяла організувати спільне використання бібліотек за допомогою COM. Так сторінки могли звертатися до низькорівневих компонентів, що працюють із високою ефективністю, або користуватися низькорівневою функціональністю. Крім того, ця можливість дозволяла створювати бібліотеки компонентів, які покращують структуру програми та сприяють повторному використанню коду.

ASP.NET

Після випуску сервера Internet Information Services версії 4.0 у 1997 році, компанія Microsoft почала досліджувати можливість нової моделі веб-додатку, яка задовольнила б скарги на ASP, особливо пов'язані з «відділенням оформлення від змісту» і яка дозволить писати «чистий» код. Робота з розробки такої моделі була доручена Марку Андерсу, менеджеру команди IIS, та Скотту Гатрі, який прийшов на роботу в Microsoft у 1997 році.

Андерс та Гатрі впродовж двох місяців розробили початковий проект, і Гатрі написав код початкового прототипу під час різдвяних канікул 1997 року.

Початковий проект називався "XSP". Гатрі пояснив в інтерв'ю 2007 року, що «завжди запитують, що означає буква X. Тоді вона нічого не означала. XML починається з неї, XSLT починається із неї, все кльове починається з X, тому ми його так і назвали. Прототип XSP був розроблений на Java, так як на той момент у Microsoft не було Java-подібної технології. На той час вже передбачалося

(небезпідставно, як з'ясувалося надалі), що ліцензування Java для Microsoft не буде продовжено у 2003 році (у 2003 закінчувався термін виданої Sun Microsystems ліцензії). У 1999 компанією Microsoft було вирішено побудувати платформу із загальномовним середовищем виконання Common Language Runtime (CLR) та на її основі розвинути технології. У цій платформі, як і у Java, використовувалися програмування за принципами ООП, автоматичне керування пам'ятью (збірник сміттята) та інші можливості. Гатрі описав це рішення як "величезний ризик", оскільки успіх нової розробки був пов'язаний з успіхом CLR, яка, як і XSP, була на ранній стадії розробки.

Так, разом з появою в 2002 році .NET Framework з новими мовами програмування C# та Visual Basic .NET, що входять до її складу, була представлена нова версія ASP, яка була названа ASP.NET.

Оскільки ASP.NET ґрунтується на Common Language Runtime(CLR), яка є основою всіх програм Microsoft .NET, розробники можуть писати код для ASP.NET, використовуючи мови програмування, що входять до комплекту .NET Framework (C#, Visual Basic.NET, J# та JScript .NET).

Програмна модель ASP.NET ґрунтується на протоколі HTTP та використовує його правила взаємодії між сервером та браузером.

В ASP.NET підтримується кілька програмних моделей для створення веб-застосунків:

ASP.NET Web Forms – фреймворк для створення модульних веб-сторінок з компонентів із обробкою подій інтерфейсу користувача на стороні сервера;

ASP.NET MVC – фреймворк для створення веб-сторінок із використанням шаблону проектування *MVC*;

ASP.NET Web Pages – спрощений синтаксис для додавання динамічного коду та доступу до даних усередині HTML розмітки веб-сторінок;

ASP.NET Web API – фреймворк для створення веб-сервісів;

ASP.NET WebHooks – Реалізація шаблону*Webhook* для підписки на події та публікації подій через HTTP;

SignalR – фреймворк для обміну повідомленнями в реальному часі між клієнтом та сервером.

Переваги ASP.NET у порівняні з ASP:

- скомпільований код виконується швидше, а більшість помилок виявляється ще на стадії розробки;
- значно покращена обробка помилок у час виконання запущеної програми завдяки використанню блоків try..catch;
- користувацькі елементи управління (controls) дозволяють виокремлювати часто використовувані шаблони, такі як меню сайту;
- використання знайомих метафор з Windows-додатків, як-от елементи керування та події;
- розширений набір елементів управління та класів, що сприяє швидшій розробці додатків;
- ASP.NET підтримує багатомовність .NET, дозволяючи писати код сторінок у VB.NET, Delphi.NET, Visual C# тощо;
- можливість кешування цілої сторінки або її частин для підвищення продуктивності;
- можливість кешування даних, які використовуються на сторінці;
- відокремлення візуальної частини та бізнес-логіки в різних файлах («code behind»);
- вдосконалена модель обробки запитів;
- розширена подієва модель;
- Вдосконалена модель серверних елементів керування;
- наявність master-сторінок для визначення шаблонів оформлення;
- підтримка CRUD-операцій при роботі з таблицями через GridView;
- вбудована підтримка AJAX.

ASP.NET WebForms

Вже наприкінці 1990-х років стало очевидним, що Всесвітня павутина — це щось більше, ніж тимчасове захоплення. Але, на жаль, у компанії Microsoft виникла одна проблема - компанія витратила роки на створення інструментів, що дозволяли проектувати додатки для настільних систем у візуальному редакторі. Розробники звикли дотримуватись саме такого підходу у побудові додатків, а компаніям зовсім не хотілося оплачувати перепідготовку всього свого персоналу.

Взаємодія з елементами управління забезпечувалася за моделлю обробки подій. Кнопці на формі призначався обробник події, який виконував деяку приховану від очей дію, після чого в інтерфейсі користувача (UI, User Interface) відбувалися зміни.

Спільнота розробників звикла до цього методу розробки, і, що ще важливіше, він виявився надзвичайно ефективним. До цих пір не існує практично жодного інструменту, здатного зрівнятися за ефективністю з принципом генерування коду WYSIWYG (What You See Is What You Get - "Що бачиш, те й отримуєш"), що застосовується у Visual Basic, а пізніше і у конструкторі WinForms. Можливо, візуальний інтерфейс виходив не вражаючим, але якщо вашому продукту не потрібно нічого, крім сірих прямокутників на сірому фоні, розробка в Visual Basic 6 або конструкторі WinForms давала далеко не гірший результат.

Технологія ASP.NET WebForms стала спробою перенести цю ефективність на веб-розробку. У ній була реалізована аналогічна функціональність перетягування мишею для розміщення елементів управління по сітці та взаємодії з ними на рівні відкомпільованого коду на стороні сервера.

На жаль, модель зворотної передачі (postback), яка використовувалася для обробки взаємодій користувача з сервером, не враховувала фундаментальні відмінності між веб- та настільними додатками. Кожна зворотна передача даних та виконання дії на сервері потребували повного циклу HTTP 'запит-відповідь' через інтернет. Незважаючи на цей недолік, ASP.NET Web Forms став надзвичайно успішним продуктом. Багато сайтів, від корпоративних внутрішніх порталів до широко відомих веб-сайтів, працюють на його основі.

Незважаючи на ефективність розробки, взаємодії користувача в додатках WebForms не повністю відповідають вимогам до сучасних веб-додатків. При спробі створити абстракцію, яка б наближала веб-програмування до настільного, виникає суттєва проблема: інтернет та настільні системи — це різні середовища. Внаслідок цього, взаємодія в WebForms часто видається громіздкою та незручною. Хоча такий підхід може бути прийнятним для корпоративних інтранет-сайтів, ви вряд лише знайдете багато великих веб-сайтів, побудованих на основі WebForms.

Програмування з WebForms багато в чому нагадує спроби загнати круглу пробку в квадратний отвір – пробка входить, але залишаються зазори.

У WebForms розробники з Microsoft спробували приховати як особливості протоколу HTTP (з властивою йому відсутністю стану), так і мову HTML (яка на той була малознайома деяким розробникам) шляхом моделювання інтерфейсу користувача через ієрархію об'єктів, що представляють серверні елементи управління. Кожен елемент управління відстежує свій стан між запитами за допомогою механізму View State, візуалізуючи себе як HTML-розмітку та

автоматично зв'язуючи клієнтські події (наприклад, клацання на кнопках) з відповідним серверним кодом обробки.

Фактично WebForms - це гігантський рівень абстракції, спроектований для надання класичного керованого подіями графічного інтерфейсу користувача у веб-середовищі. Головною ідеєю було зробити веб-розробку схожою на розробку за допомогою Windows Forms. Розробники вже не мали справу з послідовностями незалежних запитів та відповідей HTTP, а замість цього могли мислити термінами інтерфейсу користувача, який підтримує стан. Такий підхід дав Microsoft можливість перемістити армію розробників настільних Windows-додатків у новий світ веб-додатків.

Втім, технологія з використанням традиційної технології ASP.NET Web Forms в принципі була чудовою, проте реальність виявилася складнішою.

Одним із головних недоліків ASP.NET Web Forms є ресурсоємність ViewState, яка вимагає передачі великих обсягів даних між клієнтом та сервером, призводячи до уповільнення відповіді та підвищення навантаження на канали передачі даних. Крім того, складний і крихкий життєвий цикл сторінки ускладнює подієву обробку, збільшуючи ризик помилок. Модель Web Forms створює хибне почуття поділу відповідальності, часто змушуючи розробників змішувати код відображення та бізнес-логіку, що робить архітектуру додатків тендітною та важкою для розуміння.

Веб-форми також обмежують контроль над HTML-розміткою. Через абстракції, що часто виявляються недостатньо гнучкими, розробники змушені вдаватися до низькорівневих маніпуляцій для досягнення бажаної поведінки, ускладнюючи тестування та підтримку додатків. Незважаючи на певні переваги Web Forms у ситуаціях, що вимагають швидкого результату, складність та висока залежність компонентів ускладнюють розробку та супровід веб-додатків.

1.2. Сучасний стан веб-розробки

Сучасний стан веб-розробки значно відрізняється від часів запровадження WebForms від Microsoft. Останніми роками відбувся значний розвиток у кількох ключових напрямках. Серед них — активне дотримання веб-стандартів та поширення використання стандарту HTML5, який розширив можливості веб-додатків, дозволяючи клієнтській стороні виконувати завдання, які раніше були

прерогативою сервера. Це сприяло зростанню значущості JavaScript-бібліотек, таких як jQuery, що зміцнило позиції стандартів як основи для створення насичених веб-додатків.

Суттєві зміни сталися і в архітектурі взаємодії між додатками через HTTP, де REST (Representational State Transfer) затьмарив традиційний SOAP, який був характерний для первинного підходу до веб-служб в ASP.NET.

Концепція REST передбачає використання стандартних HTTP-методів для керування ресурсами (URI), змінюючи парадигму взаємодії в інтернеті. Це забезпечує більш природний спосіб обробки даних у форматах JSON або XML для клієнтських технологій, таких як AJAX і мобільних додатків, але також ставить нові вимоги до обробки HTTP та URL, що було складно втілити в рамках ASP.NET WebForms."

Не тільки веб-розробка досягла зрілості, але й у цілому програмна індустрія зазнала зміщення у бік гнучких методологій. Це означає перехід від інтенсивного попереднього планування до адаптивних процесів виявлення та реагування на зміни у проектах розробки ПЗ. Гнучкі методології тісно пов'язані з набором методик та інструментів розробки, часто з відкритим кодом, які сприяють їх застосуванню.

Особливої уваги заслуговують такі підходи, як розробка через тестування (TDD) та розробка через тестування поведінки (BDD). Вони передбачають проектування ПЗ з вихідною орієнтацією на тести або специфікації поведінки, щоб в будь-який час можна було перевірити стабільність та коректність роботи програми. Однак, у контексті Web Forms модульне тестування було практично неможливе через їх монолітну структуру, а інструменти автоматизації для інтерфейсу користувача часто втрачають ефективність при зміні структури сторінок.

Спільнота відкритого сирцевого коду та незалежних розробників розробила ряд високоякісних інструментів для модульного тестування (NUnit, xUnit), імітації (Moq, Rhino Mocks), інверсії управління (Ninject, AutoFac), безперервної інтеграції (Cruise Control, TeamCity) та об'єктно-реляційного відображення (NHibernate, Subsonic). Але, знов таки, використання цих інструментів з проектами ASP.NET Web Forms ускладнено через монолітну архітектуру останніх.

1.3. Роль Ruby on Rails у розвитку технологій для розробки веб-додатків

У 2004 році платформа Ruby on Rails була тихим і непомітним продуктом з відкритим кодом від невідомого гравця. Однак дуже несподівано вона досягла популярності, змінивши самі правила веб-розробки. Це було пов'язано не з тим, що платформа Ruby on Rails запропонувала революційну технологію, а з тим, що вона зібрала існуючі інгредієнти і змішала їх настільки чудово, що змогла буквально "присоромити" платформи, що існувалиь на той час.

Платформа Ruby on Rails (або просто Rails, як її зазвичай називають) містила в собі архітектуру MVC (Model-View-Controller, Модель-Подання-Контролер). Застосування архітектури MVC та робота в гармонії з протоколом HTTP, впровадження угод замість обов'язкового конфігурування та інтеграція інструменту об'єктно-реляційного відображення (object-relational mapping - ORM) у ядро дозволило додаткам, заснованим на Rails без особливих зусиль завоювати відносно високу популярність.

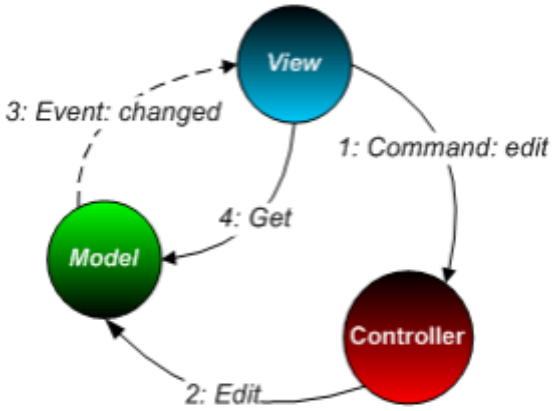
Це було схоже на відкриття того, якою має бути веб-розробка загалом. Платформа Rails показала, що дотримання веб-стандартів і відповідність REST не обов'язково повинні бути складним завданням, яке важко реалізувати. Вона також про-демонструвала, що гнучка розробка та розробка через тестування виявляються найбільш успішними, якщо сама інфраструктура спроектована для їхньої підтримки. З того часу інші представники світу веб-розробки почали надолужувати втрачене.

1.4. Архітектурний шаблон MVC

У жовтні 2007 року Microsoft анонсувала нову платформу веб-розробки - ASP.NET MVC, побудовану на основі ASP.NET і явно спроектовану безпосередньо у відповідь на розвиток технологій, подібних до Rails, а також як реакцію на критику WebForms.

Важливо розрізняти архітектурний шаблон MVC та інфраструктуру ASP.NET MVC. Шаблон MVC далеко не новий (його поява датується 1978 і пов'язана з проектом Smalltalk в Xerox PARC), але в наші дні він завоював величезну популярність як шаблон для веб-додатків з наведених нижче причин:

– взаємодія користувача в MVC відбувається за природним циклом: користувач здійснює дію, програма змінює свою модель даних і надсилає оновлене подання користувачу, а потім цикл повторюється. Це ідеально підходить для веб-додатків, які працюють за схемою запитів та відповідей через протокол HTTP:



– веб-додатки часто комбінують різні технології (наприклад, взаємодію з СУБД, HTML-розмітку тощо) і зазвичай поділяються на декілька шарів. Шаблони, що випливають з цього поділу, природно вписуються у концепцію MVC.

Інфраструктура ASP.NET MVC реалізує шаблон MVC, забезпечуючи значно кращий поділ відповідальності. Вона представляє сучасну версію MVC, яка особливо добре підходить для веб-додатків. Завдяки адаптації шаблону MVC, ASP.NET MVC створює серйозну конкуренцію платформам на кшталт Ruby on Rails, виводячи модель MVC на передову позицію у світі .NET. Узагальнивши досвід та найкращі практики, які виявили розробники на інших платформах, ASP.NET MVC нерідко перевершує навіть ті мож-ливості, які може запропонувати Rails.

1.5. Особливості архітектурного шаблону MVC

Якщо оперувати високорівневими поняттями, то архітектурний шаблон MVC передбачає, що код програми буде розділений принаймні на три частини:

– моделі, містять опис даних, з якими працюють користувачі; вони можуть бути простими, які лише представляють дані для передачі між поданнями та контролерами, або ж більш складними моделями предметної області, що включають бізнес-дані та логіку для їх обробки

− подання, застосовуються для візуалізації деякої частини моделі у вигляді інтерфейсу користувача.

− контролери, обробляють вхідні запити, виконують операції з моделлю і вибирають подання для візуалізації даних користувачеві.

Моделі у архітектурі MVC визначають 'всесвіт', в якому функціонує ваш додаток. Наприклад, у банківському додатку модель представляє всі аспекти банківської діяльності, які підтримуються додатком – це можуть бути розрахункові рахунки, головна бухгалтерська книга, кредитні ліміти для клієнтів, а також операції для маніпулювання даними в моделі, такі як внесення коштів на рахунки та їх списання. Модель також відповідає за збереження стану та цілісності даних, забезпечуючи, наприклад, що всі транзакції коректно відображені у головній книзі і що клієнти не знімають більше коштів, ніж мають право чи доступно в банку.

Важливо розуміти, що моделі не займаються візуалізацією інтерфейсу користувача або обробкою запитів – це завдання подань та контролерів в архітектурі MVC. Таким чином, моделі є відокремленими від інших частин додатку, зосереджуючись лише на бізнес-логіці та даних.

Подання (View) містять логіку, необхідну для відображення елементів моделі (тобто даних) користувачеві – і нічого більше. Вони не мають жодних прямих відомостей про модель та не обмінюються даними безпосередньо з нею.

В інфраструктурі ASP.NET MVC використовується механізм візуалізації, який відіграє ключову роль у обробці подань та створенні відповідей для браузера. У ранніх версіях MVC застосовувався стандартний механізм візуалізації ASP.NET, який обробляв aspx-сторінки, використовуючи модифікований синтаксис розмітки Web Forms. Починаючи з MVC версії 3, був введений механізм візуалізації Razor, який отримав подальші удосконалення у MVC версії 4 і зберіг свої основні характеристики у MVC версії 5, пропонуючи зовсім інший синтаксис.

Подання зазвичай використовують мову розмітки HTML, але вони не є html-сторінкою в загальному розумінні, а скоріше є програмним кодом з елементами HTML. Подання є файлами з розширенням .cshtml, які містять опис інтерфейсу користувача з використанням мови розмітки HTML та C#-подібної мови Razor.

В результаті компіляції файлу .cshtml спочатку генерується C#-клас, а потім цей клас компілюється.

Подання, зазвичай, відповідають методам (діям) контролерів і записуються у відповідні їм папки в каталозі Views. Наприклад, файли-подання для методів контролера Home, будуть знаходитись в проекті в папці Views/Home.

За потреби можна створити в каталозі Views папку з довільним іменем, де будуть зберігатись файли подань, не обов'язково пов'язані з певними методами контролерів.

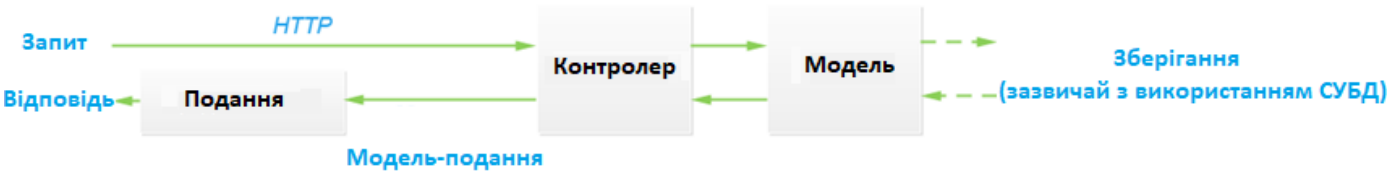
Для рендерингу (відправки) представлення з контролера у вихідний потік використовується метод View(). Якщо в цей метод не передається імені подання (не вказано відповідний параметр), то, за замовчуванням, використовується сторінка подання з тим же іменем, що і назва методу.

Середовище Visual Studio надає підтримку IntelliSense для механізму Razor, що спрощує роботу з поданнями та даними, переданими контролером.

Контролери є шлюзом між поданнями та моделлю – запити надходять від клієнта та обслуговуються контролером, який вибирає відповідне подання для відображення користувачеві та наповнює його даними, отримуючи їх з шару моделей, за потреби викликаючи відповідні методи.

В інфраструктурі MVC контролери – це звичайні класи C#, як правило, похідні від класу System.Web.Mvc.Controller. Кожен відкритий (public) метод в класі, похідному від Controller, називається методом дії (Action), який за допомогою системи маршрутизації ASP.NET зв’язується з певною URL-адресою. Коли надсилається запит за URL-адресою, пов'язаною з методом дії, виконується відповідний метод в класі контролера, де, в свою чергу, проводяться деякі операції над моделлю предметної області і потім обирається подання для відображення клієнту. Іншими словами маршрутизація пов’язує метод дії зі сторінкою, яка є результатом компіляції певного подання.

Взаємодії між контролером, моделлю та поданням відображено на малюнку нижче:



За замовчанням контролери знаходяться в папці Controllers в межах проекту. Коли браузер запитує стартову сторінку сайту, то отримує об’єкт подання, який

генерується методом Index контролера HomeController. За потреби, початкову сторінку можна змінити (редагуванням файлу web.config та відповідним налаштуванням маршрутизації).

Одним із способів передачі даних від контролера до подання є використання об'єкта ViewBag. Це динамічний об'єкт, якому можна присвоїти довільні значення. Вони доступні для будь-якого подання, яке може їх використовувати. Наприклад, @ViewBag.Message в стандартному файлі Index.cshtml використовується для вставки в макет сторінки тексту з властивості Message, визначеного у файлі HomeController.cs

Як бачимо кожна частина архітектури MVC є чітко визначеною та самодостатньою – це те, що називається поділом відповідальності. Логіка для керування даними міститься тільки в моделі, логіка для відображення даних – тільки у поданні, а код, який обробляє вхідні запити та формує відповіді – тільки в контролері. При чіткому розмежуванні всіх частин додаток буде легше супроводжувати та розширювати протягом його терміну існування, незалежно від того, наскільки великим та складним він стане.

1.6. Створення слабко пов'язаних компонентів

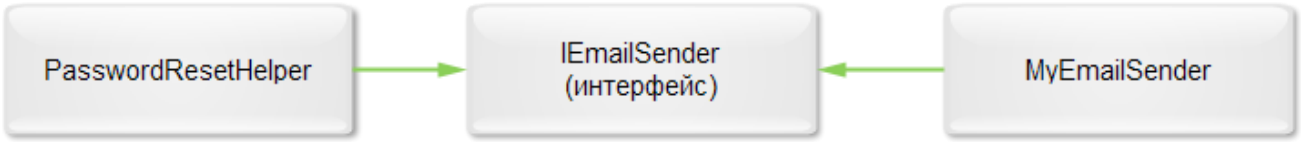
Однією з ключових переваг шаблону MVC є його здатність до розподілу відповідальності між компонентами. Важливо, щоб компоненти в додатках були якомога більш незалежними один від одного, з обмеженою кількістю взаємозалежностей, яка все ще піддається ефективному керуванню.

В ідеальному випадку кожен компонент не має знань про інші компоненти і взаємодіє з рештою частин додатку виключно через абстрактні інтерфейси. Така взаємодія називається слабким зв'язком, що спрощує процеси тестування та модифікації додатків.

Для кращого розуміння розглянемо простий приклад. Якщо ми створюємо компонент під назвою MyEmailSender для відправлення електронних листів, ми також реалізуємо інтерфейс IEmailSender, що визначає всі необхідні публічні методи для цього процесу.

Таким чином, будь-який інший компонент додатку, що потребує надсилання електронної пошти - наприклад, клас PasswordResetHelper для скидання пароля - використовує методи з цього інтерфейсу для відправлення повідомлень.

Це демонструє, що між PasswordResetHelper та MyEmailSender більше не існує прямої залежності, як показано на наступному малюнку:



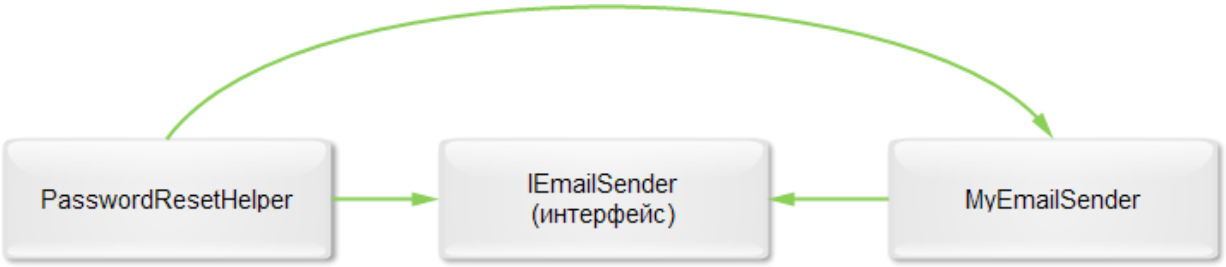
1.7. Використання впровадження залежностей

Інтерфейси ефективно допомагають роз'єднувати компоненти. Проте в мові C# виникає проблема: відсутність простого вбудованого механізму для створення об'єктів, які реалізують ці інтерфейси. Зазвичай, це робиться за допомогою ключового слова new, що призводить приблизно до наступного коду:

```
public interface IEmailSender {
    void SendEmail();
}

public class MyEmailSender : IEmailSender {
    public void SendEmail() { }
}

public class PasswordResetHelper {
    public void ResetPassword() {
        IEmailSender mySender = new MyEmailSender();
        // do required configuration here
        mySender.SendEmail();
    }
}
```

Цей підхід частково порушує ідею слабко пов'язаних компонентів, оскільки клас PasswordResetHelper залишається прив'язаним до конкретної реалізації MyEmailSender, що обмежує гнучкість заміни компонентів.

PasswordResetHelper, хоч і використовує інтерфейс IEmailSender для відправлення електронної пошти, він залежить від створення конкретного екземпляра MyEmailSender, що погіршує ситуацію. Тепер цей клас залежить як від конкретної реалізації MyEmailSender так і самого інтерфейсу IEmailSender.



Отже нам потрібен механізм, який дозволить отримувати об'єкти, що реалізують певні інтерфейси, без необхідності їх безпосереднього створення. Рішенням цієї проблеми є впровадження залежностей (dependency injection - DI) або інверсія управління (inversion of control - IoC).

Dependency Injection – це патерн проектування, який підсилює концепцію слабкого зв'язку. Наведений далі опис впровадження залежностей може здатися не зовсім зрозумілим, але повірте — це важлива концепція, яка є центральною для ефективної розробки додатків MVC і може викликати чимало плутанини.

1.8. Розрив та оголошення залежностей

Шаблон впровадження залежностей (DI) включає два основних етапи. Перший крок полягає у видаленні прямих залежностей від конкретних класів у компонентах, таких як PasswordResetHelper. Це досягається за допомогою створення конструктора класу, який приймає реалізації потрібних інтерфейсів як параметри:

```
public class PasswordResetHelper {

    private IEmailSender emailSender;

    public PasswordResetHelper(IEmailSender emailSenderParam) {
        emailSender = emailSenderParam;
    }

    public void ResetPassword(){
         // do required configuration here
        emailSender.SendEmail();
    }
}
```

Таким чином, конструктор класу PasswordResetHelper оголошує залежність від інтерфейсу IEmailSender. Клас не може бути створений і використовуватися без передачі об'єкта, що реалізує IEmailSender. Завдяки цьому оголошенню залежності, PasswordResetHelper відмовляється від прямого знання про клас MyEmailSender, стаючи залежним тільки від абстракції, що надається інтерфейсом IEmailSender.

Отже, клас PasswordResetHelper тепер не має інформації та не переймається тим, як саме реалізовано інтерфейс IEmailSender.

1.9. Використання залежностей

Другий ключовий аспект шаблону DI — це реалізація та використання оголошених залежностей у класі PasswordResetHelper під час створення його екземплярів, звідси й назва "використання залежностей".

Цей процес передбачає визначення класу, який реалізує інтерфейс IEmailSender, створення об'єкта цього класу і передачу його як аргументу конструктору PasswordResetHelper. PasswordResetHelper оголошує свої залежності у своєму конструкторі, що є прикладом впровадження залежностей через конструктор. Інший спосіб оголошення залежностей — через відкриту властивість, відомий як впровадження залежностей через setter.

Залежність передається до PasswordResetHelper під час виконання програми. Це означає, що екземпляр класу, який реалізує IEmailSender, створюється і передається конструктору PasswordResetHelper при його ініціалізації. На етапі компіляції PasswordResetHelper не має прямих залежностей від конкретних класів, що реалізують інтерфейси, на які він покладається.

Завдяки тому, що DI використовується під час виконання, можна гнучко визначати, які конкретні реалізації інтерфейсів будуть застосовані під час запуску додатку. Це дає можливість обирати серед різних провайдерів електронної пошти або використовувати спеціалізовані імітаційні (mock) реалізації для автоматизованого естування. Таким чином, DI дозволяє створити гнучкі та здатні до налаштування зв'язки між компонентами, що і є основною метою такого підходу.

1.10. Використання контейнера для впровадження залежностей

З'ясувавши, як розривати залежності, постає питання: як створити екземпляр конкретної реалізації інтерфейсу, не створюючи залежностей в інших частинах програми? Іншими словами, як і раніше, необхідно мати десь у коді додатка оператори наступного виду:

```
IEmailSender sender = new MyEmailSender();
helper = new PasswordResetHelper(sender);
```

Рішенням є використання контейнера для впровадження залежностей, також знаного як контейнер інверсії управління (IoC).

Цей контейнер слугує посередником між залежностями, які оголошує клас (наприклад, PasswordResetHelper), і класами, що можуть бути використані для їх реалізації (наприклад, MyEmailSender). У контейнері DI реєструються інтерфейси та абстрактні типи, які використовує програма, із вказівкою на класи, які мають бути створені для задоволення цих залежностей. Так, для інтерфейсу IEmailSender ми реєструємо клас MyEmailSender як його реалізацію.

Коли у програмі потрібний об'єкт PasswordResetHelper, його створення запитується у контейнера DI. Контейнер знає, що PasswordResetHelper потребує IEmailSender, і використовує MyEmailSender як реалізацію цього інтерфейсу. Контейнер створює об'єкт MyEmailSender і передає його як аргумент при створенні PasswordResetHelper, готового до використання у додатку.

Важливою перевагою цього підходу є те, що ми відмовляємося від самостійного створення об'єктів за допомогою new, замість цього звертаючись до контейнера DI. Це може бути новим для тих, хто тільки починає працювати з DI, але інфраструктура MVC Framework надає засоби, які спрощують цей процес.

Сам контейнер DI вам не доведеться створювати самостійно – на ринку вже є численні відмінні реалізації з відкритим сирцевим кодом, однією з них є, наприклад Ninject.

Роль DI контейнера може здатися простою і тривіальною, але насправді це далеко не так – якісний контейнер DI, такий як Ninject, пропонує ряд корисних функціональних можливостей:

- **Розпізнавання ланцюжка залежностей**
  Коли запитується компонент із власними залежностями (наприклад, параметрами конструктора), контейнер задовольняє ці залежності. Наприклад, якщо конструктор класу MyEmailSender потребує реалізацію інтерфейсу INetworkTransport, контейнер DI створить відповідний екземпляр, передасть його конструктору MyEmailSender і поверне результат у вигляді IEmailSender..

- **Управління життєвим циклом об'єктів**
  Якщо компонент запитується більше одного разу, чи повинен щоразу видаватися той самий або зовсім новий екземпляр? Якісний контейнер DI дозволяє задавати життєвий цикл компонента, надаючи можливість вибору із заздалегідь визначених варіантів: єдиний екземпляр (один і той самий екземпляр у

всіх випадках), короткочасний екземпляр (новий екземпляр у кожному випадку), екземпляр на потік, екземпляр на HTTP-запит, екземпляр з пулу, тощо.

- **Налаштування значень параметрів конструктора**
  Якщо реалізація інтерфейсу INetworkTransport вимагає, наприклад, рядок на ім'я serverName, має бути можливість встановлення цього значення при налаштуванні контейнера DI. Це груба, але проста система налаштувань, яка позбавляє код від необхідності передавати рядки підключення, адреси серверів, тощо.

Розробка власного контейнера DI – чудовий спосіб зрозуміти, яким чином C# та .NET обробляє типи та рефлексію, і я рекомендую зайнятися таким проектом у дощові вихідні дні. Однак не піддавайтеся спокусі розгортати свій код у реальному проекті. Створення надійного, стійкого та високопродуктивного контейнера DI пов'язане з труднощами і для розробки вашого додатку краще віддати перевагу випробуваному та протестованому пакету. Є безліч доступних пакетів, тому ви, напевно, знайдете такий, який виявиться підходящим для вашого стилю розробки.

1.11. Модель предметної області

Найбільш важливою частиною додатку MVC є модель предметної області. Ця модель створюється за рахунок визначення реальних об'єктів, операцій та правил, що існують у даній галузі чи роду діяльності, які має підтримувати додаток; така сукупність і називається предметною областю.

Після цього створюється програмна реалізація предметної області – модель предметної області. Для цілей ASP.NET MVC модель предметної області - це набір типів C# (класів, структур тощо), які разом називаються типами предметної області. Операції з предметної області визначаються методами, визначеними у типах предметної області, а правила предметної області виражаються логікою, укладеною всередині цих методів або застосуванням атрибутів C#.

На основі типів моделі створюються екземпляри (об'єкти) моделі, які віповідають даним предметної області, обробляються контролером і надсилаються в подання.

Поширений спосіб відокремлення моделі предметної області від інших частин додатку ASP.NET MVC передбачає розміщення моделі в окрememому складанні C#. Такий підхід дозволяє створювати посилання на модель предметної

області з інших частин додатку, але не гарантуватиме відсутність будь-яких посилань у зворотному напрямку. Це особливо корисно у великомасштабних проектах.

Інфраструктура ASP.NET MVC не накладає обмежень на реалізацію моделі предметної галузі. Модель можна розробити з використанням звичайних об'єктів C# і реалізувати зберігання даних з будь-якою базою даних або інфраструктурою об'єктно-реляційного відображення, що підтримується .NET.

1.12. ASP.NET Core

З одного боку, ASP.NET Core є продовженням розвитку платформи ASP.NET. Але з іншого боку, це не просто черговий реліз. Випуск ASP.NET Core фактично означає революцію всієї платформи, її якісну зміну.

Розробка над платформою почалася ще у 2014 році. Тоді платформа умовно називалася ASP.NET vNext. У червні 2016 року вийшов перший реліз платформи. А в листопаді 2020 року вийшла версія ASP.NET Core 5.0. Поточна версія ASP.NET Core вийшла разом з релізом .NET версії 8 у листопаді 2023 року.

ASP.NET Core зараз є повністю opensource-фреймворком. Всі файли сирцевого коду фреймворку доступні у github репозиторії за адресою:

https://github.com/dotnet/aspnetcore/

ASP.NET Core працює поверх крос-платформного середовища .NET Core, і може бути розгорнута на основних популярних операційних системах: Windows, Mac OS, Linux. Таким чином, за допомогою ASP.NET Core ми можемо створювати крос-платформні додатки. І хоча Windows як середовище для розробки та розгортання додатку досі превалює, тепер вже ми не обмежені тільки цією операційною системою. Тобто ми можемо запускати веб-додатки не тільки на ОС Windows, а й на Linux та Mac OS. А для розгортання веб-додатків можна використовувати традиційний IIS, або крос-платформний веб-сервер Kestrel.

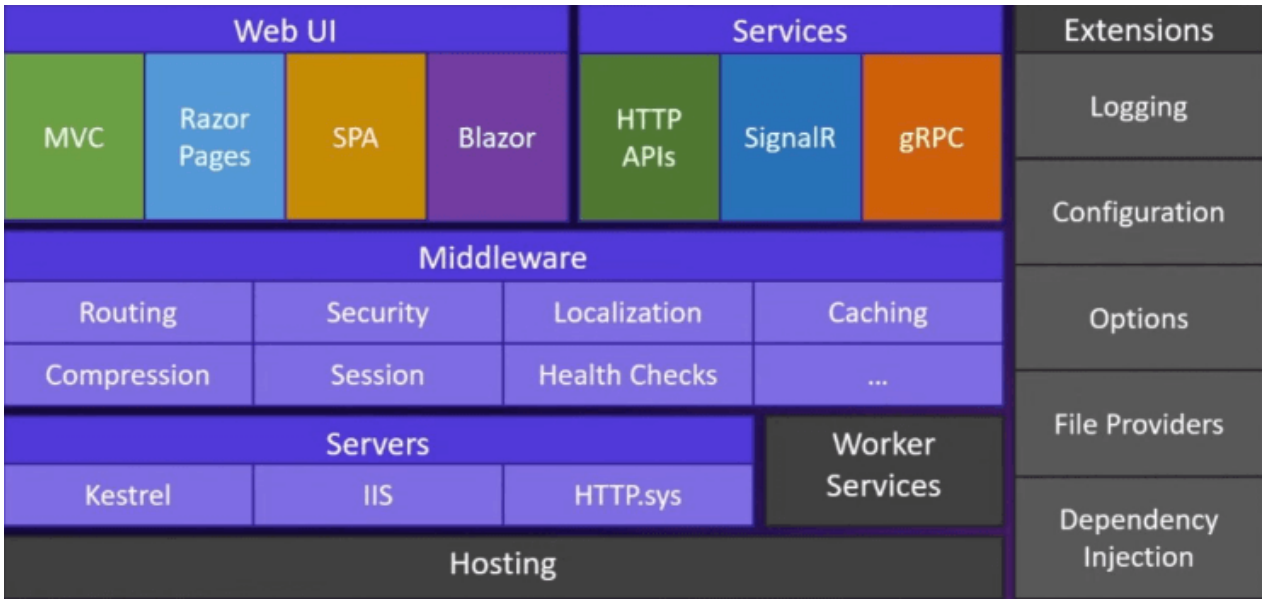
Завдяки модульності фреймворку всі необхідні компоненти веб-додатка можуть завантажуватися як окремі модулі через пакетний менеджер Nuget. Крім того, на відміну від попередніх версій платформи немає необхідності використовувати бібліотеку System.Web.dll.

ASP.NET Core включає в себе фреймворк MVC, який об'єднує функціональність MVC, Web API та Web Pages. У попередніх версіях платформи

ці технології реалізовувалися окремо і тому містили багато дублюючої функціональності. Зараз же вони об'єднані в одну програмну модель ASP.NET Core MVC. А Web Forms повністю пішли у минуле.

1.13. Архітектура та моделі розробки

Поточну архітектуру платформи ASP.NET Core можна уявити так:



На самому верхньому рівні розташовуються різні моделі взаємодії з користувачем. Це технології реалізації користувацького інтерфейсу та обробки введення користувача, такі як MVC, Razor Pages, SPA (Single Page Application - односторінкові додатки з використанням Angular, React, Vue) та Blazor. Крім того, це сервіси у вигляді вбудованих HTTP API, бібліотеки SignalR або сервісів GRPC.

Всі ці технології базуються та/або взаємодіють з чистим ASP.NET Core, який представлений перш за все різними вбудованими компонентами middleware, які застосовуються для обробки вхідних запитів та відповідей. Крім того, технології вищого рівня також взаємодіють з різними розширеннями, які не є безпосередньою частиною ASP.NET Core, такі як розширення для логування, конфігурації тощо.

І на самому нижньому рівні додаток ASP.NET Core працює в рамках деякого веб-сервера, наприклад, Kestrel, IIS, бібліотеки HTTP.sys.

Базовий ASP.NET Core підтримує всі основні моменти, необхідні для роботи сучасного веб-додатка: маршрутизація, конфігурація, логування, можливість роботи з різними системами баз даних тощо. В ASP.NET Core версії 6 у

фреймворк був доданий так званий Minimal API - мінімізована спрощена модель, яка ще спростила процес розробки коду додатка. Всі інші моделі розробки працюють поверх базового функціоналу ASP.NET Core.

ASP.NET Core MVC передбачає, як вже було зазначено вище, розробку додатка навколо трьох основних компонентів - Model (моделі), View (представлення) та Controller (контролери), де моделі відповідають за роботу з даними, контролери представляють логіку обробки запитів, а подання визначають візуальну складову.

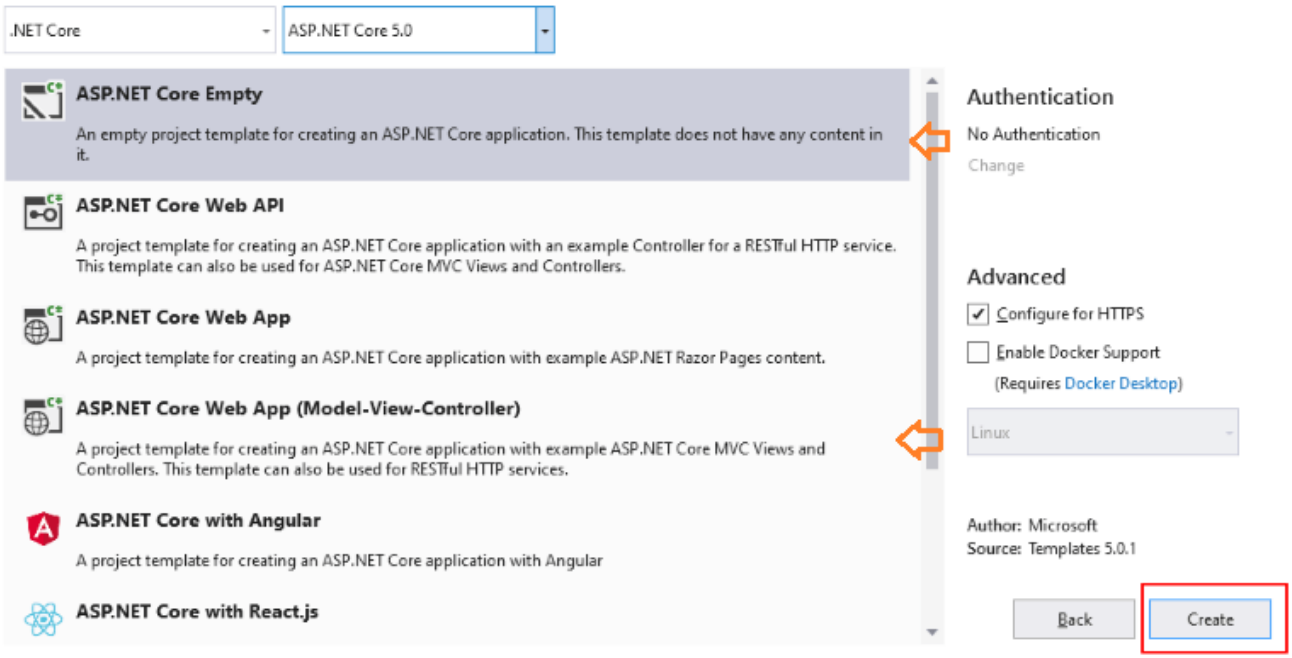
Razor Pages являє собою модель, при якій за обробку запиту відповідають спеціальні сутності - сторінки Razor Pages. Кожну окрему таку сутність можна асоціювати з окремою веб-сторінкою.

ASP.NET Core Web API представляє реалізацію патерну REST, при якому для кожного типу http-запиту (GET, POST, PUT, DELETE) призначений окремий ресурс. Подібні ресурси визначаються у вигляді методів контролера Web API. Дана модель особливо підходить для односторінкових додатків, але не тільки. Web API буде розглянуто в лабораторній роботі №5

Blazor представляє фреймворк, який дозволяє створювати інтерактивні додатки як на стороні сервера, так і на стороні клієнта і дозволяє задіяти на рівні браузера низькорівневий код WebAssembly.

1.14. Шаблони веб-додатків

При створенні нового проекту ASP.NET Core, вам надаються декілька базових варіантів для початку роботи: шаблон "Empty" (Пустий), шаблон "MVC" та інші. Ці назви можуть здатися трохи вводячими в оману, оскільки додати необхідні базові папки та залежності в пустий проект можна буде і надалі власноруч.



Це надає гнучкість у процесі створення проектів у Microsoft Visual Studio з .NET Core, дозволяючи вибирати між простим базовим шаблоном, який може бути розширений згодом, та більш комплексним шаблоном, який вже містить усі необхідні елементи для швидкого початку розробки.

Реальна відмінність між цими шаблонами полягає у додатковому вмісті, який генерується автоматично при виборі відповідного шаблону. Наприклад шаблон "MVC" пропонує готову точку відправлення, що включає стандартні контролери та представлення, конфігурацію безпеки, набір популярних пакетів JavaScript та CSS (наприклад, jQuery та Bootstrap), а також макет, який використовує бібліотеку Bootstrap для створення теми, яка формує користувацький інтерфейс додатку.

У той час як при виборі варіанту "Empty" (Пустий) створений проект буде містити лише базові посилання, необхідні для коректної роботи, та базову структуру папок.

Додаткові файли, які додаються в проект за допомогою шаблону "MVC", насправді є більш значущими, ніж може здатися на перший погляд. Деякі з цих файлів відносяться до аутентифікації, тоді як інші представляють собою файли JavaScript та CSS, для яких передбачені як стандартні, так і мінімізовані версії.

Як вже зазначалось раніше, середовище Visual Studio збирає проект, створений під платформу .NET Core, використовуючи пакети NuGet. Це означає, що ви можете переглянути, які саме пакети будуть використані, вибравши пункт "Manage NuGet Packages for Solution" (Управління пакетами NuGet для рішення) у

меню "Tools" => "Library Package Manager" (Сервіс => Диспетчер бібліотечних пакетів). Це також дає можливість додавати ті ж самі пакети у будь-який інший проект, включаючи ті, що створені з використанням шаблону "Empty".

Незалежно від обраного шаблону, ви помітите, що створені проекти мають дуже схожі структури папок.

1.15. Структура проекту MVC .NET Core

Стандартний проект містить наступні розділи та компоненти:

/Connected Services: підключені хмарні служби;

/Properties: містить файл налаштувань для запуску проекту через Visual Studio;

/wwwroot: вузол (на диску йому відповідає однойменна папка) призначений для зберігання статичних файлів - зображень, скриптів javascript, файлів css, тощо, які використовуються додатком; мета існування цієї папки в проекті - розмежування доступу до статичних файлів, до яких дозволений доступ з боку клієнта і до яких безпосередній доступ заборонений; це є необов'язковою конвенцією, статичний вміст може зберігатися в будь-якому зручному місці.

/References: всі додані в проект пакети та бібліотеки;

/Controllers: у цій папці розміщуються класи контролерів; це конвенція, класи контролерів можуть розташовуватися будь-де у межах проекту;

/Models: у цій папці розміщуються класи моделей представлень та моделей предметної області, хоча для всього, окрім найпростіших застосунків, краще визначати модель предметної області в окремому проекті; це конвенція, класи моделей можуть бути визначені де завгодно в поточному проекті або взагалі винесені в окремий проект;

/Views: у цій папці зберігаються подання та часткові подання, зазвичай згруповані разом у папках з назвами контролерів, з якими вони пов'язані.

/Views/Shared: У цій папці зберігаються макети та представлення, які не є специфічними для будь-якого контролера. У підкаталозі Views\Shared розташований файл _Layout.cshtml, який використовується для редагування макета сторінки застосування.

/appsettings.json: зберігає конфігурацію додатку.

/bower.json: файл, який керує клієнтськими залежностями (бібліотеки javascript і css), які підключаються через менеджер пакетів Bower.

/bundleconfig.json: файл, який містить завдання по мініфікації скриптів і стилів, які виконуються при побудові проекту.

/Program.cs: є точкою входу в додаток. Він містить метод Main, який ініціалізує та запускає веб-додаток. В ASP.NET Core до версії 5, Program.cs використовується для створення та конфігурації хосту додатку, використовуючи WebHostBuilder або HostBuilder, залежно від типу додатку.

/Startup.cs: містить код для налаштування служб додатку та його конвеєра запитів, використовуючи методи ConfigureServices та Configure. Цей файл використовується для додавання компонентів, які обробляють запити (наприклад, MVC, статичні файли, Identity) та конфігурації міжпрограмної взаємодії (middleware).

/bin: У цю папку поміщається скомпільована збірка додатку разом з усіма залежностями, які не знаходяться в GAC. Щоб побачити папку bin у вікні Solution Explorer, потрібно натиснути на кнопку "Show All Files" (Показати всі файли). Оскільки всі файли у цій папці є двійковими і генеруються в результаті компіляції, вони зазвичай не зберігаються у системі керування сирцевим кодом.

Примітка: у версіях ASP.NET Core 6 та 7 (порявняно з версією 5) структура веб-додатків зазнала змін. Головною відмінністю є відсутність файлу Startup.cs. Замість цього, всі налаштування, які раніше знаходились у Startup.cs, тепер інтегровані безпосередньо в Program.cs.

1.16. Запуск програми. Класи Program та Startup

У будь-якому типі проектів ASP.NET Core, як і в проекті консольного типу додатку, ми можемо знайти файл Program.cs, в якому визначений однойменний клас `Program` і з якого по суті починається виконання додатку. У ASP.NET Core 5 цей файл виглядає так:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;
```

```
namespace HelloApp
{
    public class Program
    {
        public static void Main(string[] args) {
            CreateHostBuilder(args).Build().Run();
        }

        public static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureWebHostDefaults(webBuilder =>
                {
                    webBuilder.UseStartup<Startup>();
                });
    }
}
```

Щоб запустити додаток ASP.NET Core, необхідно створити об'єкт, що реалізує інтерфейс IHost, в рамках якого розгортається веб-додаток. Для створення такого об'єкту застосовується об'єкт, що реалізує інтерфейс IHostBuilder.

У програмі за замовчуванням у статичному методі CreateHostBuilder саме створюється і налаштовується об'єкт IHostBuilder. Безпосередньо створення об'єкту IHostBuilder відбувається за допомогою методу Host.CreateDefaultBuilder(args).

Цей метод виконує ряд завдань:

– Встановлює кореневий каталог (для цього використовується властивість Directory.GetCurrentDirectory). Кореневий каталог – це папка, де буде здійснюватися пошук файлів різноманітного вмісту.

– Встановлює конфігурацію хоста. Для цього завантажуються змінні середовища з префіксом "DOTNET_" та аргументи командного рядка.

– Встановлює конфігурацію додатку. Для цього завантажується вміст з файлів appsettings.json та appsettings.{Environment}.json, а також змінні середовища та аргументи командного рядка. Якщо додаток перебуває у статусі розробки, то також використовуються дані Secret Manager (менеджера секретів), який дозволяє зберігати конфіденційні дані, що використовуються під час розробки.

– Додає провайдери логування.

– Якщо проект перебуває у статусі розробки, забезпечує валідацію сервісів.

Далі викликається метод ConfigureWebHostDefaults(). Цей метод покликаний виконувати конфігурацію параметрів хоста, а саме:

– Завантажує конфігурацію з змінних середовища з префіксом "ASPNETCORE_".

– Запускає та налаштовує веб-сервер Kestrel, в рамках якого буде розгортатися додаток.

– Додає компонент Host Filtering, який дозволяє налаштовувати адреси для веб-сервера Kestrel.

– Якщо змінна середовища ASPNETCORE_FORWARDEDHEADERS_ENABLED дорівнює true, додає компонент Forwarded Headers, який дозволяє зчитувати з запиту заголовки "X-Forwarded-".

Якщо додатку потрібен IIS, то цей метод також забезпечує інтеграцію з IIS.

Метод ConfigureWebHostDefaults() в якості параметра приймає делегат Action<IWebHostBuilder>. За допомогою послідовного виклику ланцюжка методів у об'єкта IWebHostBuilder здійснюється ініціалізація веб-сервера для розгортання веб-додатка. Зокрема, в даному випадку у IWebHostBuilder викликається метод UseStartup():

```
webBuilder.UseStartup<Startup>()
```

Цим викликом встановлюється стартовий клас додатка - клас Startup, з якого і починається обробка вхідних запитів.

У методі Main викликається метод у створеного об'єкта IHostBuilder – метод Build(), який власне і створює хост - об'єкт IHost, в рамках якого розгортається веб-додаток. А потім для безпосереднього запуску додатку викликається метод Run:

```
CreateHostBuilder(args).Build().Run();
```

Після цього додаток запущено, і веб-сервер починає прослуховувати всі вхідні HTTP-запити.

Примітка: У .NET 6 та новіших версіях процес запуску додатку відрізняється від попередніх версій. Однією з ключових змін є відсутність файлу Startup.cs та зміна у підході до визначення класу Program та методу Main.

В .NET 6 та новіших версіях використовується концепція "top-level statements", яка дозволяє визначати код запуску додатку без явного визначення класу Program та методу Main. Тобто, весь код, який раніше розміщувався у методі Main, тепер може бути розміщений безпосередньо на верхньому рівні файлу

Program.cs. Це робить структуру проекту більш лаконічною та зрозумілою, особливо для нових розробників.

Конфігурація, яка раніше визначалася у класі Startup, тепер інтегрована безпосередньо у код на верхньому рівні в Program.cs. Такий підхід спрощує процес конфігурації та зменшує кількість шаблонного коду, який потрібно писати при створенні нового проекту.

Приклад файлу Program.cs для .NET 6 та новіших версій:

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();
app.MapGet("/", () => "Hello World!");
app.Run();
```

Клас Startup є вхідною точкою в додаток ASP.NET Core. Цей клас проводить конфігурацію додатка, налаштовує сервіси, які додаток буде використовувати, встановлює middleware-компоненти які будуть використовуватись для обробки запитів. Якщо ми звернемося до файлу Program.cs, то там є наступні рядки коду:

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder => {
            webBuilder.UseStartup<Startup>();
         });
```

Метод webBuilder.UseStartup<Startup>() встановлює клас Startup як стартовий. І при запуску додатка середовище ASP.NET буде шукати в збірці додатка клас з ім'ям Startup і завантажувати його. Однак, не обов'язково, щоб клас називався саме як Startup; ми можемо змінити відповідний виклик у файлі Program.cs наприклад так:

```
      webBuilder.UseStartup<Proccessor>()
```

Тепер середовище буде шукати при запуску додатка клас Proccessor. І в цьому випадку нам потрібно буде визначити в проекті клас з ім'ям Proccessor, який буде аналогічний файлу Startup.cs.

Клас Startup має визначати метод Configure(), і також опціонально в Startup можна визначити конструктор класу та метод ConfigureServices().

При запуску додатка спочатку спрацьовує конструктор, потім викликається метод ConfigureServices() і на останок – метод Configure(). Ці методи викликаються середовищем виконання ASP.NET.

У проекті ASP.NET Core створеного за шаблоном MVC клас Startup виглядає так:

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.HttpsPolicy;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace WebApplication1 {
    public class Startup {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }
        public IConfiguration Configuration { get; }
        // This method gets called by the runtime.
        // Use this method to add services to the container.
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddControllersWithViews();
        }
```

(продовження)

```
        // This method gets called by the runtime.
        // Use this method to configure the HTTP request pipeline.
        public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
        {
            if (env.IsDevelopment()) {
                app.UseDeveloperExceptionPage();
            }
            else {
                app.UseExceptionHandler("/Home/Error");
                // The default HSTS value is 30 days.
                // You may want to change this for production scenarios,
                //    see https://aka.ms/aspnetcore-hsts.
                app.UseHsts();
            }
            app.UseHttpsRedirection();
            app.UseStaticFiles();
            app.UseRouting();
            app.UseAuthorization();
            app.UseEndpoints(endpoints => {
                endpoints.MapControllerRoute(
                    name: "default",
                    pattern: "{controller=Home}/{action=Index}/{id?}");
            });
        }
    }}
```

1.17. Метод ConfigureServices()

Необов'язковий метод ConfigureServices() реєструє сервіси, які використовуються додатком. В якості параметра він приймає об'єкт IServiceCollection, який і представляє колекцію сервісів у додатку. За допомогою методів розширень цього об'єкта відбувається конфігурація додатку для використання сервісів. Всі методи мають форму Add[назва_сервісу].

У проекті, створеному за шаблоном типу Empty цей метод не виконує жодних дій:

```
public void ConfigureServices(IServiceCollection services) {
}
```

А, наприклад, у проекті, створеному за шаблоном ASP.NET Core Web App MVC цей метод має наступне визначення:

```
public void ConfigureServices(IServiceCollection services) {
   services.AddControllersWithViews();
}
```

Виклик методу services.AddControllersWithViews() додає в колекцію сервісів сервіси, які необхідні для роботи контролерів MVC. Після додавання в колекцію сервісів додані сервіси стають доступними для додатку. Як правило, вбудовані методи, які додають вбудовані сервіси, починаються з префікса Add, наприклад, AddControllersWithViews().

1.18. Метод Configure()

Метод Configure встановлює, як саме додаток буде обробляти запити. Цей метод є обов'язковим. Для встановлення middleware-компонентів, які оброблятимуть запити, використовуються методи об'єкта IApplicationBuilder. Об'єкт IApplicationBuilder є обов'язковим параметром для методу Configure.

Крім того, метод часто приймає ще один необов'язковий параметр – об'єкт IWebHostEnvironment, який дозволяє отримати інформацію про середовище, в якому запускається додаток, і взаємодіяти з ним.

Проте по суті в метод Configure як параметр може передаватися будь-який сервіс, який зареєстрований у методі ConfigureServices або який реєструється для додатку за замовчуванням (наприклад, IWebHostEnvironment).

Наприклад, наступний код метода Configure() у проекті, створеному за шаблоном типу Empty безпосередньо обробляє запити:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    // якщо додаток у стані розробки
    if (env.IsDevelopment()) {
        // то друкуємо детальну інформацію про помилку, при наявності помилки
        app.UseDeveloperExceptionPage();
    }

    app.UseRouting();  // додаємо можливості маршрутизації

    // встановлюємо адреси, які будуть оброблятися
    app.UseEndpoints(endpoints => {
        // обробка запиту - отримуємо контекст запиту у вигляді об'єкта context
        endpoints.MapGet("/", async context => {
            // відправка відповіді у вигляді рядка "Hello World!"
            await context.Response.WriteAsync("Hello World!");
        });
    });
}
```

Розглянемо по кроках, що робить цей метод:

Спочатку перевіряється, чи знаходиться додаток у стані/статусі розробки. Що це означає? Для проекту можна вказати, наприклад, через налаштування, що він знаходиться в стані розробки. Загалом умовно є три стани або стадії проекту: у стані розробки (Development), у стані підготовки до розгортання (Staging) і у стані повноцінного використання (Production), коли він уже розгорнутий на якомусь сервері, і користувачі можуть до нього звертатися. За замовчуванням Visual Studio встановлює для проекту стан розробки. І даний вираз саме перевіряє цей стан. Якщо проект знаходиться в стані розробки, то, можливо, ми захочемо застосувати деякі дії, які не потрібні, коли додаток вже розгорнуто. Так, за замовчуванням викликається метод app.UseDeveloperExceptionPage() який виводить детальні повідомлення про помилки. Подібні повідомлення небажані і можуть розкривати деякі чутливі деталі реалізації, коли додаток вже розгорнуто на сервері і з ним можуть працювати користувачі, тому подібні повідомлення за замовчуванням виводяться тільки в стані розробки.

Виклик app.UseRouting() додає деякі можливості маршрутизації, завдяки чому додаток може відповідати запитам з певними маршрутами.

Далі йде виклик app.UseEndpoints, який дозволяє визначити маршрути, які будуть оброблятися додатком.

Ланцюжок викликів завершується виразом endpoints.MapGet який вказує, що для всіх запитів за маршрутом "/" (тобто до кореня веб-додатка) у відповідь буде відправлятися рядок "Hello World!".

1.19. Макет на базі майстер-сторінки

Коли у проекті багато сторінок, і всі вони містять якісь спільні елементи, то замість того, щоб повторювати всі ці елементи у кожному поданні, набагато зручніше визначити один загальний шаблон. У цьому випадку при зміні якихось спільних елементів буде достатньо внести зміни у загальному шаблоні, не змінюючи всі інші подання. В ASP.NET MVC таким шаблоном є майстер-сторінки.

Майстер-сторінки застосовуються для створення однорідного, уніфікованого вигляду сайту. По суті майстер-сторінки – це ті ж самі подання, які можуть включати в себе інші подання. Наприклад, можна визначити на майстер-сторінці спільне для всіх інших сторінок меню, а також підключити спільні стилі та скрипти. В результаті нам не доведеться на кожному окремому представленні

прописувати шлях до файлів стилів, а потім при необхідності його змінювати. А спеціальні теги дозволяють вставляти в певне місце на майстер-сторінках інші подання.

За замовчуванням при створенні нового проекту за шаблоном ASP.NET MVC Core у проект додається майстер-сторінка під назвою _Layout.cshtml, яку можна знайти в каталозі Views/Shared.

Код майстер-сторінки нагадує повноцінну веб-сторінку: тут присутні основні теги <html>, <head>, <body> тощо. І також тут можуть використовуватися конструкції Razor. Фактично це те ж саме подання. Основна ж відмінність від звичайних сторінок полягає у використанні методу @RenderBody(), який є плейсхолдером і на місце якого потім будуть підставлятися інші подання, які використовують дану майстер-сторінку. В результаті ми зможемо легко встановити для всіх сторінок веб-додатку єдиний стиль оформлення.

В поданні кожної сторінки є властивість Layout, яка зберігає посилання на майстер-сторінку. За замовчуванням, для задання трансляторові назви майстер-сторінки в @RenderBody() якої має розміщуватись подання даної, використовують файл _ViewStart.cshtml, який знаходиться в папці Views . Код цього файлу автоматично додається в початок коду подання конкретної сторінки після її запуску. При цьому файли подань, до яких застосовується _ViewStart.cshtml, повинні знаходитися з цим файлом у одному каталозі.

Файл _ViewStart.cshtml містить наступний код:

```
@{
    Layout = "_Layout";
}
```

В прикладі вказано, що майстер-сторінкою, за замовчуванням буде файл _Layout.cshtml, розширення можна не використовувати.

Під час рендеринга сторінки (тобто її формування для відправки клієнту), система буде шукати майстер сторінку _Layout за наступними шляхами:

/Views/[Назва_контролера]/_Layout.cshtml
/Views/Shared/_Layout.cshtml

Якщо раптом ми забажаємо глобально по всьому проекту змінити майстер-сторінку на інший файл, який розташований в якійсь іншій папці, наприклад, в корені каталогу Views, то нам потрібно використовувати повний шлях до файлу:

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

1.20. Бандлінг та мініфікація

У веб-додатках для оптимізації завантаження css-стилів та скриптів використовується два механізми: бандлінг та мініфікація .

Бандлінг (bandling) здійснює з'єднання скриптів або стилів в один загальний файл або бандл. Мініфікація (minification) зменшує розмір файлу скриптів або стилів за рахунок видалення форматування коду та коментарів. Для виконання операцій бандлінга і мініфікації в ASP.NET Core використовується розширення BundlerMinifier[4].

Файл bundleconfig.json проекту містить декілька параметрів.

Параметр outputFileName задає шлях до вихідного файлу-результату об'єднання файлів, вказаних в параметрі inputFiles. Файли, які будуть об'єднуватися, розділяються комою. З файлу wwwroot/js/site.js, наприклад, буде формуватися файл wwwroot/js/site.min.js . Додатковий параметр minify вказує, чи будуть мінізуватись файли, які включаються в бандл. Значення enabled:true включає мініфікацію. А значення renameLocals:true дозволяє скоротити імена локальних змінних. Останній параметр sourceMap вказує, чи треба генерувати файл-карту, яка визначатиме зв’язок сирцевого та вихідного файлів.

1.21. Угоди у MVC

У проекті MVC застосовуються два типи угод.

Угоди першого типу – це припущення про те, як може виглядати структура проекту.

Наприклад, прийнято розміщувати файли JavaScript у папці wwwroot\js. Тут їх розраховують виявити інші розробники, які використовують MVC, і сюди встановлюватимуться пакети NuGet. Однак ви можете перейменувати цю папку або взагалі видалити її - розмістивши файли сценаріїв десь в іншому місці. Це не завадить інфраструктурі MVC запустити вашу програму за умови, що елементи <script> всередині подань посилатимуться на місце розташування, в якому знаходяться файли сценаріїв.

Угоди другого типу виростають із принципу угоди щодо конфігурації (convention over configuration) або угоди над конфігурацією, якщо наголошувати на перевазі угоди перед конфігурацією, яка була одним з головних аспектів, що забезпечили популярність платформі **Ruby on Rails**.

Угода конфігурації означає, що Ви не повинні явно конфігурувати, наприклад, асоціації між контролерами та відповідними поданнями. Потрібно просто дотримуватись певної угоди про назву для файлів - і все буде працювати. За угоди такого роду знижується гнучкість у зміні структури проекту. У наступних розділах пояснюються угоди, які застосовуються замість конфігурації.

Дотримання угод для класів контролерів

Класи контролерів повинні мати імена, що закінчуються на слово Controller, наприклад, ProductController, ShopController. При посиланні на контролер десь у проекті, скажімо, при виклику допоміжного методу HTML, вказується лише перша частина імені (така як Product), а інфраструктура MVC автоматично додає до цього імені слово Controller і починає пошук класу контролера.

Цю поведінку можна змінити, створивши власну реалізацію інтерфейсу IControllerFactory.

Дотримання угод для подань

Подання та часткові подання розміщуються в папці:
/Views/Ім'я_Контролера

Наприклад, подання, асоційоване з класом ProductController, знаходиться у папці:
/Views/Product

Зверніть увагу, що частина Controller імені класу в імені папки всередині Views не вказується, тобто, використовується папка /Views/Product, а не /Views/ProductController. Спочатку такий підхід може здатися нелогічним, але це незабаром увійде у звичку.

Інфраструктура MVC очікує, що стандартне подання для методу дії повинно мати ім'я цього методу.

Наприклад, подання, асоційоване з методом дії List(), має називатися List.cshtml.

Таким чином, очікується, що для методу дії List() в класі ProductController стандартним поданням буде /Views/Product/List.cshtml.

Стандартне подання використовується при поверненні результату виклику методу View() у методі дії, приблизно так:

```
return View();
```

Можна вказати ім'я іншого подання:

```
return View("MyOtherView");
```

Зверніть увагу, що ми не включаємо при вказуванні подання, а ні розширення імені файлу, а ні шлях до файлу. При пошуку MVC Framework переглядає папку, що має ім'я контролера, а потім папку /Views/Shared. Це означає, що подання, які застосовуються більш ніж одним контролером, можна помістити в папку /Views/Shared та інфраструк-тура знайде їх самостійно**.**

Дотримання угод для майстер-сторінок

Угода про іменування для майстер-сторінок передбачає додавання до імені файлу символу підкреслення_, а самі файли компонування поміщаються в папку /Views/Shared.

Середовище Visual Studio створює компонування на ім'я Layout.cshtml для всіх шаблонів проектів, крім Empty. Це стандартне компонування застосовується до всіх стандартних подань через файл /Views/_ViewStart.cshtml.

Якщо ви не хочете, щоб стандартне компонування застосовувалося до подань, можете змінити налаштування у файлі _ViewStart.cshtml (або взагалі видалити цей файл), вказавши інше компонування у поданні, наприклад:

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

Або ж можна вимкнути компонування для заданого подання:

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

Найкращий спосіб оцінки інфраструктури, призначеної для розробки програмного забезпечення, полягає в тому, щоб приступити безпосередньо до її використання.

При виконанні цієї лабраторної роботи Ви створите простий додаток із застосуванням ASP.NET MVC Framework.

Далі буде покроково описаний цей процес, щоб Ви зрозуміли, як будується додаток ASP.NET MVC.

Для спрощення деякі технічні подробиці будуть, поки що, пропущені. Але не турбуйтеся – якщо ви тільки починаєте знайомство з MVC, то дізнаєтесь багато цікавого.

ХІД РОБОТИ

2.1 Встановлення MS SQL Server + MS SQL Server Management Studio (SSMS)

MS SQL *Server* доступний у різних варіаціях. Насамперед, це *MS SQL Server Enterprise* - повний випуск, орієнтований на використання у реальних проектах. Саме він використовується на різних хостингах та серверах баз даних. Однак він доступний тільки у платній версії (не рахуючи триального періоду) і коштує досить пристойних грошей.

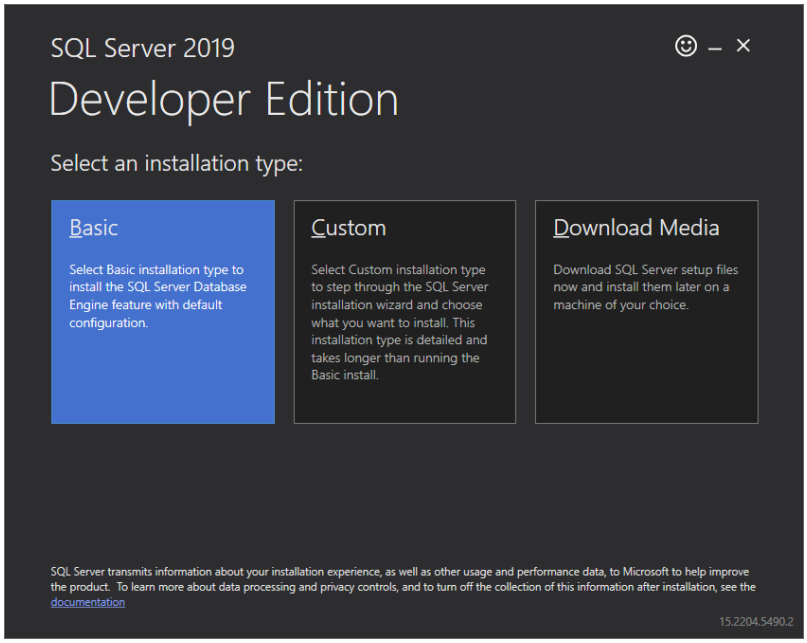
Для простих додатків може вистачити версії *Express*: вона безкоштовна. *MS SQL Server Express* можна ставити як реальний сервер і використовувати в реальних проектах, однак слід пам'ятати що він має урізаний функціонал порівняно з повною версією.

Також є *MS SQL Server Developer Edition*. Це повнофункціональний випуск, який містить весь функціонал, як і повна версія *MS SQL Server Enterprise*, але націлена тільки на потреби розробки. Ця версія не може бути використана для розгортання як реальний сервер на реальних проектах. Проте для вивчення всього функціоналу *MS SQL Server* ця версія представляє оптимальний варіант. Для виконання цієї лабораторної роботи можна використовувати як *Developer*, так і Express випуск.

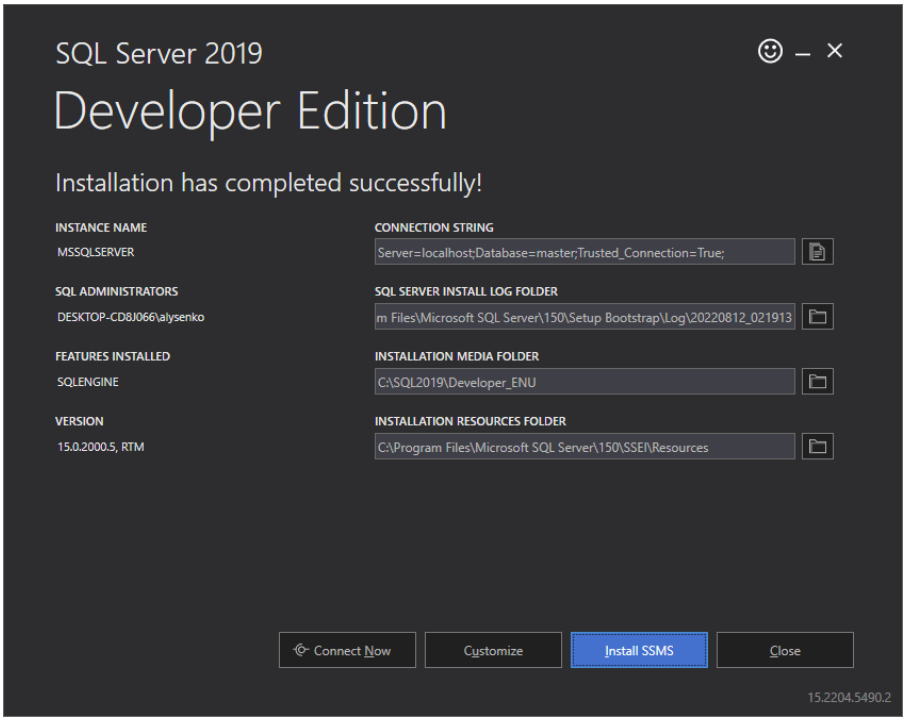
Програма-установник доступна за адресою:
https://www.microsoft.com/en-us/sql-server/sql-server-downloads

Знайдіть на сторінці пункт «Developer» та натисніть на кнопку завантаження.

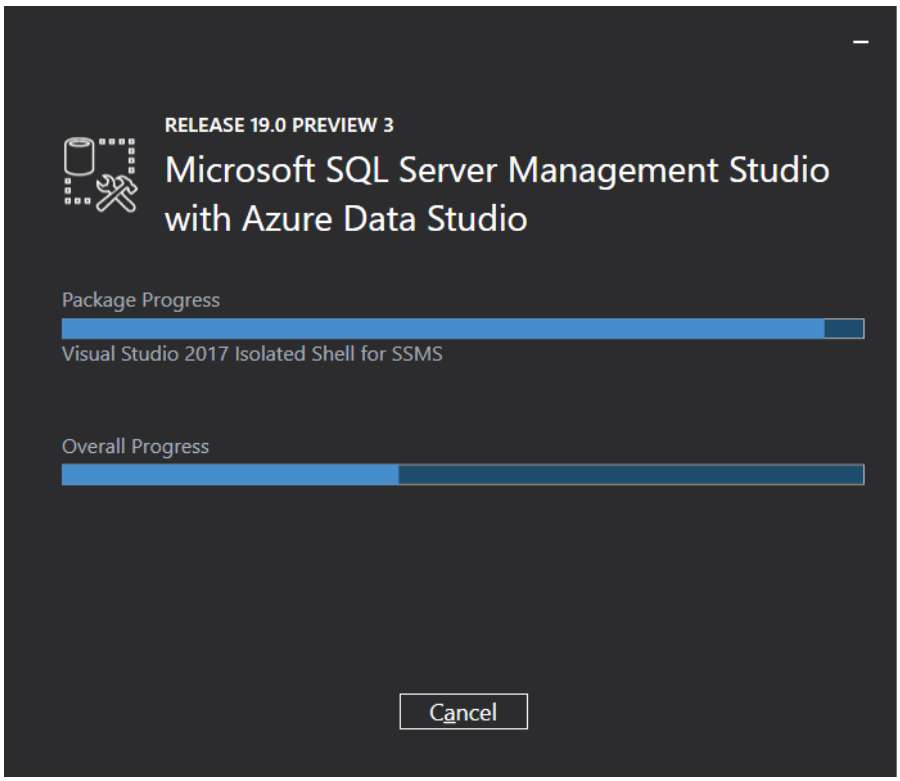
Далі потрібно запустити програму-установник та вибрати варіант «Basic»:



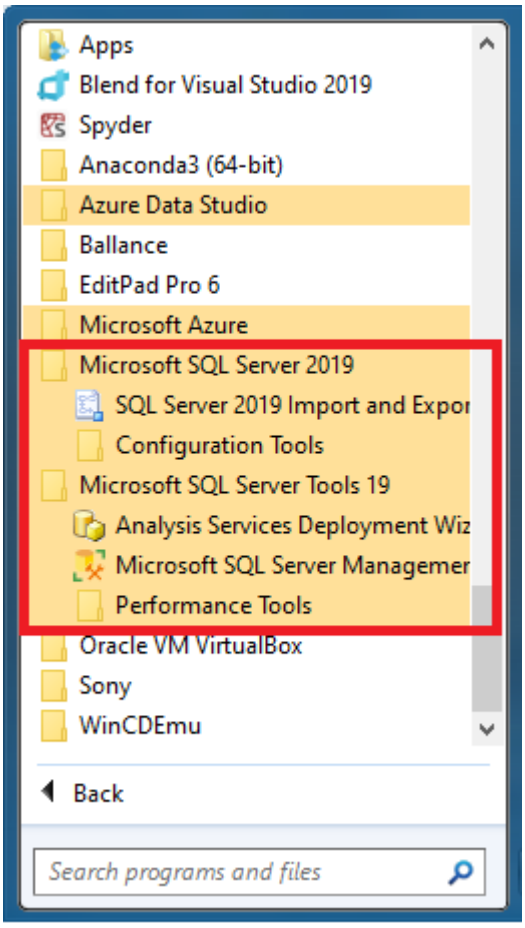
Після процесу встановлення на фінальному скрині буде кнопка для встановлення *SQL Server Management Studio*; натиснувши її Ви потрапите на веб-сторінку де можна буде завантажити програму-установник для *SQL Server Management Studio*:



Після завантаження запустіть програму-установник для *SSMS* і дотримуйтесь інструкцій; загалом, там потрібно тільки кнопку «Next» натискати:

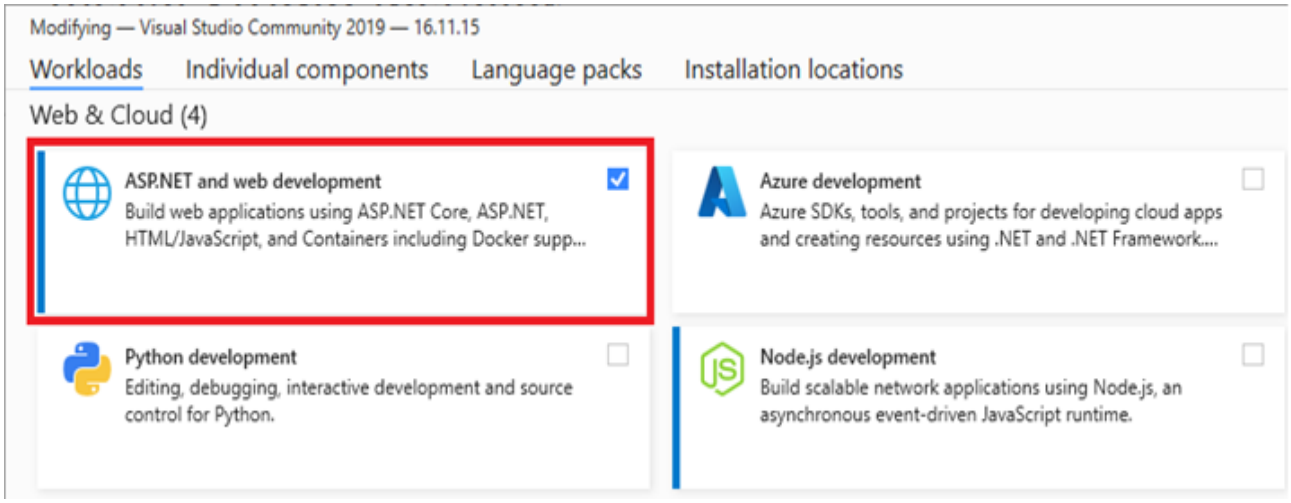


Після успішного встановлення в меню «Start» будуть доступні ці два продукти:



2.2 Налаштування MS Visual Studio

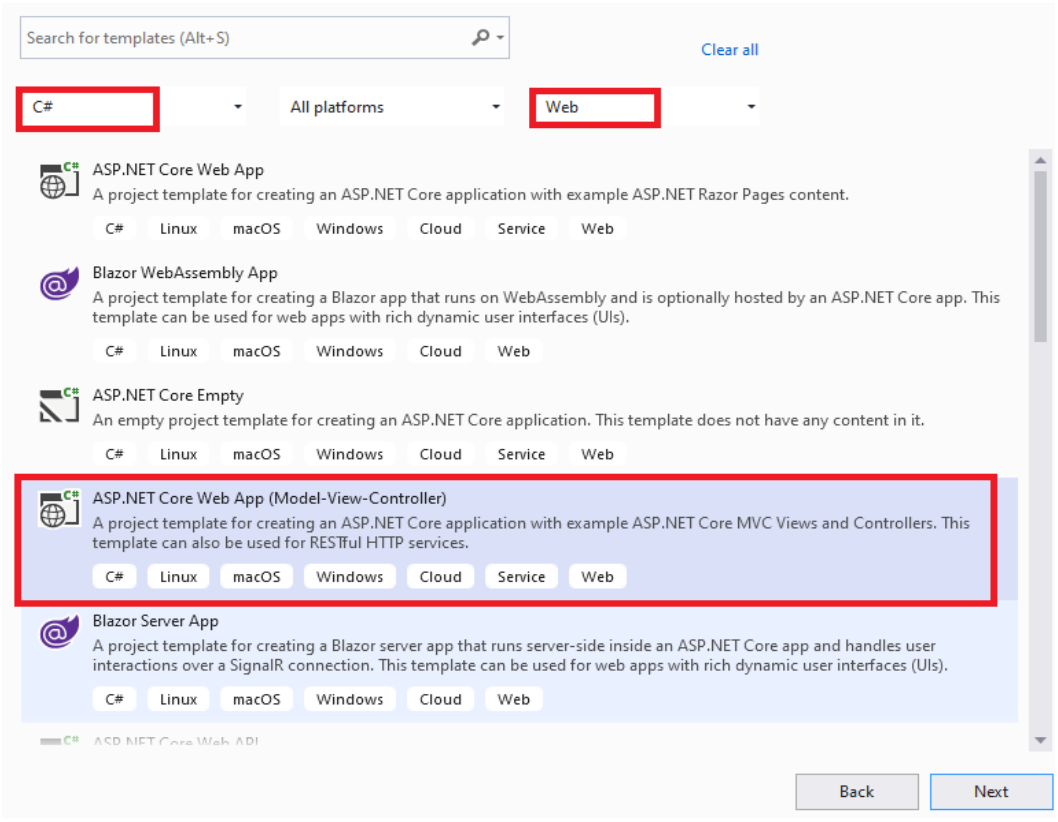
Для виконання даної лабораторної роботи необхідно, щоб у Вас були встановлені компоненти для веб-розробки:



2.3 Створення проекту

Відкрийте Visual Studio (надалі просто VS).

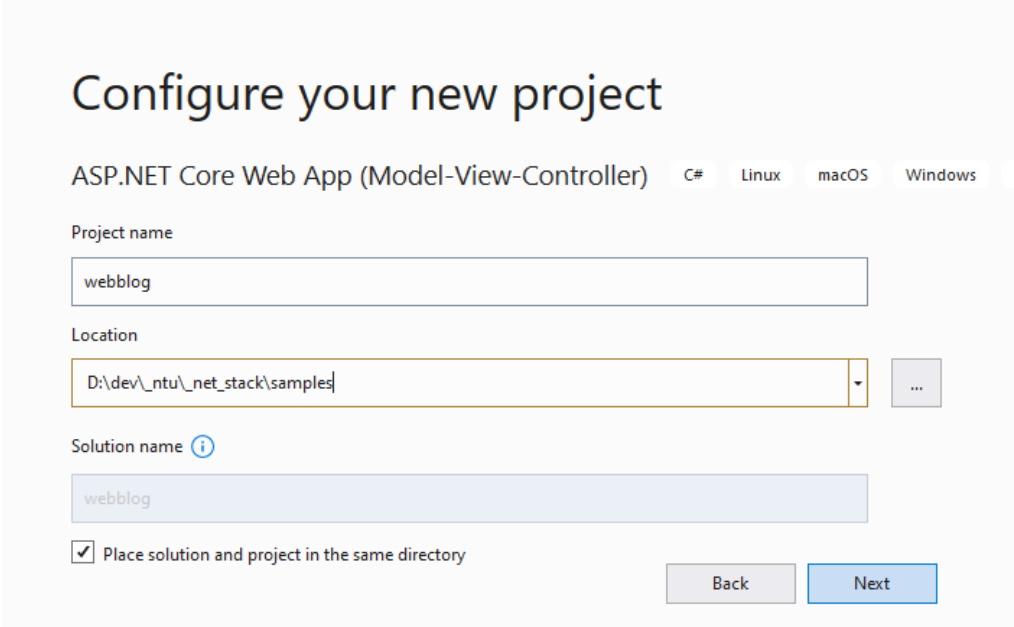
Виберіть команду File – Create – Project, або використовуйте клавіші Ctrl+Shift+N:



Тут необхідно вибрати тип проекту:

«Веб-додаток (модель-подання-контролер)»

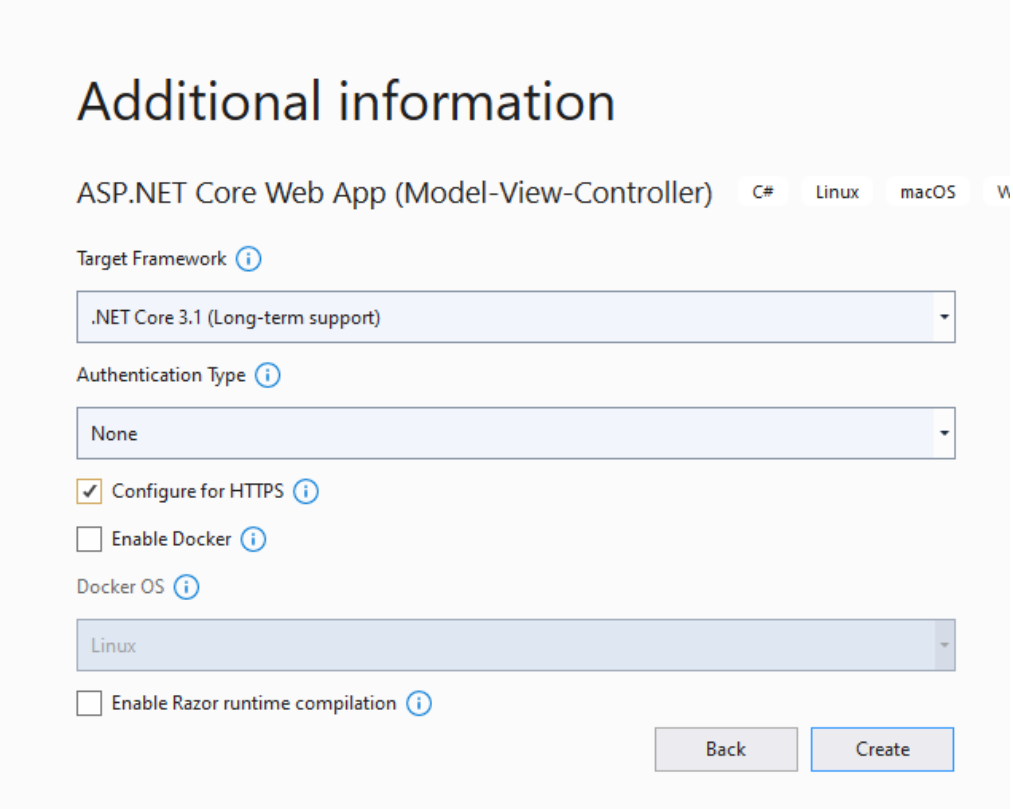
Натискаємо Next і в наступному діалозі задаємо ім'я проекту та папку, куди будуть збережені файли проекту. Щоб уникнути проблем, рекомендую використовувати латинські літери і не використовувати різні спец-символи та пробіли (окрім знака підкреслення _):



Тепер потрібно встановити параметри для проекту, який створюється

У комбобоксах форми вибираємо:

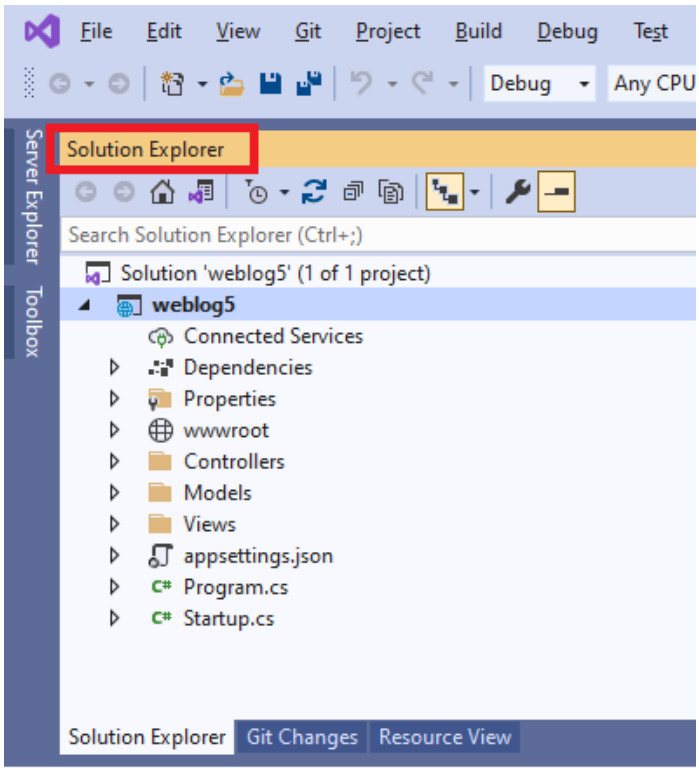
- ✓ *Версію ASP.NET (Core 3.1, 5.0, 6.0 або 7.0)*
- ✓ Авторизацію використовувати не будемо, тому ставимо None
- ✓ *HTTPS* та контейнеризацію Docker ми також використовувати не будемо, галочки прибираємо, якщо вони там стоять:



Таким чином, ми вибрали платформу, тип та параметри майбутнього додатку.

Буде створено веб-додаток з наступною структурою:

1. **Connected Services** – підключені сервіси з Azure.
2. **Dependencies** – всі додані до проекту пакети та бібліотеки, інакше кажучи залежності.
3. **Controllers** – Контролери додатку; тут ми опишемо всю логіку роботи з веб-сервером, а також маніпулювання даними.
4. **Models** – Дані програми, а також подання даних.
5. **Views** – «обличчя» нашої програми; саме тут ми визначаємо, який HTML код відображатиме програму.
6. **wwwroot** – Усі статичні файли (CSS, JS, images, etc).



7. **Program.cs** – Головний файл програми, з якого і починається його виконання. Код цього файлу налаштовує та запускає веб-хост, в рамках якого розгортається програма.
8. **Startup.cs** – файл, який визначає клас Startup та містить логіку обробки вхідних запитів.
9. **Properties** – Вузол, який містить деякі налаштування проекту. Зокрема, у файлі launchSettings.json описані налаштування запуску проекту, наприклад, адреси, за якими запускатиметься програма.
10. **appsettings.json** – файл конфігурації проекту у форматі json.

Це той каркас, від якого ми можемо відштовхуватися, додаючи до нього якісь свої файли та папки.

Вивчіть файл *Controllers -> HomeController.cs*. Як бачите, все в додатку зав'язано на контролерах та діях. Дія – це звичайний метод, який повертає тип IActionResult.

Натисніть **Ctrl+F5** для запуску сервера. Через кілька секунд відкриється веб-браузер і початкова сторінка додатку, яка була створена за замовчанням із написом «Welcome».

Зміна шаблону початкової сторінки

Змінимо подання, яке відповідає за головну сторінку. Для цього відкриємо файл ~/Views/Home/Index.cshtml і змінимо шаблон, додавши туди, наприклад, елемент h1, пару параграфів, і порожнє якірне посилання:

```
@{
    ViewData["Title"] ="Home Page";
 }
<div class="text-center">
  <h1 class="text-muted">Блог – weblog на ASP.NET Core</h1>
  <p>Сюди ми можемо помістити наш текст</p>
  <p>Learn about<a href="https://docs.microsoft.com/aspnet/core">
     building Web apps with ASP.NET Core</a>.</p>
  <a class="btn btn-primary btn-lg" asp-controller="Home" asp-action=”ShowAll>
     All records</a>
  </div>
```

Тут @{...} - покажчики на так званий вбудований (embedded) код мовою C #. Система візуалізації подань у ASP.NET Core називається Razor.

Пояснення: тут, у додатку, використовується Bootstrap – готовий набір CSS класів, що допомагає відразу розпочати візуалізацію будь-якого проекту. Докладніше про нього можна почитати на сайті http://getbootstrap.com

Тут і настає час дізнатися про так звані **tag-хелпери**.

Tag-хелпери є функціональністю, призначеною для генерації HTML-розмітки. Tag-хелпери використовуються в поданнях та виглядають як звичайні html-елементи або атрибути, проте при роботі додатку вони обробляються двигуном Razor на боці сервера і в кінцевому результаті перетворюються на стандартні html-елементи.

Tag-хелпери надають більш зручніший спосіб для генерації html-елементів, ніж звичайні html-хелпери; оскільки tag-хелпери багато в чому виглядають як звичайні html-елементи, Visual Studio має вбудовану підтримку IntelliSense для tag-хелперів.

Використовувати tag-хелпери досить просто. Наприклад, вище ми визначили у поданні наступний код:

```
<a class="btn btn-primary btn-lg" asp-controller="Home" asp-action=”ShowAll>
   All records</a>
```

Як бачимо, тут використовуються tag-хелпери asp-controller та asp-action.

Дані хелпери створюють посилання, для обробки якого буде використовуватись контролер Home та метод дії ShowAll() в зазначеному контролері.

Можливо, такий підхід Вам здасться більш інтуїтивно зрозумілим і звичним, ніж створення посилання за допомогою Html.ActionLink, наприклад:

@Html.ActionLink("Контакти", "Contacts", "Home")

У той же час взагалі необов'язково використовувати саме tag-хелпери. Ви можете використовувати звичайні html-теги, якщо вони Вам бачаться зручнішими.

2.4 Проектування моделі даних

У абревіатурі MVC літера "M" позначає model (модель), і вона є найважливішою частиною додатку. Модель – це подання реальних об'єктів, процесів і правил, які визначають сферу додатку, відому як предметна область.

Модель, яку часто називають моделлю предметної області, містить об'єкти C# (або об'єкти предметної області), які утворюють "всесвіт" додатку, і методи, які дозволяють маніпулювати ними. А подання та контролери відкривають предметну область клієнтам в узгодженій манері, і будь-який коректно розроблений додаток MVC починається з добре спроектованої моделі, яка потім служить центральним вузлом при додаванні контролерів та подань.

Наш додаток буде дуже простим і представлятиме звичайний блог. Основна функція будь-якого блогу – зберігати записи користувача. У зв'язку з цим ми можемо виділити перш за все дві моделі: модель посту (Post) і модель категорії посту (Category).

За умовами MVC класи, які визначають модель даних додатку, розміщуються у папці Models, яку Visual Studio створює під час початкового налаштування проекту. Отже, нам потрібно додати до неї два нові класи, які назвемо, відповідно, Post та Category. Клацніть правою кнопкою миші на папці Models у вікні Solution Explorer і виберіть у контекстному меню пункт Add, а потім пункт Class (Клас). В якості імен файлів вкажіть Post.cs та Category.cs.

Відредагуйте код класу, щоб він відповідав прикладу нижче:

Post.cs
```
using System;
namespace weblog.Models {
    public class Post {
        public Guid Id { get; set; }
        public string Title { get; set; }
        public DateTime PublishDate { get; set; }
        public bool IsDraft { get; set; }
        public string Content { get; set; }
        public int CategoryId { get; set; }
        public virtual Category Category { get; set; }
    }
}
```

Відсутність пункту Class у контекстному меню може означати, що ви залишили відладчик Visual Studio у стані виконання. Середовище Visual Studio обмежує види змін, які можна вносити у проект при функціонуючому додатку.

В данномувипадку ми вибрали для pk тип GUID. Entity Framework Core створить нам поле в БД з типом UNIQUEIDENTIFIER і автоматично запускатиме функцію NEWID() при додаванні нового запису.

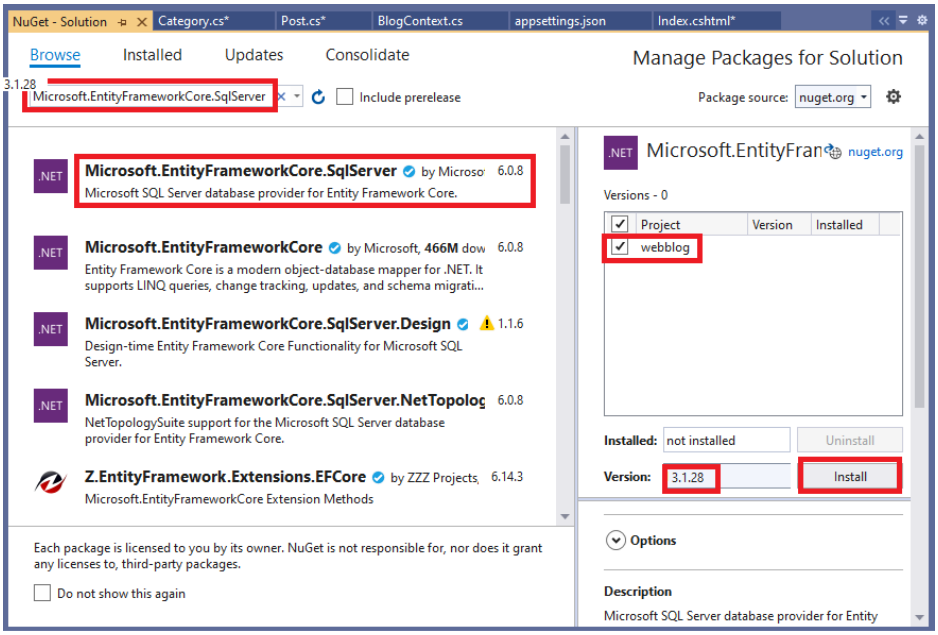
Category.cs
```
namespace weblog.Models {
    public class Category {
        public int Id { get; set; }
        public string Title { get; set; }
        public string Description { get; set; }
    }
}
```

Після визначення моделей предметної області потрібно вибрати сховище даних для цих моделей. Ми будемо використовувати *MS SQL Server*. Для роботи з *MS SQL Server* компанія Microsoft рекомендує використовувати технологію ORM Entity Framework, хоча її використання не є обов'язковим. Ми також можемо застосовувати інші технології ORM або доступні засоби ADO.NET. Перевага фреймворку Entity Framework полягає в тому, що він дозволяє абстрагуватися від структури конкретної бази даних та вести всі операції з даними через визначену на попередньому кроці модель.

В даному випадку для взаємодії з MS SQL Server через Entity Framework нам знадобиться пакет Microsoft.EntityFrameworkCore.SqlServer. Тому додамо цей пакет через пакетний менеджер NuGet, для цього заходимо в меню:

Tools -> NuGet Package Manager -> Manage NuGet Packages for Solution

і далі встановлюємо пакет, який нас цікавить:



Важливо обрати правильну версію пакета, який встановлюється, сумісну з версією .NET Core обрану для проекту, оскільки за замовчуванням буде,швидше за все, вказана latest stable, і можливо вона не буде сумісна і тоді пакет не встановиться.

Щоб взаємодіяти з базою даних, нам потрібен контекст даних. Причому Entity Framework Core використовує підхід Code First, при якому нам треба спочатку визначити моделі та контекст даних, а потім вже виходячи з цих моделей та класу контексту створюватиметься база даних та всі її таблиці.

Тож, додамо до папки Models новий клас, який назвемо BlogContext і який матиме наступний код:

```
using Microsoft.EntityFrameworkCore;

namespace weblog.Models {
    public class BlogContext : DbContext {
        public BlogContext(DbContextOptions<BlogContext> options)
            : base(options)
        {
            Database.EnsureCreated();
        }

        public DbSet<Post> Posts { get; set; }
        public DbSet<Category> Categories { get; set; }
    }
}
```

Щоб створити контекст, ми повинні успадкувати новий клас від класу DbContext.

Властивості на кшталт:

```
public DbSet<Post> Posts { get; set; }
public DbSet<Category> Categories { get; set; }
```

допомагають отримувати з БД набір даних певного типу, фактично кожна властивість DbSet буде співвідноситися з окремою таблицею бази даних.

За замовчуванням у нас немає бази даних. Тому у конструкторі BlogContext визначено виклик Database.EnsureCreated(), який за відсутності бази даних автоматично створює її. Якщо база даних вже існує, нічого не відбуватиметься.

Щоб підключатися до бази даних, нам потрібно встановити параметри підключення. Для цього змінимо файл *appsettings.json*, додавши визначення рядка підключення:

```
{
    "ConnectionStrings": {
    "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=blogstoredb;
                          Trusted_Connection=True;MultipleActiveResultSets=true"
    },
    "Logging": ...
```

За замовчуванням цей файл містить лише налаштування логування.

В нашому випадку ми будемо використовувати спрощений двигун бази даних *localdb*, який представляє легковажну версію *SQL Server Express*, призначену спеціально для взаємодії з додатками на стадії розробки. Базу даних назвемо *blogstoredb*.

І останнім кроком у налаштуванні проекту є зміна файлу *Startup.cs*. Цей файл відповідає за конфігурацію всіх сервісів програми. Для того, щоб додати підтримку SQL Server у наш проект, знаходимо метод ConfigureServices і змінюємо його наступним чином (виділені рядки потрібно додати):

```
public void ConfigureServices(IServiceCollection services)
{
    string connection = Configuration.GetConnectionString("DefaultConnection");
    services.AddDbContext<BlogContext>(options => options.UseSqlServer(connection));
    services.AddControllersWithViews();
}
```

Також необхідно додати на початку файлу наступні рядки:

```
using webblog.Models;  // простір імен моделей
```

```
using Microsoft.EntityFrameworkCore;  // простір імен EntityFramework
```

Тепер, коли у нас створені всі моделі та налагоджено підключення до бази даних, ми можемо скористатися механізмом Міграцій для того, щоб створити базу даних та всі необхідні в ній таблиці у відповідності до моделей проекту. Для цього необхідно виконати наступні дії:

1. Встановити пакет Microsoft.EntityFrameworkCore.Tools

Для цього виконуємо дії, які ми робили для встановлення пакету для роботи з MS SqlServer , тобто, йдемо в меню:

Tools -> NuGet Package Manager -> Manage NuGet Packages for Solution

і на формі, що відкрилася, вказуємо назву пакета **.Tools*, а також вказуємо версію *3.1.x* (або таку, яка відповідає версії .NET яку Ви використовуєте у проекті) і натискаємо *Install*.

2. Тепер знову йдемо в меню:

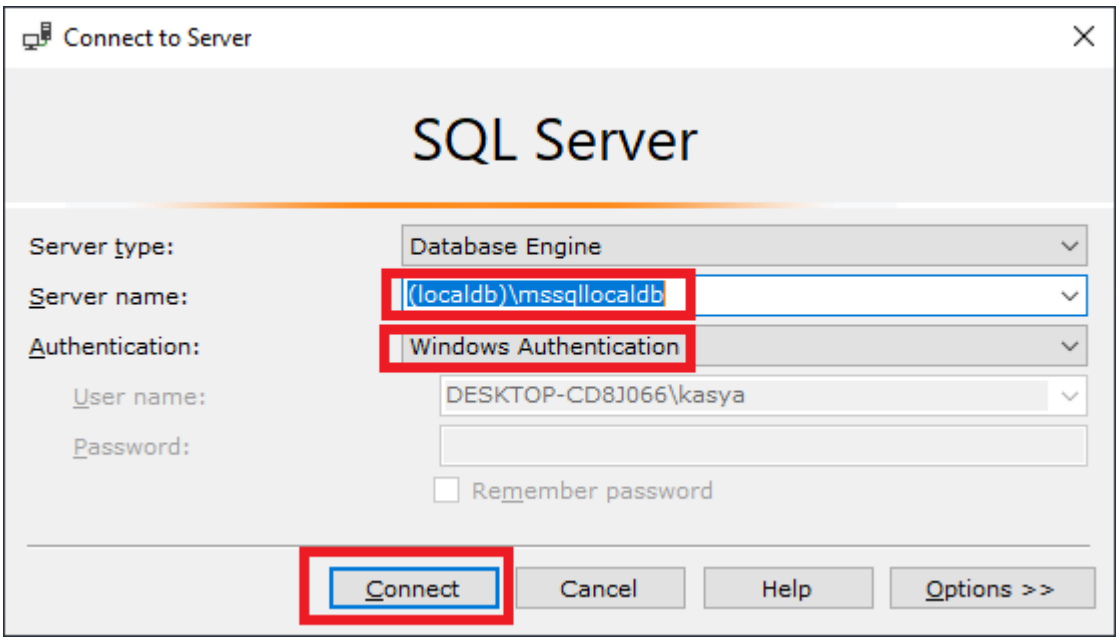
Tools -> NuGet Package Manager -> Package Manager Console

і у вікні виконуємо команду: *Add-Migration Initial*

Якщо все пройшло добре, то Ви побачите наступний результат:

```
PM> Add-Migration Initial
Build started...
Build succeeded.
To undo this action, use Remove-Migration.
```

Для адміністрування бази даних може бути використаний продукт, який називається *MS SQL Server Management Studio*. Для того щоб підключитися до локального *MS SQL Server* і перевірити, що база даних дійсно створена необхідно запустити *SSMS* (скориставшись ярликом у меню Start) і вказати на формі підключення наступні дані:

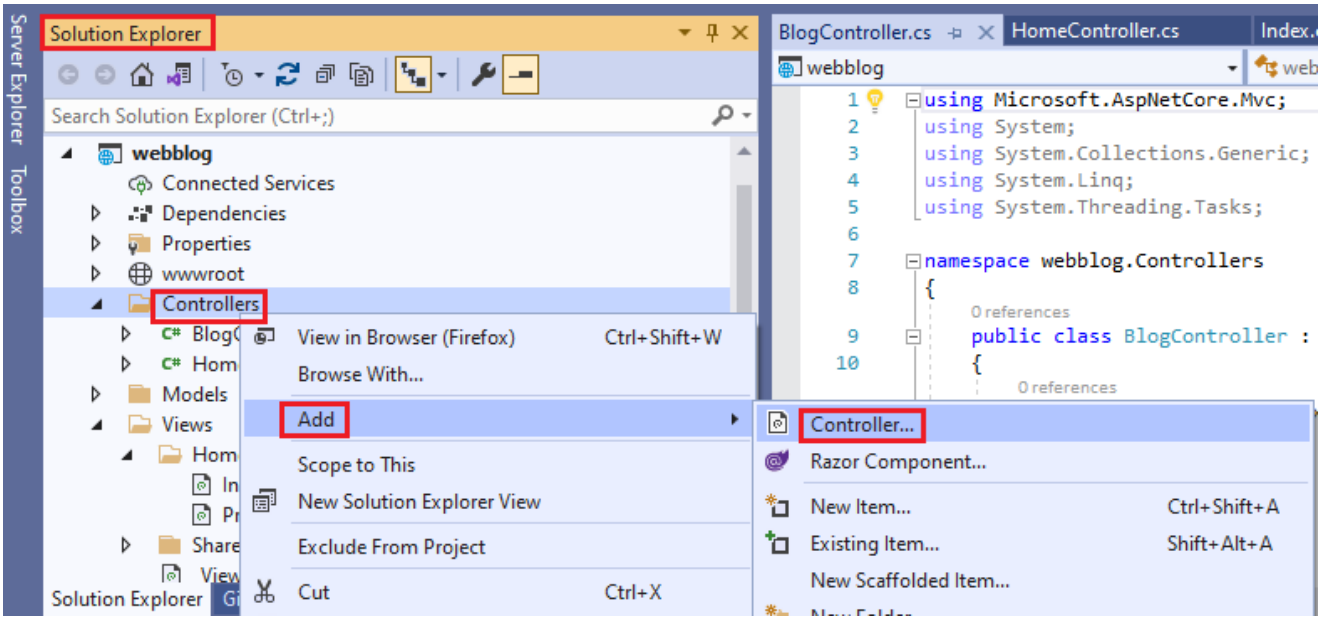


2.5 Створення контролера BlogController

Настав час створити контролер. У архітектурі MVC вхідні запити обробляються контролерами. У ASP.NET MVC контролери – це звичайні класи C# (зазвичай успадковані від System.Web.Mvc.Controller, вбудованого в інфраструктуру, базового класу контролера). Кожен публічний метод у контролері називається методом дії, і це означає, що його можна викликати з веб-середовища через певну URL-адресу для виконання дії. Відповідно до угоди MVC контролери розміщуються у папці Controllers, яка автоматично створюється Visual Studio при налаштуванні проекту.

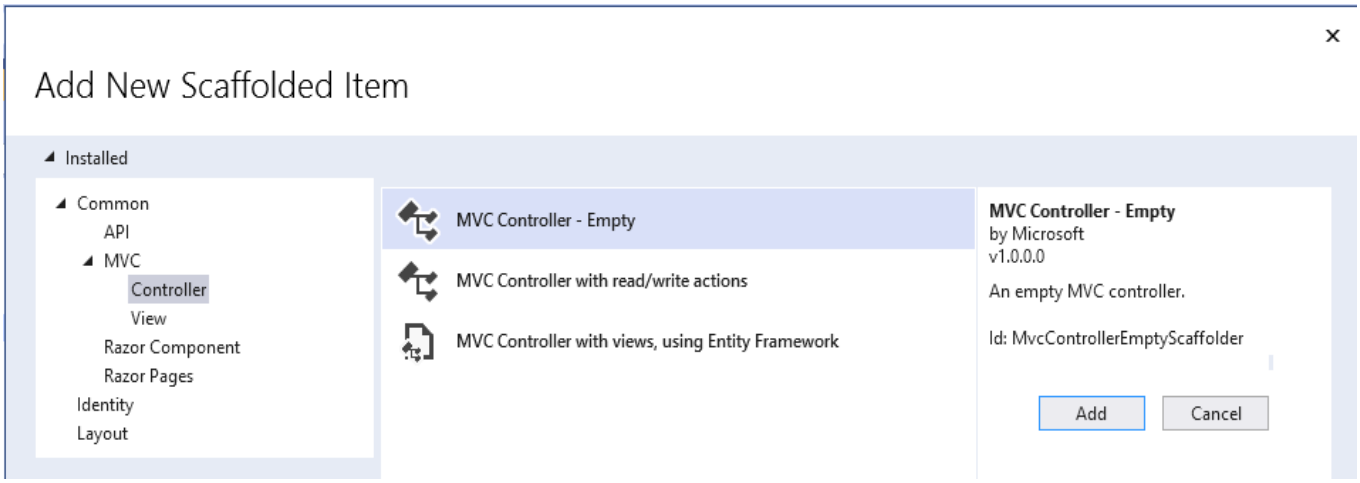
Дотримання цієї чи більшості інших угод MVC не обов'язкове, але рекомендується.

Щоб додати контролер у проект, клацніть правою кнопкою миші на папці Controllers у вікні Solution Explorer середовища Visual Studio та виберіть у контекстному меню пункт Add (Додати), а потім Controller (Контролер), як продемонстровано на малюнку нижче:



Коли відкриється діалогове вікно Add New Scaffolded Item (Додавання шаблону), виберіть варіант MVC Controller - Empty (Контролер MVC - Порожній), як показано на малюнку нижче, і клацніть на кнопці Add (Додати).

Далі, у вікні вибираємо «Контролер MVC – Порожній» і натискаємо ОК:



Відкриється діалогове вікно Add Controller (Додавання контролера). Вкажіть ім'я контролера BlogController і клацніть на кнопці Add (Додати). З цією назвою пов'язано кілька угод: імена, які вибираються для контролерів, повинні відображати їх призначення, стандартний контролер називається Home, і імена контролерів мають суфікс Controller.

Вікно виконання операції розпочне збирання інформації про проект, це може зайняти деякий час. Після закінчення буде сформовано контролера і одну дію – Index():

```
using Microsoft.AspNetCore.Mvc;

namespace webblog.Controllers {
    public class BlogController : Controller {
        public IActionResult Index() {
            return View();
        }
    }
}
```

Зручний спосіб розпочати знайомство з MVC передбачає внесення в клас контролера пари простих змін. Відредагуйте код у файлі BlogController.cs так, щоб він набув вигляду, показаного в прикладі нижче:

```
using Microsoft.AspNetCore.Mvc;

namespace webblog.Controllers {
    public class BlogController : Controller {
        public IActionResult Index() {
            return Content("Hello, world!");
        }
    }
}
```

Зміни не призводять до якихось особливо вражаючих результатів, але їх цілком достатньо для непоганої демонстрації. Метод дії під назвою Index() змінений так, що тепер він повертає рядок "Hello, World". Знову запустіть проект, вибравши пункт Start Debugging з меню Debug у Visual Studio. Браузер відобразить результат виконання методу дії Index(), як показано на малюнку нижче:

Зверніть увагу, що середовище Visual Studio направляє браузер на tcp-порт 45361. У URL-адресі, який буде запитувати ваш браузер, майже напевно буде присутній інший номер порту, оскільки Visual Studio виділяє випадковий tcp-порт при створенні проекту. Якщо ви заглянете в область повідомлень панелі завдань Windows, то знайдете там значок для IIS Express. Цей значок представляє усічену версію повного сервера додатків IIS, який входить у склад Visual Studio і використовується для доставки вмісту та служб ASP.NET під час розробки.

2.6 Поняття маршрутів

Крім моделей, подань та контролерів додатки MVC використовують систему маршрутизації ASP.NET, яка визначає, як саме URL-адреси відображаються на контролери та методи дій. Коли середовище Visual Studio створює проект MVC, додається ряд стандартних початкових маршрутів. Можна запитати будь-який із наступних URL, і вони будуть направлені на дію Index відповідного контролера:

/
/Home
/Home/Index

Отже, коли браузер запитує http://ваш-сайт/ або http://ваш_сайт/Home , він отримує вивід з методу Index() контролера HomeController. Ви можете спробувати це самостійно, змінивши URL у браузері. На даний момент він буде виглядати як http://localhost:45361/ , причому частина, яка визначає tcp-порт, може бути іншою. Якщо ви додасте до URL частину /Home або /Home/Index і натиснете клавішу Enter, то отримаєте від додатка MVC той же самий результат – рядок "Hello, world".

Це гарний приклад отримання вигоди від дотримання угод MVC. У даному випадку угода полягає в тому, що контролер названий HomeController, і він буде служити стартовою точкою нашого додатка MVC. При створенні стандартних маршрутів для нового проекту середовище Visual Studio виходить з припущення, що ми будемо слідувати цій угоді. І оскільки ми дійсно дотримуємося угоди, ми автоматично отримуємо підтримку всіх URL із наведеного вище переліку.

Якщо ж не слідувати угоді, маршрути довелося б змінити так, щоб вони вказували на контролер, створений замість стандартного. У наведеному простому прикладі стандартна конфігурація – це все, що було потрібно.

2.7 Візуалізація веб-сторінок

Виводом попереднього прикладу була не HTML-розмітка, а просто рядок "Hello World". Щоб згенерувати HTML-відповідь на запит браузера, потрібно створити подання. Перш за все, необхідно модифікувати метод дії Index(), як показано в прикладі нижче:

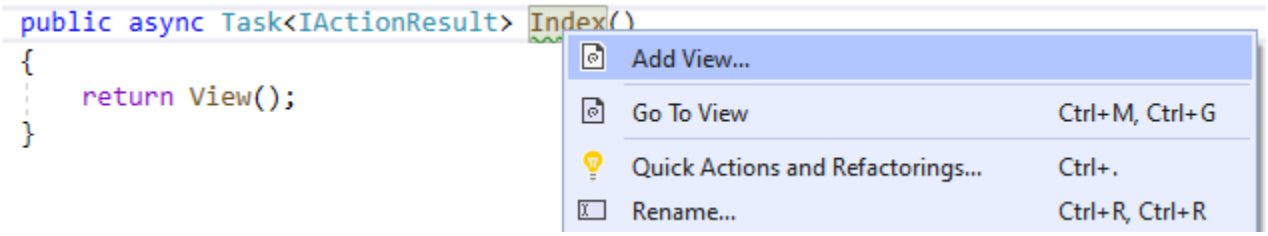
```
using Microsoft.AspNetCore.Mvc;

namespace webblog.Controllers {
    public class BlogController : Controller {
        public IActionResult Index() {
            return View();
        }
    }
}
```

Повертаючи об'єкт IActionResult із методу дії, ми тим самим вказуємо MVC, що потрібно візуалізувати представлення. Екземпляр IActionResult створюється викликом методу View() без параметрів. Це вказує інфраструктурі MVC на те, що для дії необхідно візуалізувати стандартне подання.

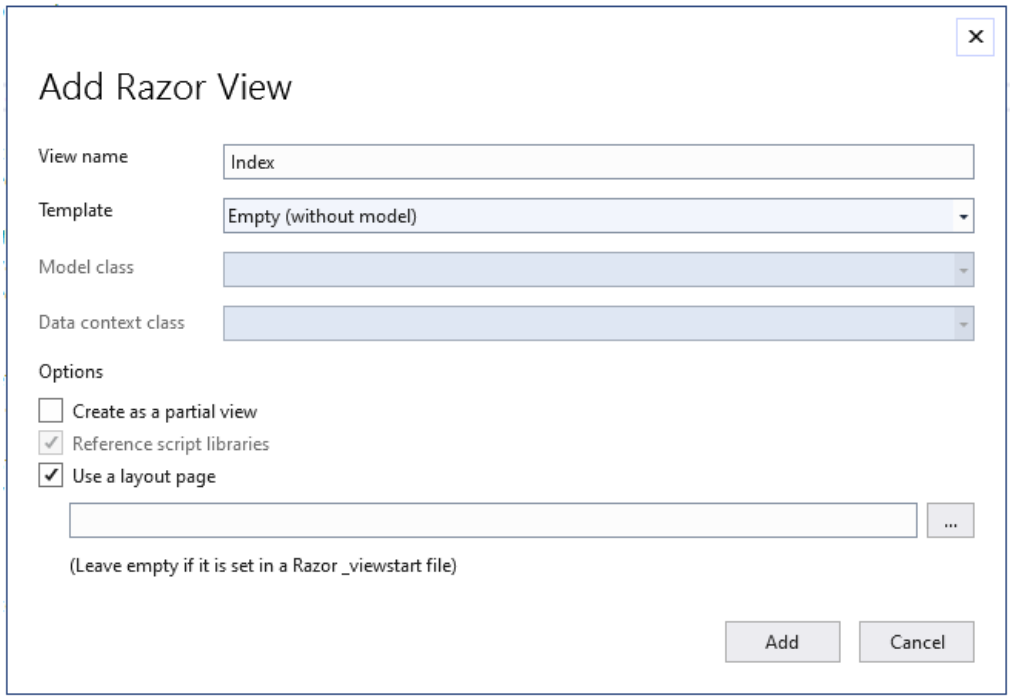
Якщо запустити додаток на цьому етапі, легко переконатися, що MVC Framework намагатиметься знайти для використання стандартне подання за замовчуванням, як показано в повідомленні про помилку на малюнку нижче:

Дане повідомлення про помилку надзвичайно корисне. Воно не тільки пояснює, що інфраструктура MVC не змогла знайти подання для методу дії, але й показує, де проводився пошук. Це ще одна хороша ілюстрація угоди MVC: подання асоціюються з методами дій за допомогою угоди про іменування. Метод дії має назву Index(), а контролер – Home, і, як бачимо на малюнку, інфраструктура MVC намагається знайти у папці Views файли з відповідним ім'ям:

Найпростіший спосіб створення подання передбачає доручення цієї роботи середовищу Visual Studio. Клацніть правою кнопкою всередині визначення методу дії Index() у вікні редактора коду, в якому відкритий файл BlogController.cs, і виберіть у контекстному меню пункт Add View (Додати подання):



Середовище Visual Studio відображає діалогове вікно Add View (Додавання подання), яке дозволяє налаштувати початковий вміст файлу подання, який буде створено. У полі View Name (Ім'я подання) введіть Index (ім'я методу дії, з яким буде асоційовано подання – ще одна угода), у переліку Template (Шаблон) оберіть варіант Empty (Порожній) та залиште прапорці Create as a partial view (створити як часткове подання) та Use a layout page (використовувати майстер-сторінку) не відміченими.



Клацніть на кнопці Add (Додати), щоб створити новий файл подання.

Середовище Visual Studio створить новий файл під назвою Index.cshtml у папці Views/Blog. Якщо це не те, що Ви очікували, видаліть створений файл та

спробуйте зробити все заново. Це ще одна угода MVC Framework: подання розміщуються у папці Views, організуючись всередині неї у папки, імена яких відповідають іменам асоційованих з ними контролерів.

Розширення файлу .cshtml позначає подання C#, яке буде оброблено механізмом візуалізації Razor. Ранні версії MVC покладалися на механізм візуалізації ASPX, файли подання для якого мали розширення .aspx.

В результаті встановленних опцій у діалоговому вікні Add View середовище Visual Studio створює дуже базове подання з вмістом, наведеним у наступному прикладі:

```
@{
    Layout = null;
}

<!DOCTYPE html>

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

Середовище Visual Studio відкриває файл Index.cshtml для редагування. Ви побачите, що цей файл містить переважно HTML-розмітку. Винятком є частина, яка має наступний вигляд:

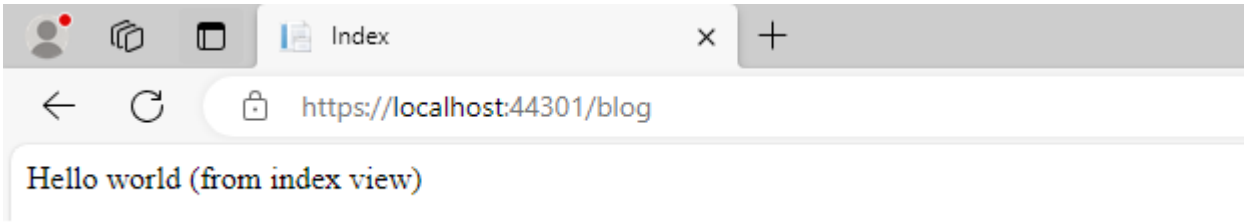
```
@{
    Layout = null;
}
```

Це вираз, який буде інтерпретований механізмом візуалізації Razor. Даний механізм обробляє вміст подання і генерує HTML-розмітку, яка надсилатиметься браузеру. Наведений вище вираз Razor повідомляє механізму візуалізації що майстер-сторінка застосовуватись не повинна. Ми поки не будемо звертати увагу на Razor, а повернемося до цього пізніше. Змініть файл Index.cshtml наступним чином:

```
@{
    Layout = null;
}

<!DOCTYPE html>
```

```
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <div>
        Hello world (from index view)
    </div>
</body>

</html>
```

Доданий код відображає ще одне просте повідомлення. Виберіть у меню Debug пункт Start Debugging, щоб запустити додаток і протестувати створене подання. Має відобразитися приблизно те, що показано на наступному малюнку:



Після першого редагування метод дії Index() повертав рядкове значення. Це означало, що інфраструктура MVC всього лише передавала браузеру рядок в незмінному вигляді. Тепер же, коли метод Index() повертає об'єкт IActionResult, інфраструктура MVC Framework візуалізує подання і повертає браузеру згенеровану HTML-розмітку.

Ми не повідомляли інфраструктурі MVC, яке подання повинно застосовуватися, тому була застосована угода про іменування для його автоматичного пошуку. Угода передбачає, що ім'я подання збігається з ім'ям методу дії і міститься в папці, названій у відповідності з ім'ям контролера: ~/Views/Home/Index.cshtml.

Окрім рядків і об'єктів IActionResult із методів дій можна повертати і інші результати. Наприклад, якщо ми повертаємо RedirectResult, то це приведе до перенаправлення браузера на іншу URL-адресу. Якщо ж ми повернемо HttpUnauthorizedResult, то змусимо користувача увійти до системи. Ці об'єкти у сукупності називаються результатами дій, і всі вони є похідними від класу ActionResult. Система результатів дій дозволяє інкапсулювати та повторно використовувати відповіді які найчастіше зустрічаються при виконанні методів дій.

Пояснення: Щоб посилання "All Records" вказувало на адресу */Blog/*, використовуючи розглянуті раніше tag-хелпери, її визначення має виглядати так:

```
<a class="btn btn-primary btn-lg" asp-controller="Blog" asp-action="">All records</a>
```

2.8 Додавання динамічного виводу

Весь сенс платформи для розробки веб-додатків зводиться до конструювання та відображення динамічного контенту. У рамках MVC робота контролера полягає у створенні певних даних та передачі їх поданню, яке відповідає за їх візуалізацію у вигляді HTML-розмітки.

Один зі способів передачі даних із контролера у подання передбачає використання об'єкта ViewBag, який є членом базового класу ControllerBase.

Об'єкт ViewBag – це динамічний об'єкт, у якому можна встановлювати довільні властивості, роблячи ці значення доступними в будь-якому візуалізованому далі поданні. У прикладі нижче демонструється передача простих динамічних даних у файлі BlogController.cs зазначеним способом:

```
public async Task<IActionResult> Index()
{
    int hour = DateTime.Now.Hour;
    ViewBag.Greeting = hour < 12 ? "Доброго ранку" : "Доброго дня";
    return View();
}
```

Дані для подання надаються під час присвоєння значення властивості ViewBag.Greeting. Властивість Greeting не існує до моменту присвоєння значення – це дозволяє передавати дані з контролера в подання у вільний та гнучкий спосіб, без необхідності у попередньому визначенні класів. Щоб отримати передані значення, необхідно ще раз звернутися до властивості ViewBag.Greeting, але вже в поданні, як наведено у наступному прикладі, що містить відповідну зміну, яка була внесена в файл Index.cshtml:

```
@{
    Layout = null;
}

<!DOCTYPE html>

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

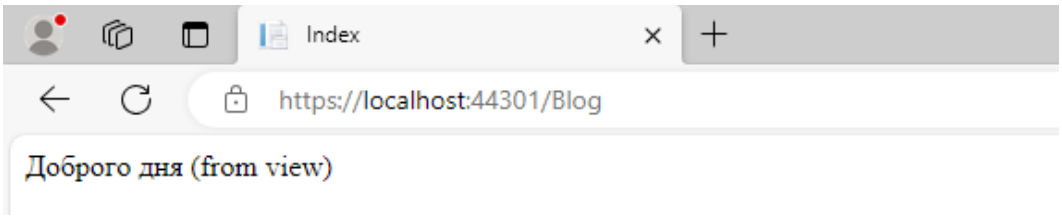
```
<body>
    <div>
        @ViewBag.Greeting (from view)
    </div>
</body>
</html>
```

Фрагмент, наведений у прикладі – це вираз Razor. Коли всередині методу Index() контролера викликається метод View(), інфраструктура MVC знаходить файл подання Index.cshtml і просить механізм візуалізації Razor про синтаксичний аналіз вмісту цього файлу. Механізм Razor шукає вирази, подібні до доданого у прикладі, і обробляє їх.

У розглянутому прикладі обробка виразу означає вставку у подання значення, яке було присвоєно властивості ViewBag.Greeting у методі дії.

Вибір для властивості імені Greeting не диктується якимось особливими міркуваннями. Його можна було б замінити будь-яким іншим ім'ям, і все працювало б так само за умови, що ім'я, яке застосовується в контролері, збігається з ім'ям, що використовується у поданні. Присвоюючи значення більш ніж одній властивості, можна передавати з контролера в подання безліч даних.

Після запуску проекту можна побачити результат внесених змін:



2.9 Ініціалізація бази даних

Щоб у нас у базі даних вже були початкові дані при запуску додатку, нам потрібен якийсь клас, який виконував би роль ініціалізатора бази даних. Для цього додамо до проекту (до папки Models) новий клас, який назвемо SampleData, код цього класу наведено нижче:

```
using System.Linq;
using System.Data;

namespace webblog.Models {
    public static class SampleData {
        public static void Initialize(BlogContext context)
```

```
        {
            if (! context.Categories.Any()) {
                context.Categories.AddRange(
                    new Category {
                        Title = "Розробка",
                        Description = "Розробка програмного забезпечення"
                    },
                    new Category {
                        Title = "Рецепти",
                        Description = "Кулінарні рецепти на всі випадки життя"
                    });

                context.SaveChanges();
            }

            if (! context.Posts.Any()) {
                context.Posts.Add(
                    new Post {
                        Title = "sample post",
                        IsDraft = false,
                        Content = "Сьогодні ми створили MVC веб-додаток!",
                        Category = context.Categories.Where(
                            b => b.Title == "Розробка").FirstOrDefault(),
                        PublishDate = DateTime.Now
                    });

                context.SaveChanges();
            }
        }
    }
}
```

Цей клас визначає один статичний метод Initialize(), у якому відбувається додавання початкових елементів – об'єктів Category та Post. Для додавання об'єктів у БД методу Initialize() передається контекст даних. І якщо дані у таблицях Category та Post у БД відсутні, то додаються нові об'єкти. Щоб ініціалізатор бази даних викликався при старті додатку, необхідно змінити клас Program, а саме метод Main(), який необхідно привести до наступного вигляду:

```
public static void Main(string[] args)
{
    var host = CreateHostBuilder(args).Build();
    using (var scope = host.Services.CreateScope())
    {
        var services = scope.ServiceProvider;
        try {
            var context = services.GetRequiredService<BlogContext>();
            SampleData.Initialize(context);
        }
        catch (Exception ex) {
            var logger = services.GetRequiredService<ILogger<Program>>();
            logger.LogError(ex, "An error occurred seeding the DB.");
        }
    }
```

```
    host.Run();
}
```

Також необхідно додати наступні залежності:

```
using webblog.Models;
using Microsoft.Extensions.DependencyInjection;
```

Можна перевірити, що база даних дійсно ініціалізована початковими даними, для цього запустіть *MS SQL Server Management Studio*:

2.10 Виведення записів блогу

Отже, початкові дані визначені, і тепер ми хочемо їх виводити на веб-сторінку. Для цього перейдемо до папки *Controllers*, де знаходиться контролер BlogController і внесемо до нього наступні зміни:

1. Додамо властивість _db типу BlogContext і в конструкторі надамо йому значення:

```
BlogContext _db;
public BlogController(BlogContext ctx) {
    _db = ctx;
}
```

2. Тепер нам необхідно написати логіку отримання записів із БД, для цього змінимо метод Index() наступним чином:

```
public async Task<IActionResult> Index() {
    var posts = await _db.Posts.Include(c => c.Category).ToListAsync();
    return View(posts);
}
```

Тут наведено приклад асинхронного методу. Він буде викликаний в окремому потоці, і всі обчислення, які він здійснюватиме, так само будуть виконуватись в окремому потоці.

3. Необхідно також до списку залежностей додати відповідні інструкції, а саме:

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using webblog.Models;
using Microsoft.EntityFrameworkCore;
```

Для відображення користувачю колекції записів нашого блога необхідно додати відповідне подання для методу дії Index(), але зараз ми зробимо це трохи іншим способом – ми створимо суворо типізоване подання (strongly-typed view). Суворо типізоване подання призначено для візуалізації певного типу предметної області, і якщо вказати тип, з яким потрібно працювати (у нашому випадку це буде List<webblog.Models.Post>), то MVC може створити ряд корисних скорочень, що полегшує кодування.

Змінимо файл *~/Views/Blog/Index.cshtml*, додавши туди наступний код (HTML розмітка залишається на розсуд студента):

```
@model List<webblog.Models.Post>

<div class="text-center">
    <h2 class="text-muted">Мої записи</h2></div>

@foreach(var p in Model)
{
    @p.Title <!-- заголовок запису -->
    @p.Category.Title<br> <!-- доступ до Foreign Key -->
    @p.Content
}
```

Як бачите, змін не багато – щоб змінити звичайне подання на суворо типізоване, треба лише додати директиву @model у редакторі коду і далі скористатися властивістю подання Model для доступу до переданих з контролера даних.

Використання суворо типізованого подання в MVC додатках має ряд значних переваг. По-перше, це забезпечує більш чітке і безпечне управління даними, оскільки компілятор може перевіряти типи на етапі компіляції, що зменшує ймовірність помилок, пов'язаних з типами даних. По-друге, суворо типізовані подання підвищують читабельність та підтримуваність коду, дозволяючи розробникам швидко розуміти, які дані використовуються в поданні, і як вони повинні бути структуровані. Також це сприяє кращій інтеграції з інструментами розробки, такими як IntelliSense в Visual Studio, які надають автоматичне завершення коду, підказки по API та іншу підтримку, що значно спрощує процес розробки. Врешті-решт, суворо типізовані подання дозволяють легше реалізувати принципи розробки, орієнтовані на доменні моделі, що призводить до більш організованого та модульного коду.

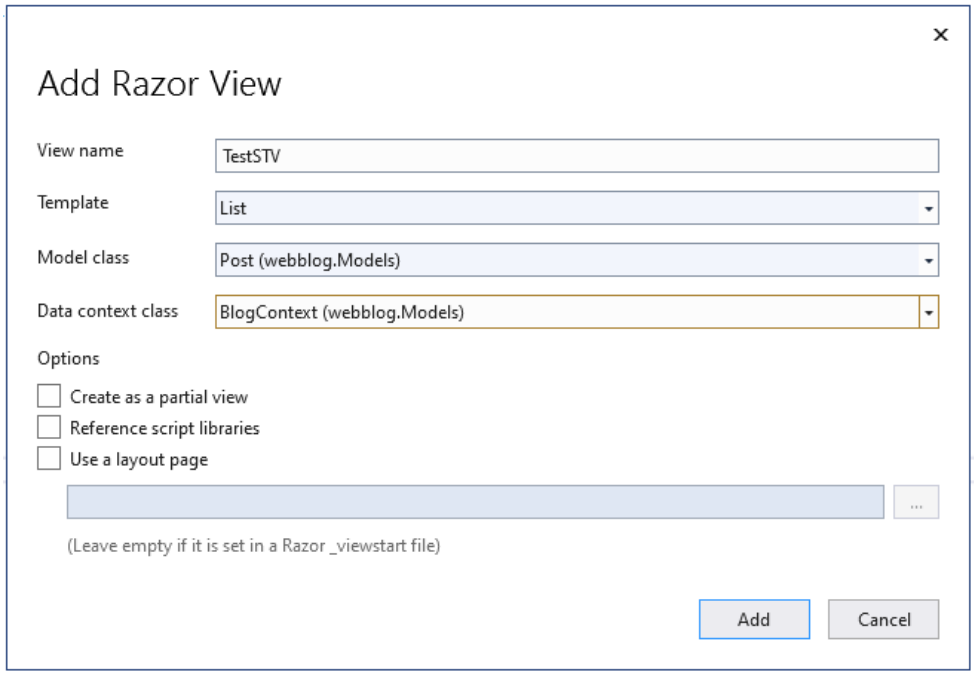
Вище ми розробили код подання для друку вмісту нашого блогу вручну, але Visual Studio може генерувати різноманітні варіанти суворо типізованого подання для певних типів автоматично.

Давайте спробуємо... Переконайтеся, що проект MVC скомпільовано. Якщо код класу webblog.Models.Post був написаний, але не скомпільований, інфраструктура MVC не зможе створити суворо типізоване представлення для цього типу. Щоб скомпілювати додаток, виберіть пункт Build Solution у меню Build середовища Visual Studio.

Створіть новий метод дії в контролері BlogController:

```
public async Task<IActionResult> TestSTV() {
    var posts = await _db.Posts.Include(c => c.Category).ToListAsync();
    return View(posts);
}
```

Клацніть правою кнопкою миші всередині методу дії TestSTV() у редакторі коду та виберіть у контекстному меню пункт Add View (додати подання), щоб відкрити діалогове вікно Add View. Переконайтеся, що у полі View Name (ім'я подання) вказано ім'я TestSTV, виберіть у списку Template (шаблон) варіант List (перелік), а у списку Model Class (клас моделі) – варіант webblog.Models.Post. Залиште всі прапорці у розділі View Options (параметри подання) не відміченими:



Після натискання на кнопку Add середовище Visual Studio створить новий файл під назвою TestSTV.cshtml у папці Views/Blog та відкриє його для

редагування. Опції, які вибираються та відмічаються під час створення подання, визначають початковий вміст файлу подання; на цьому про суворо типізовані подання все.

2.11 Керування постами у блозі – Робота з формами

Форми є однією з форм передачі даних на сервер. Як правило, для створення форм та їх елементів у MVC застосовуються або звичайні html-теги, або tag-хелпери а також директиви Razor.

Далі ми додамо в наш додаток можливість додавати записи в блог, редагувати вже існуючи та видаляти обрані.

Першім кроком удосконалимо подання Index.cshtml яке використовується для друку всіх постів у блозі. Додамо елементи керування записами, тобто посилання на виконання дій додавання, редагування та видалення постів:

```
@model IEnumerable<webblog.Models.Post>

<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Blog Posts</title>
    <!-- Add relevant stylesheets and scripts here -->
</head>
<body>
    <div class="container">
        <h2>Blog Posts</h2>
        @foreach (var post in Model)
        {
            <div class="blog-post">
                <h3>@post.Title</h3>
                <p>Category: @post.Category?.Title</p>
                <p>Published on: @post.PublishDate.ToString("d")</p>
                <div class="blog-content">@post.Content</div>
                <div class="blog-actions">
                    <a asp-action="Edit" asp-route-id="@post.Id">Edit</a> |
                    <a asp-action="Delete" asp-route-id="@post.Id">Delete</a>
                </div>
            </div>
        }
        <br><a asp-action="Create">Create New Post</a>
    </div>
    <!-- Add any additional scripts here -->
</body>
</html>
```

2.12 Моделі подання (view-models)

Додатки на ASP.NET MVC бувають різними: маленькими, великими, простими або зі складною логікою. І залежно від складності проекту ми можемо використовувати одну й ту ж саму модель для зберігання даних у базі даних, для передачі даних у подання та отримання даних з подання. Однак нерідко все ж моделі можуть не збігатися. Наприклад, нам не потрібно передавати у подання всі дані певної моделі або потрібно передати у подання об'єкти одразу двох моделей. І в цьому випадку ми можемо скористатися моделями подання.

У нашому випадку, при створенні або редагуванні постів блогу, можна буде обирати категорію (тему), до якої буде відноситися пост, з випадаючого списку. Тобто перелік категорій буде обмежений. Для реалізації цієї функціональності створимо спеці-альну модель для передачі даних у подання, або модель-подання (View Model).

Для цього спочатку додамо в проект нову папку ViewModels. В принципі, моделі представлень не обов'язково визначати саме в папці ViewModels; це може бути будь-яка папка, у тому числі і наявна за замовчуванням папка Models. У папці ViewModels створимо файл PostViewModel.cs в якому визначемо класс PostViewModel, який буде включати інформацію про пост та список категорій для створення випадаючого списку, що дозволить користувачам легко вибирати категорію при створенні або редагуванні постів блогу:

```
using System;
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc.Rendering;

namespace webblog.ViewModels {
    public class PostViewModel {
        public Guid Id { get; set; }
        public string Title { get; set; }
        public DateTime PublishDate { get; set; }
        public bool IsDraft { get; set; }
        public string Content { get; set; }
        public int CategoryId { get; set; }
        public IEnumerable<SelectListItem> Categories { get; set; }
    }
}
```

Створення моделей представлень є важливою практикою у розробці програмного забезпечення, особливо в контексті архітектури MVC (Model-View-Controller). Використання View Model дозволяє оптимізувати взаємодію між

даними, які ви бажаєте відобразити в поданні (View), і вашою моделлю домену (Models.Post).

Переваги, які надає такий підхід:

Розділення відповідальності: Models.Post є частиною моделі домену і являє собою структуру даних, яка відображає вашу бізнес-логіку та взаємодію з базою даних. В той час як PostViewModel призначена спеціально для потреб подання, і містить тільки дані та логіку, які безпосередньо використовуються в інтерфейсі користувача.

Безпека та мінімізація даних: За допомогою моделі подання ви можете контролювати, які саме дані надсилатимуться до подання. Це запобігає відображенню чутливої інформації та забезпечує, що подання отримує лише необхідні дані.

Гнучкість у форматуванні даних: використання окремого класу PostViewModel дозволяє легко форматувати дані для подання. Наприклад, ви можете додати властивості, які являють собою форматовані дати, перетворені значення та інші специфічні для UI елементи.

Спрощення зв'язування даних: У випадках, коли необхідно зв'язати декілька об'єктів або колекцій даних з поданням (наприклад, як от список категорій для випадаючого списку на формі), модель подання може об'єднати всі ці елементи в одному класі.

Валідація: модель подання може включати атрибути валідації, які специфічні для подання, без внесення змін у моделі домену.

Наступним кроком створемо у контролері BlogController відповідні методі дій:

```
public IActionResult Create() {  // Додавання нового посту
    var viewModel = new PostViewModel {
        Categories = _db.Categories.Select(c => new SelectListItem {
            Value = c.Id.ToString(),
            Text = c.Title })
    };
    return View(viewModel);
}
public async Task<IActionResult> Edit(Guid id) {  // Редагування посту
    var post = await _db.Posts.FindAsync(id);
    if (post == null) {
        return NotFound();
    }
    var viewModel = new PostViewModel {
```

```
        Id = post.Id,
        Title = post.Title,
        PublishDate = post.PublishDate,
        IsDraft = post.IsDraft,
        Content = post.Content,
        CategoryId = post.CategoryId,
        Categories = _db.Categories.Select(c => new SelectListItem {
            Value = c.Id.ToString(),
            Text = c.Title,
            Selected = post.CategoryId == c.Id })
    };
    return View(viewModel);
}
public async Task<IActionResult> Delete(Guid id) {  // Видалення посту
    var post = await _db.Posts.FindAsync(id);
    if (post != null) {
        _db.Posts.Remove(post);
        await _db.SaveChangesAsync();
    }
    return RedirectToAction(nameof(Index));
}
```

2.13 Створення форм

Тепер, маючи методи дій для створення та редагування постів блога, можна приступити до розробки саме форм для введення відповідних даних. Тобто на цьому кроці створюємо подання для методів дій Create та Edit з наступним кодом:

```
@model webblog.ViewModels.PostViewModel
Create.cshtml

@{
    ViewData["Title"] = "Create Post";
}

<h2>Create Post</h2>

@using (Html.BeginForm("Create", "Blog", FormMethod.Post))
{
    <p><label asp-for="Title">Title:</label>
       <input asp-for="Title" /></p>
    <p><label asp-for="PublishDate">Publish Date:</label>
       <input asp-for="PublishDate" /></p>
    <p><label asp-for="IsDraft">Is Draft:</label>
       <input asp-for="IsDraft" /></p>
    <p><label asp-for="Content">Content:</label>
       <textarea asp-for="Content"></textarea></p>
    <p><label asp-for="CategoryId">Category:</label>
       <select asp-for="CategoryId" asp-items="Model.Categories"></select></p>
    <input type="submit" value="Create" />
}
```

```
@model webblog.ViewModels.PostViewModel
```

```
Edit.cshtml

@{
    ViewData["Title"] = "Edit Post";
}

<h2>Edit Post</h2>

@using (Html.BeginForm("Edit", "Blog", FormMethod.Post))
{
    @Html.HiddenFor(model => model.Id)

    <p><label asp-for="Title">Title:</label>
       <input asp-for="Title" /></p>
    <p><label asp-for="PublishDate">Publish Date:</label>
       <input asp-for="PublishDate" /></p>
    <p><label asp-for="IsDraft">Is Draft:</label>
       <input asp-for="IsDraft" /></p>
    <p><label asp-for="Content">Content:</label>
       <textarea asp-for="Content"></textarea></p>
    <p><label asp-for="CategoryId">Category:</label>
        <select asp-for="CategoryId" asp-items="Model.Categories"></select></p>
    <input type="submit" value="Save Changes" />
}
```

У коді цих форм використовуються як тег-хелпери, так і HTML-хелпери. Вважається, що тег-хелпери пропонують більш читабельний та інтуїтивно зрозумілий синтаксис порівняно з традиційними HTML-хелперами (такими як @Html.HiddenFor), які були популярні в старших версіях ASP.NET MVC.

Тег-хелпери (наприклад asp-for) в ASP.NET Core розроблені для того, щоб більш ефективно інтегруватися з HTML-розміткою, забезпечуючи при цьому потужний механізм біндінгу даних і взаємодії з моделями Razor. Вони дозволяють розробникам писати код, який є більш схожим на стандартний HTML, і при цьому забезпечується динамічне конструювання веб-елементів.

З іншого боку, HTML-хелпери використовуються у вигляді методів C#, що дозволяє писати більш звичний для серверної логіки код. Однак цей підхід може бути менш інтуїтивним для розробників, які звикли до роботи зі стандартним HTML.

Вибір між тег-хелперами та HTML-хелперами часто залежить від переваг розробника та специфіки проекту. На практиці багато команд використовують обидва підходи в залежності від потреб та контексту завдань.

Так, допоміжний метод HTML @Html.HiddenFor використовується для створення прихованого поля вводу в HTML-формі. Це дозволяє передавати дані,

які не відображаються користувачеві, але є важливими для обробки форми на сервері.

Фактично цей HTML-хелпер генерує HTML-розмітку для елемента вводу типу hidden. Цей метод встановлює параметр type в значення hidden та атрибути id та name, відповідно до назви вибраної властивості класу моделі:

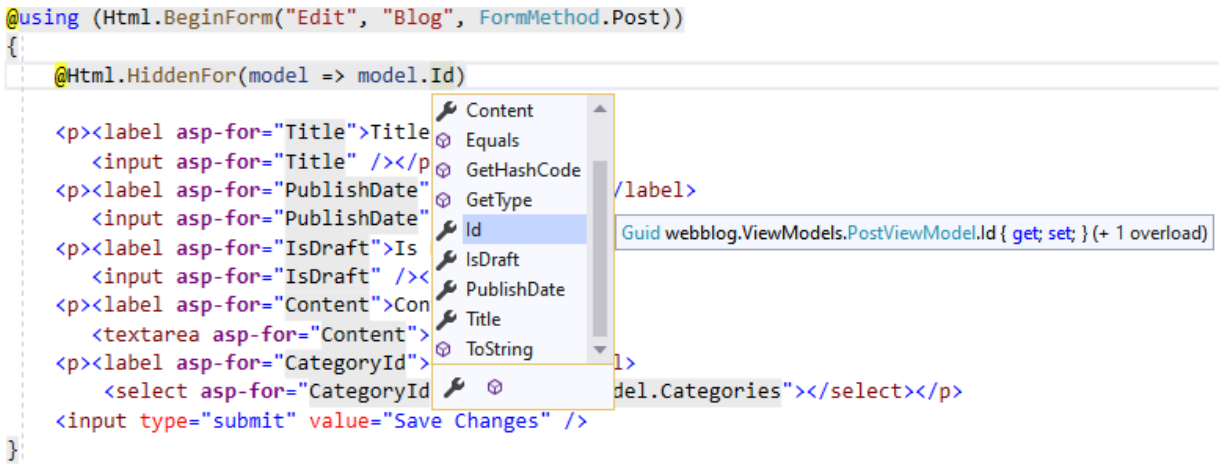
```
<input id="Id" name="Id" type="hidden" value="…" />
```

Цей зручний інструмент працює ефективно завдяки тому, що подання є суворо типізованим. Це означає, що інфраструктурі MVC вказано, який тип моделі потрібно візуалізувати за допомогою цього подання. Таким чином HTML-хелпери отримують необхідну інформацію для визначення того, з якого типу даних мають бути отримані певні значення через razor-вираз @model.

Лямбда-вираз model => model.Id вказує на властивість Id моделі Post, яка передається до цього подання. Ця властивість визначає дані, які будуть використані для створення прихованого поля. Альтернативою використанню лямбда-виразів є пряме звернення до імені властивості типу моделі як до рядка:

```
@Html.HiddenFor("Id")
```

Підхід з використанням лямбда-виразів запобігає помилковому введенню назви властивості типу моделі, оскільки середовище Visual Studio активує інструмент IntelliSense, що дозволяє автоматично вибрати певну властивість:



Ще одним зручним допоміжним методом є Html.BeginForm(), який генерує HTML-елемент form, налаштований на виконання оберненої відправки (post) методу дії. Якщо ніякі аргументи цьому допоміжному методу не передавати, він припускатиме, що потрібно виконати обернену відправку за тією ж самим URL

адресою, з якої запитувався HTML-документ. Вишуканий трюк полягає в тому, щоб розмістити цей метод всередині C#-оператора using, як наведено нижче:

```
@using (Html.BeginForm())
{
    // тут розміщується вміст форми...
}
```

Зазвичай при такому застосуванні оператор using гарантує звільнення об'єкта, коли він залишає область дії. Таке використання поширене, наприклад, для підключень до баз даних, щоб гарантувати закриття з'єднання негайно після завершення запиту (таке застосування ключового слова using відрізняється від того його різновиду, яке включає класи з простору імен в область дії якогось класу).

В нашому випадку замість звільнення об'єкта допоміжний метод Html.BeginForm закриває HTML-елемент form, коли той залишає область дії. Це означає, що допоміжний метод Html.BeginForm створює обидві частини елементу форми:

```
<form action="/Blog/Edit" method="post">
<!-- ... вміст форми -->
</form>
```

2.14 Обробка форм

Інфраструктурі MVC ми поки що не визначили, що слід робити, коли форма відправляється серверу. У нинішньому стані додатку клік на кнопці Save Changes лише очищує будь-які значення, введені у форму. Причина в тому, що форма здійснює зворотну відправку відповідному методу дії (Create або Edit) контролера BlogController, який лише повідомляє MVC про необхідність повторної візуалізації подання.

Факт втрати введених даних при повторній візуалізації подання може вас здивувати. Якщо це так, то вам, швидше за все, доводилося розробляти додатки за допомогою інфраструктури ASP.NET Web Forms, яка в такій ситуації автоматично зберігає дані. Далі буде наведено, як досягти подібного ефекту в MVC.

Щоб отримати та обробити дані, відправлені через форму, ми використаємо ще одну цікаву можливість – додамо альтернативні реалізації методів дій (Create та Edit), що дозволять реалізувати наступні аспекти:

Метод для обробки GET-запитів. Коли хтось клікає на посилання, браузер, як правило, відправляє запит типу GET. Ця версія методу дії буде відповідальна за відображення первісно порожньої форми.

Метод для обробки POST-запитів. Зазвичай форми, створені за допомогою Html.BeginForm, відправляються браузером через POST-запит. Ця версія методу буде займатися обробкою відправлених даних і прийняттям рішення про їх подальше використання.

Розділення обробки GET та POST-запитів на різні методи C# сприяє чіткості та структурованості коду контролера, оскільки кожен з методів відповідає за свої завдання. Обидва методи викликаються за однією і тією ж URL-адресою, але MVC вибирає відповідний метод залежно від типу отриманого запиту – GET чи POST.

Цей механізм реалізується за допомогою атрибуту [HttpPost], який вказує MVC, що новий метод буде працювати тільки з POST-запитами. До вже існуючих методів дій, основна задача яких полягає в відображенні порожніх форм, можна за бажанням додати атрибут [HttpGet], хоча це й не є обов'язковим.

Нижче наведено приклад коду методів дій, які потрібно додати до класу BlogController:

```
[HttpPost]
public async Task<IActionResult> Create(PostViewModel model) {
    if (ModelState.IsValid) {  // перевірки достовірності введених даних
        var post = new Post {
            Title = model.Title,
            PublishDate = model.PublishDate,
            IsDraft = model.IsDraft,
            Content = model.Content,
            CategoryId = model.CategoryId
        };

        _db.Add(post);
        await _db.SaveChangesAsync();

        return RedirectToAction(nameof(Index));  // перенаправлення до списку постів
    }

    // якщо є помилки в моделі
    model.Categories = _db.Categories.Select(c => new SelectListItem {
        Value = c.Id.ToString(),
        Text = c.Title
    });

    return View(model);
}

[HttpPost]
public async Task<IActionResult> Edit(PostViewModel model) {
    if (ModelState.IsValid) {  // перевірки достовірності введених даних

        var post = await _db.Posts.FindAsync(model.Id);
        if (post == null)
            return NotFound();

        post.Title = model.Title;
        post.PublishDate = model.PublishDate;
        post.IsDraft = model.IsDraft;
        post.Content = model.Content;
        post.CategoryId = model.CategoryId;

        _db.Update(post);
        await _db.SaveChangesAsync();

        return RedirectToAction(nameof(Index));  // перенаправлення до списку постів
    }

    // якщо є помилки в моделі
    model.Categories = _db.Categories.Select(c => new SelectListItem {
        Value = c.Id.ToString(),
        Text = c.Title
    });

    return View(model);
}
```

2.15 Використання прив'язки моделі

Як вже було зазначено перші перевантажені версії методів дій (Create та Edit) відповідальні за відображення форм.

Друга перевантажена версія, яка відповідає за обробку введених даних, більш цікава через наявність параметра. Але з урахуванням того, що цей метод дії буде викликатися у відповідь на HTTP-запит типу POST, а тип PostViewModel є класом C#, як вони поєднуються між собою?

Секрет полягає у прив'язці моделі – надзвичайно корисній функціональній можливості MVC, за якою вхідні дані аналізуються, а пари "ключ/значення" в HTTP-запиті використовуються для заповнення властивостей у типах моделі предметної області або моделі представлення. Цей процес є протилежним застосуванню допоміжних методів HTML; тобто при створенні даних форми для відправки клієнту ми генеруємо HTML-елементи input, в яких значення для атрибутів id та name створюються з імен властивостей відповідного класу моделі.

Навпаки, завдяки прив'язці моделі, імена елементів input використовуються для встановлення значень властивостей в екземплярі класу відповідної моделі, який потім передається методу дії, який обробляє POST -запит.

Прив'язка моделі – потужний та гнучкий інструмент, який звільняє програміста від копіткої та важкої роботи по безпосередньої взаємодії з HTTP-запитами і дозволяє працювати з об'єктами C#, а не мати справу зі значеннями Request.Form[] та Request.QueryString[]. Об'єкт PostViewModel, який передається як параметр цьому методу дії, автоматично заповнюється даними з полів форми.

2.16 Додавання перевірки достовірності

Тепер настав час додати до додатку перевірку достовірності введених даних. Без перевірки достовірності користувачі зможуть вводити безглузді дані або навіть відправляти порожню форму.

У додатку MVC перевірка достовірності зазвичай застосовується у моделі предметної області, а не в користувацькому інтерфейсі. Це означає можливість визначення в одному місці потрібних критеріїв перевірки достовірності, які вступають у дію скрізь, де використовується клас моделі.

Інфраструктура ASP.NET MVC підтримує декларативні правила перевірки достовірності, визначені за допомогою атрибутів з простору імен System.ComponentModel.DataAnnotations, і це означає, що обмеження (constraints) перевірки достовірності виражаються за допомогою стандартних атрибутів C#. У прикладі нижче показано, як застосувати ці атрибути до класу моделі PostViewModel:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using Microsoft.AspNetCore.Mvc.Rendering;

namespace webblog.ViewModels
{
    public class PostViewModel
    {
        public Guid Id { get; set; }

        [Required(ErrorMessage = "Заголовок обов'язковий")]
        [StringLength(100,
            ErrorMessage = "Довжина заголовка не може перевищувати 100 символів")]
        public string Title { get; set; }

        [Required(ErrorMessage = "Дата публікації обов'язкова")]
        public DateTime PublishDate { get; set; }

        public bool IsDraft { get; set; }

        [Required(ErrorMessage = "Вміст поста не може бути порожнім")]
        public string Content { get; set; }

        [Required(ErrorMessage = "Необхідно вибрати категорію")]
        public int CategoryId { get; set; }

        public IEnumerable<SelectListItem> Categories { get; set; }
    }
}
```

Інфраструктура MVC виявляє атрибути перевірки достовірності та використовує їх для перевірки даних під час процесу прив'язки моделі. Зверніть увагу, що ми імпортували простір імен, який містить засоби перевірки достовірності, так що до них можна звертатися без вказівки повних імен.

Перевірка на наявність проблем із достовірністю даних виконується за допомогою властивості ModelState.IsValid у класі контролера BlogController в методах дій Create та Edit, які відповідають за отримання та обробку введених даних (тобто викликаються при отриманні POST-запитів).

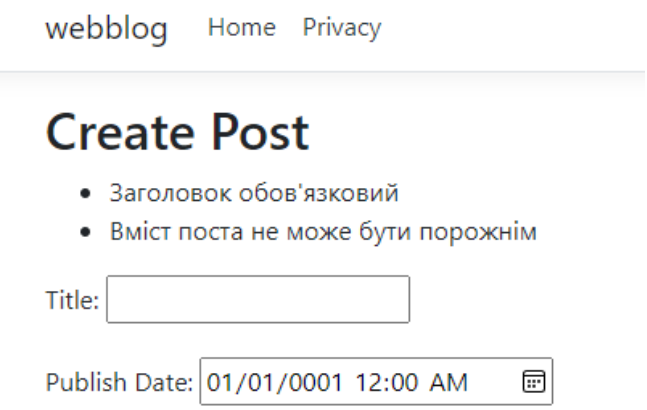
Отже, у випадку відсутності помилок перевірки достовірності, ми зберігаємо дані в базі даних та вказуємо MVC про необхідність здійснення перенаправлення до списку постів блога. У разі виявлення помилок перевірки достовірності, ми повторно візуалізуємо відповідне подання, викликаючи метод View().

Проте, просте повторне відображення форми у ситуації, коли виявлено помилку, може бути не дуже інформативним для користувача. Важливо надати зрозумілу інформацію про те, в чому конкретно полягає проблема, і чому відправлені дані не можуть бути прийняті. Для цього у поданнях Create та Edit можна скористатися допоміжни методом Html.ValidationSummary() як наведено у наступному прикладі:

```
@model webblog.ViewModels.PostViewModel

@{
    ViewData["Title"] = "Create Post";
}

<h2>Create Post</h2>

@using (Html.BeginForm("Create", "Blog", FormMethod.Post))
{
    @Html.ValidationSummary()
    <p><label asp-for="Title">Title:</label>
        <input asp-for="Title" /></p>
    <p><label asp-for="PublishDate">Publish Date:</label>
       <input asp-for="PublishDate" /></p>
    <p><label asp-for="IsDraft">Is Draft:</label>
       <input asp-for="IsDraft" /></p>
    <p><label asp-for="Content">Content:</label>
       <textarea asp-for="Content"></textarea></p>
    <p><label asp-for="CategoryId">Category:</label>
       <select asp-for="CategoryId" asp-items="Model.Categories"></select></p>
    <input type="submit" value="Create" />
}
```

При відсутності помилок метод Html.ValidationSummary() створює прихований елемент списку у вигляді заповнювача всередині форми. Інфраструктура MVC робить заповнювач видимим та додає повідомлення про помилки, визначені атрибутами перевірки достовірності.

Якщо перевірка достовірності даних не була пройдена, буде відображено узагальнений перелік помилок, які були виявлені як наведено на зображенні:



Зверніть увагу, що дані, введені у форму, були збережені і показані знову при візуалізації подання зі сводкою перевірки достовірності. Це ще одна перевага, яку забезпечує засіб прив'язки до моделі, і вона спрощує роботу з даними форми.

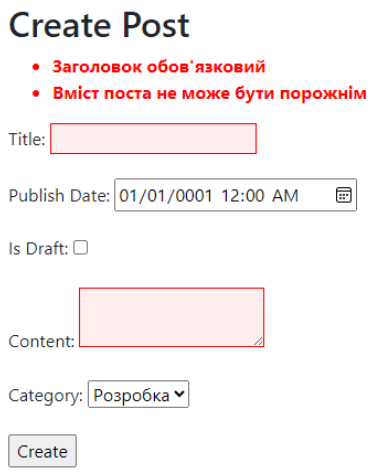
2.17 Підсвічування полів з помилковими значеннями

Для підсвічування полів із помилками червоним кольором потрібно визначити класи CSS, які автоматично застосовуються до елементів форми з помилками у ASP.NET Core (тобто вам не потрібно додавати використання цих css-стилів до елементів форми вручну в коді опису представлень). Ці класи зазвичай включають input-validation-error для полів вводу, select-validation-error для випадаючих списків і textarea-validation-error для текстових областей. Угода, яка прийнята в проектах MVC, передбачає розміщення статичного вмісту, такого як таблиці стилів CSS, у папці з назвою wwwroot. Отже, додайте у файл site.css, розташований у папці wwwroot/css/ наступний код:

```
.field-validation-error {
    color: #f00;
}

.field-validation-valid {
    display: none;
}

.input-validation-error {
    border: 1px solid #f00;
    background-color: #fee;
```

Навіть після зміни вмісту файла site.css ваш браузер може продовжити використовувати стару версію файлу, це пов'язано з кешуванням. Тому виконайте примусове оновлення сторінки з ігноруванням кешу натиснувши **Ctrl + F5**.

```
}

.validation-summary-errors {
    font-weight: bold;
    color: #f00;
}

.validation-summary-valid {
    display: none;
}
```



Після застосування наведених вище css-стилів при відправці даних, які викликають помилку перевірки достовірності, візуально очевидне пояснення причин проблем відображатимється користувачю як показано на малюнку.

2.18 Стилізація вмісту

Базова функціональність додатку розроблена, однак його зовнішній вигляд загалом досить непривабливий. Незважаючи на те, що цей посібник з MVC зосереджений на розробці серверної частини, корисно розглянути кілька відкритих бібліотек, які прийняті Microsoft і включені у низку шаблонів проектів Visual Studio.

Bootstrap являє собою зручну бібліотеку CSS, розроблену в Twitter і згодом отримавшу широке застосування.

Звісно, вам не обов'язково використовувати шаблони проектів Visual Studio, щоб користуватися бібліотеками на зразок Bootstrap. Можна завантажити файли безпосередньо з веб-сайтів із потрібними бібліотеками або скористатися інструментом NuGet, інтегрованим у Visual Studio який надає доступ до каталогу заздалегідь упакованого програмного забезпечення, яке можна завантажити та встановити автоматично.

Однією з найкращих характеристик NuGet є те, що цей інструмент враховує залежності між пакетами, тож під час установки, наприклад, бібліотеки Bootstrap, інструмент NuGet також завантажить і встановить бібліотеку jQuery, від якої залежить Bootstrap.

2.19 Стилізація переліку постів блогу

Базові інструменти Bootstrap працюють шляхом застосування класів до елементів, які відповідають селекторам CSS, визначених у файлах, доданих до папки wwwroot/css/. Детальну інформацію про класи, визначені в бібліотеці Bootstrap, можна отримати на веб-сайті Bootstrap, а в наступному прикладі демонструється використання ряду базових стилів у поданні index.cshtml:

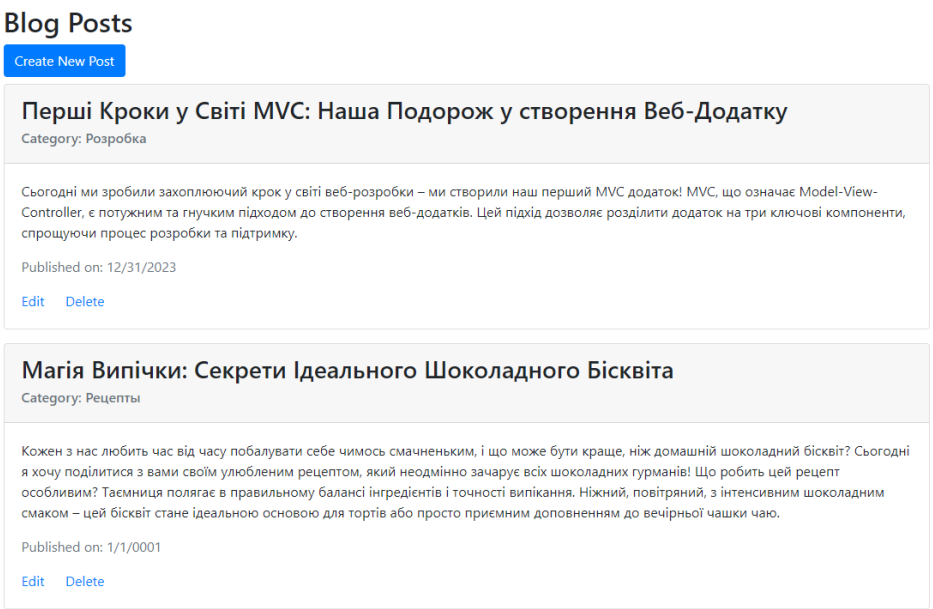
```
@model IEnumerable<webblog.Models.Post>

<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <meta name="viewport"
          content="width=device-width, initial-scale=1">
    <title>Blog Posts</title>
    <link rel="stylesheet"
          href="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css">
</head>
```

```
<body>
    <div class="container mt-3">
        <h2>Blog Posts</h2>
        <div class="mb-2">
            <a asp-action="Create" class="btn btn-primary">Create New Post</a>
        </div>

        @foreach (var post in Model) {
        <div class="card mb-3">
            <div class="card-header">
                <h3 class="card-title">@post.Title</h3>
                <h6 class="card-subtitle mb-2 text-muted">
                    Category: @post.Category?.Title</h6></div>
            <div class="card-body">
                <p class="card-text">@post.Content</p>
                <p class="text-muted">Published on: @post.PublishDate.ToString("d")</p>
                <a asp-action="Edit"
                   asp-route-id="@post.Id" class="card-link">Edit</a>
                <a asp-action="Delete"
                   asp-route-id="@post.Id" class="card-link">Delete</a></div>
        </div>
        }
    </div>
    <script src="https://code.jquery.com/jquery-3.5.1.slim.min.js">
       </script>
    <script src="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/js/bootstrap.min.js">
       </script>
</body>
</html>
```

В розмітку були додані елементи link для підключення файлу bootstrap.css та script для файлів jquery.js та bootstrap.js яки завантажуються з папки wwwroot. Це файли Bootstrap, які потрібні для базової стилізації CSS, яку забезпечує ця бібліотека.

Ви помітите, що для кожного файлу Bootstrap існує його аналог з суфіксом min - наприклад, є файли bootstrap.css та bootstrap.min.css. Це поширена практика мінімізації файлів JavaScript та CSS при розгортанні додатків у виробничому середовищі, яка полягає у видаленні усіх пробільних символів, а також у випадку файлів JavaScript скорочення назв функцій та змінних. Метою мінімізації є скорочення обсягу даних, необхідних для доставки вмісту у браузер.

Ще одна важлива деталь – це розташування тегів link та script, які підключають відповідно css-стилізацію до сторінки та сценарії JavaScript. При підключенні css-стилів на початку сторінки, браузер може швидше відобразити сторінку одразу встановивши правильний стиль, уникаючи моментів, коли вміст виглядає неоформленим або моргає (якщо певний елемент вже відображено зі стилізацією за замовчанням, а потім для нього буде завантажено інший стиль, браузеру доведеться перемалювати цей елемент). З іншого боку, JS-скрипти підключаються у кінці сторінки, це дозволяє браузеру швидше відобразити вміст сторінки, а вже потім виконувати код скриптів. Таким чином, сторінка стає доступною для користувача швидше, а динамічні елементи, оброблені JavaScript-скриптами, активуються після того, як основний вміст вже відображено. Отже тепер перелік постів блогу виглядатиме так:



Замість того щоб замінювати початковий вміст подання Index.cshtml без стилізації, з учбовою метою, стилізоване подання можна розташувати в іншому файлі, наприклад з назвою IndexBootstrapStyled.cshtml. Щоб створити зазначене подання, клацніть правою кнопкою миші на будь-якому з методів дій класу BlogController та оберіть у контекстному меню пункт Add View (додати подання). За допомогою наступного діалогового вікна) створіть строго типізоване подання з назвою IndexBootstrapStyled, яке використовує клас моделі Models.Post та базується на шаблоні "Empty". Середовище Visual Studio створить подання у вигляді файла Views/Blog/IndexBootstrapStyled.cshtml.

Для того щоб вказати MVC, що у відповідь на запит повинно візуалізуватися специфічне подання, відмінне від стандартного, при виклику методу View() першим параметром необхідно вказати назву відповідного подання як наведено нижче:

```
public class BlogController : Controller {
    ...
    public async Task<IActionResult> Index() {
        ...
        return View("IndexBootstrapStyled", posts);
    }
```

2.20 Стилізація форм

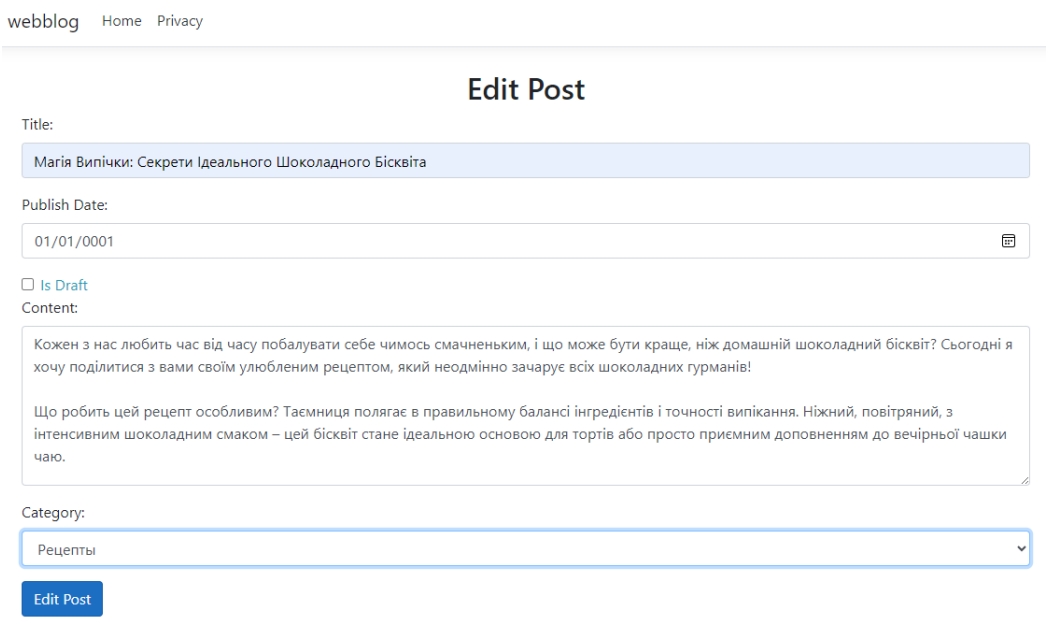
У бібліотеці Bootstrap визначені класи, які можуть використовуватися для стилізації форм. Я не планую заглиблюватися в особливі деталі, але у прикладі нижче показано, як були застосовані ці класи:

```
@model webblog.ViewModels.PostViewModel

@{
    var formTitle = Model.Id == Guid.Empty ? "Create Post" : "Edit Post";
    var formAction = Model.Id == Guid.Empty ? "Create" : "Edit";
    ViewData["Title"] = formTitle;
}

<div class="container mt-4">
    <h2 class="text-center">@formTitle</h2>
    @using (Html.BeginForm(formAction, "Blog", FormMethod.Post, new {
        @class = "needs-validation", novalidate = "novalidate" })) {
        @Html.ValidationSummary(true, "", new { @class = "text-danger" })

        @if (Model.Id != Guid.Empty) {
```

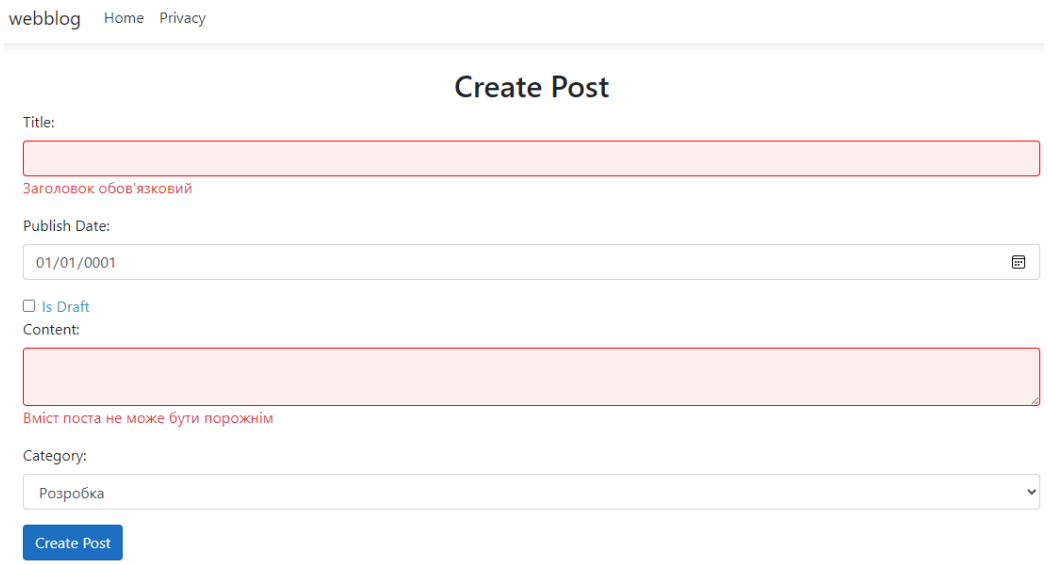
```
            @Html.HiddenFor(m => m.Id)
        }

        <div class="form-group">
            <label asp-for="Title">Title:</label>
            <input asp-for="Title" class="form-control" />
            <span asp-validation-for="Title" class="text-danger"></span></div>

        <div class="form-group">
            <label asp-for="PublishDate">Publish Date:</label>
            <input asp-for="PublishDate" type="date" class="form-control" />
            <span asp-validation-for="PublishDate" class="text-danger"></span></div>

        <div class="form-check">
            <input asp-for="IsDraft" class="form-check-input" />
            <label asp-for="IsDraft" class="form-check-label text-info">Is Draft</label>
            <span asp-validation-for="IsDraft" class="text-danger"></span></div>

        <div class="form-group">
            <label asp-for="Content">Content:</label>
            <textarea asp-for="Content" class="form-control"></textarea>
            <span asp-validation-for="Content" class="text-danger"></span></div>

        <div class="form-group">
            <label asp-for="CategoryId">Category:</label>
            <select asp-for="CategoryId"
                    asp-items="Model.Categories" class="form-control"></select>
            <span asp-validation-for="CategoryId" class="text-danger"></span></div>

        <button type="submit" class="btn btn-primary">@formTitle</button>
    }
</div>
```

Також замість використання окремого подання для дій Create та Edit це оновлене подання може бути використано для обох цих методів. Щоб це зробити та знов таки не видаляти код форм без Bootstrap-стилізації (файли Create.cshtml та Edit.cshtml), створіть ще одне пусте суворо типізоване подання, назвіть його CreateModifyPost та використайте наведений вище код. Також змініть відповідним чином виклики методу View() в методах дій Create та Edit контролера BlogController.

Результати стилізації можна побачити на малюнках нижче:



Форма створення нового поста у стані помилки:



От і все, ми реалізували невеликий за своїми масштабами додаток на платформі ASP.NET Core.

ЗАВДАННЯ ДЛЯ САМОСТІЙНОГО ВИКОНАННЯ

Створити веб-додаток за допомогою ASP.NET Core MVC.

1. Створіть новий проект ASP.NETCore MVC, використовуючи MS Visual Studio та виконайте базові налаштування проекту для з'єднання з базою даних.

2. Використовуючи принцип Code First, створіть класи моделей, згідно вашого варіанту і далі, використовуючи механізм міграцій Entity Framework, згенеруйте базу даних.

3. Заповніть модель тестовими даними за допомогою програми на мові C# (можливо використання окремого проекту C# або коду, вбудованого в проект ASP.NET Core MVC, як було розглянуто в теоретичній частині).

4. Створіть контролер та вид, який формує вибірку даних у вигляді HTML-таблиці з використанням технології LINQ to Entities, а також можливості для додавання, редагування та видалення записів з БД.

Номер варіанта предметної галузі необхідно обирати за останньою цифрою залікової книжки.

Варіант 1 - Кулінарна книга

- Рецепти (найменування, опис, джерело, тип, час приготування)
- Компоненти рецепту (продукт, кількість)
- Довідник продуктів (найменування, ціна, одиниця виміру, калорійність)

Варіант 2 - Розклад занять

- Розклад (тиждень, день, пара, група, викладач, вид занять, аудиторія)
- Викладач (кафедра, посада, науковий ступінь)
- Пара (номер, час початку, час закінчення)
- Аудиторія (корпус, номер)
- Група (номер, кількість студентів)

Варіант 3 - Структура комп'ютерної мережі організації

- Пристрої (назва, модель, версія прошивки, налаштування, додаткові відомості)
- Інтерфейси (ip-адреси, mac-адреси, якому пристрою належить цей інтерфейс)
- Зв'язки між інтерфейсами

Варіант 4 - Майстерня

Майстерня здійснює ремонт побутової техніки різних типів (придумайте не менше трьох). Клієнт привозить несправну техніку та оформляє факт передачі техніки із зазначенням її необхідних властивостей. Після прийому техніки майстер (певний співробітник майстерні) протягом трьох днів оцінює складність ремонту та формує перелік замінних частин та перелік дій (наприклад: розбирання та складання, робота із заміни екрану, тощо), які визначають вартість ремонту. Клієнта повідомляють про вартість ремонту та отримують згоду на подальші дії після первинного аналізу (діагностики). Ремонт може бути виконаний "за гарантією", у цьому випадку вартість ремонту клієнтом не оплачується.

Варіант 5 - Інформаційна система ЖЕКу

Інформаційна система ЖЕКу обслуговує групу будинків на кількох вулицях. Заявка надходить від квартири. Заявку приймає диспетчер, він задає номер та дату надходження заявки, визначає тип заявки та термін її виконання. Заявку виконує бригада спеціалістів. Кожен фахівець може працювати лише в одній бригаді, кожна бригада має бригадира.

Варіант 6 - Парки

Інформаційна система "Парки міста" зберігає інформацію про парки. Кожен парк має власне ім'я. У парку висаджено певні насадження. База даних повинна зберігати інформацію про кількість насаджень кожного типу. У парку можуть знаходитися фонтани та павільйони.

- Парк (найменування, площа, місце знаходження (адреса))
- Насадження (тип культури, найменування, середня тривалість життя)
- Фонтан (шифр, дата побудови, витрата води (макс і нормальний), площа).
- Павільйон (назва, тип (кафе, продуктовий, розважальний, прокат речей), площа).

Варіант 7 - Прокат автомобілів

Компанія пропонує низку моделей автомобілів на прокат. Клієнт може забронювати будь-яку модель чи конкретну модель автомобіля. Бронь відноситься до моделі, але не до конкретного автомобіля цієї моделі. Коли автомобіль цієї моделі стане доступним, його буде приведено для клієнта. Коли клієнт бере автомобіль, то ця інформація пов'язується з конкретним автомобілем, а не моделлю, оскільки у компанії може бути кілька автомобілів однієї моделі. Необхідно вести облік дати початку використання автомобіля та тривалості.

Варіант 8 - Мережа готелів

Мережа готелів під назвою B&B складається з багатьох філій готелів по всій країні. У кожному готелі є різні номери. Кожен номер має власну ціну в залежності від типу. B&B зберігає інформацію про своїх гостей, щоб можна було надсилати їм листи та електронні листи про спеціальні пропозиції. Повинна зберігатись інформація про бронювання. Ця інформація включає дату, з якою гість забронював номер та дату від'їзду. Потрібно знати кількість днів.

Варіант 9 - Галереї

Галереї зберігають інформацію про художників, їх імена (які унікальні), місця народження, вік та стиль мистецтва. У кожного твору мистецтва є автор, рік виготовлення, назва, вид твору (наприклад, живопис, літографія, скульптура, фотографія) та ціна. Твори мистецтва також поділяються на групи різних видів, наприклад, портрети, натюрморти, твори Пікассо або твори 19 століття; твір може належати більш ніж одній групі. Кожна група ідентифікується на ім'я, яке описує групу. Зрештою, галереї зберігають інформацію про клієнтів. Для кожного покупця зберігється його унікальне ім'я, адресу, загальну суму грошей, витрачених у галереї, а також художників та груп мистецтва, які подобаються покупцю.

Вимоги до проекту

Студент повинен розробити застосування відповідно до своєї теми. У проекті має зберігатись структурована інформація в базі даних і відображатись на сторінках сайту - каталог товарів (або послуг) і т.д.

Для спрощення процесу інформаційного наповненням сторінок і пошуком графічного матеріалу, можна використати, як аналог, готовий шаблон, або будь-який існуючий сайту зі змінами, які відповідають завданням проекту.

1. Застосування повинно бути реалізовано за трирівневою клієнт-серверною архітектурою.
2. Для ведення даних використовувати засоби SQL Server.
3. Для даних має використовуватись не менше трьох таблиць, одна з яких має бути асоційованою.
4. Для роботи з даними має використовуватись об’єктно-реляційне відображення Entity Framework (ORM) на основі Code First .
5. Для відображення схеми БД використовувати POCO-класи моделі даних, а для даних – контекст даних.
6. Для реалізації CRUD – операцій, вибірки та інших запитів використовувати LINQ або λ-вирази.
7. В клієнтській частині повинні використовуватись засоби валідації введених користувачем даних.

Приклад коду:

У репозиторії за адресою:

https://github.com/alysenko4317/ntu.net/tree/master/ Lab_5_ASP/webblog_sample

міститься код проекту, який описано у даному розділі.

ЛАБОРАТОРНА РОБОТА №6

Тема: Обробка динамічних даних з використанням технології ASP.NET Web API.

Мета: Навчитися створювати простий REST API веб-сервіс за допомогою ASP.NET та взаємодіяти з ним через клієнтську частину, реалізовану з використанням обраної технології.

ВСТУП

У цьому розділі буде показано, як за допомогою технології C# ASP.NET Core розробити простий REST API додаток. Розглянемо реалізацію unit-тестів для різних рівнів коду, відправлення json-відповідей, а також розгортання програми у Docker. На прикладі буде представлена спрощена модель сервісу з ремонту автомобілів, включаючи сутності працівників та звіти з ремонту, які надсилатимуться у відповідях.

Для прикладу буде створено спрощену модель сервісу з ремонту автомобілів, який включатиме моделі працівників, які виконують ремонт, та звіти з ремонту, які надсилатимуться у відповідях.

1.1 Підготовка

Для розробки цього додатку будемо використовувати MS Visual Studio 2019 або пізнішу версію. Взаємодія з базою даних буде здійснюватися за допомогою ORM-підходу, реалізованого на базі Entity Framework, тому знадобиться встановити наступні nuget-пакети:

Microsoft.EntityFrameworkCore
Microsoft.EntityFrameworkCore.SqlServer
Microsoft.EntityFrameworkCore.Tools

Для реалізації модульного тестування також знадобляться наступні nuget-пакети:

Microsoft.NET.Test.Sdk
Microsoft.NETCore.App
Moq
xunit
xunit.runner.visualstudio

Для встановлення перелічених вище пакетів потрібно зайти в оглядач пакетів NuGet, відкривши його через контекстне меню проекту, і вибравши там пункт «Управління пакетами NuGet».

1.2 Що таке REST?

REST – це популярний архітектурний підхід для створення API у сучасному світі.

REST розшифровується як REpresentational State Transfer (передача стану подання). Цей термін був вперше введений Роєм Філдінгом, одним із розробників протоколу HTTP. Головна ідея REST полягає у використанні стандартних методів HTTP для реалізації взаємодій між клієнтами та серверами. Це дозволяє ефективно використовувати можливості протоколу HTTP, забезпечуючи простоту та гнучкість у розробці мережевих додатків.

1.3 Протокол HTTP

Протокол HTTP є основою взаємодії у Всесвітній павутині. Коли ви вводите URL-адресу в браузері, наприклад www.google.com, ваш браузер відправляє запит на сервер, асоційований з цією URL-адресою. Відповідь сервера, що формується, має стандартний формат, визначений протоколом HTTP (Hyper Text Transfer Protocol).

Загалом, коли ви вводите URL, браузер виконує HTTP-запит типу GET до вказаного сервера. Відповідь сервера зазвичай містить дані у форматі HTML (Hyper Text Markup Language), які браузер потім інтерпретує та відображає на екрані.

Якщо ж ви взаємодієте з формою на веб-сторінці, наприклад, вводите дані чи обираєте елементи зі списку, то після натискання кнопки "Відправити" генерується HTTP-запит типу POST. Цей запит відправляє введені вами дані на сервер для обробки.

HTTP використовується не лише для завантаження веб-сторінок, але й як основа для взаємодії з веб-сервісами, включаючи RESTful API. Знання основ HTTP є важливим для розуміння того, як працюють веб-сервіси та як вони взаємодіють з клієнтами.

1.4 HTTP та RESTful веб-сервіси

Основною абстракцією, на якій фокусується HTTP, є ресурс. Ресурс – це будь-який об'єкт, який ви бажаєте зробити доступним у вашому додатку для зовнішнього світу.

Наприклад, у програмі для управління завданнями ресурсами можуть бути:

Конкретний користувач: інформація про окремого користувача, його профіль та налаштування.

Конкретне завдання: деталі окремого завдання, включаючи його статус, опис та відповідальну особу.

Список завдань: перелік усіх завдань, що належать певній групі чи користувачу.

Важливо зазначити, що підхід REST вимагає від розробників мислення у термінах ресурсів і їх представлення через URI. Такий підхід дозволяє використовувати стандартні HTTP-методи, такі як GET, POST, PUT, DELETE, для виконання операцій з ресурсами, що робить взаємодію з API інтуїтивно зрозумілою та структурованою.

Ключовим елементом тут є ідентифікація кожного ресурсу за допомогою унікального URI (універсального ідентифікатора ресурсу). Наведемо кілька прикладів:

Створення користувача: POST /users – запит POST до адреси /users служить для створення нового користувача.

Видалення користувача: DELETE /users/1 – запит DELETE до адреси /users/1 призначений для видалення користувача з ідентифікатором 1.

Отримання списку користувачів: GET /users – запит GET до адреси /users дозволяє отримати перелік усіх користувачів.

Отримання інформації про конкретного користувача: GET /users/1 – запит GET до ад-реси /users/1 забезпечує доступ до інформації про користувача з ідентифікатором 1.

1.5 Підхід до розробки веб-сервісів

Підхід REST вимагає від розробників мислити про додаток, виходячи з наступних ключових принципів:

Визначення ресурсів: Ідентифікуйте, які ресурси ви хочете зробити доступними для зовнішнього світу. Це можуть бути дані користувача, продукти, замовлення тощо.

Використання HTTP методів: Використовуйте методи HTTP (GET, POST, PUT, DELETE) для маніпулювання цими ресурсами. Кожен з цих методів має своє визначене призначення, наприклад, GET для отримання даних, POST для створення нових записів.

Аспекти реалізації REST веб-служб:

Формат обміну даними: REST не накладає обмежень на формат обміну даними. Хоча JSON є дуже популярним через свою легкість та читабельність, можна також використовувати XML або інші формати.

Транспорт: REST використовує HTTP як основний транспортний протокол, що забезпечує стандартизовані методи для доступу та маніпуляції ресурсами.

Визначення сервісу: REST є гнучкою архітектурою, що не має суворого стандарту для визначення сервісів. Це може бути недоліком, оскільки відсутність стандартизації може ускладнити розуміння API клієнтами. Однак, інструменти, такі як WADL і Swagger, вирішують цю проблему, надаючи інтуїтивно зрозумілі способи опису веб-сервісів.

1.6 Основи протоколу HTTP

HTTP (HyperText Transfer Protocol) – це текстовий протокол, який використовується для передачі даних у Всесвітній мережі. Через його текстову природу, запити та відповіді HTTP можуть бути прочитані та зрозумілі людиною, що спрощує розробку та відладку веб-додатків.

HTTP-запит (request) складається з трьох основних частин:

Рядок запиту (request Line): Перший рядок, який включає HTTP-метод (наприклад, GET чи POST), URI ресурсу та версію HTTP, наприклад: **GET /index.html HTTP/1.1**

Заголовки запиту (header fields): Містять додаткову інформацію про запит, наприклад, тип контенту, кукі, параметри авторизації тощо.

Тіло повідомлення (body): Необов'язкова частина, яка містить дані, які надсилаються серверу. Використовується в методах, таких як POST чи PUT. Заголовки й тіло повідомлення розділені одним порожнім рядком.

Приклад HTTP-запиту:

```
GET /index.html HTTP/1.1
Host: www.example.com
```

Аналогічно до запиту, HTTP-відповідь(response) включає:

Рядок стану (status line): Містить версію HTTP, код стану відповіді (наприклад, 200 чи 404) та текстовий опис статусу.

Поля заголовка відповіді (header fields): Аналогічно до заголовків запиту, містять додаткову інформацію про відповідь, наприклад, тип контенту, дату і час відповіді тощо.

Тіло повідомлення (body): Містить фактичні дані відповіді, наприклад, HTML-сторінку чи JSON-об'єкт.

Приклад HTTP-відповіді:

```
HTTP/1.1 200 OK
Content-Type: text/html

<html>
  <head><title>Example</title></head>
  <body><p>Example Page</p></body>
</html>
```

Версія HTTP в рядку запиту та стану позначає версію протоколу, яка використовується. Різні версії мають різні можливості та характеристики. Наприклад, HTTP/1.1 підтримує тривале з'єднання (keep-alive), в той час як HTTP/2 пропонує кращу продуктивність завдяки бінарному формату обміну даними і мультиплексуванню.

1.7 HTTP-методи

Метод, який використовується в HTTP-запиті, вказує, яку саме дію ви бажаєте виконати з вказаним ресурсом. Важливі приклади:

GET: Запитує представлення ресурсу. Використовується для отримання даних.

POST: Використовується для створення нового ресурсу.

PUT: Оновлює існуючий ресурс або створює новий, якщо він не існує.

DELETE: Видаляє вказаний ресурс.

HEAD: Аналогічно до GET, але відповідь не містить тіла. Використовується для отримання метаданих.

Це не виключний перелік методів, є й інші, але це ті, що використовуються найчастіше.

1.8 Налаштування підключення до СУБД

Для конфігурації підслючення до СУБД потрібно визначити клас ApplicationContext (реалізацію якого ми розглянемо далі) та вказати рядок підключення. Цей рядок підключення буде зберігатися у файлі appsettings.json. У класі ApplicationContext будуть визначені всі необхідні залежності для генерації міграцій. Рядок підключення важливий, оскільки він інформує застосунок про те, до якої бази даних звертатися і які параметри використовувати.

Щоб додати рядок підключення, необхідно відкрити файл appsettings.json та прописати наступне:

```
"ConnectionStrings": {
  "DefaultConnection":
     "Server=(localdb)\\mssqllocaldb;Database=testdb;Trusted_Connection=True;"
},
```

1.9 Опис шарів програмного коду додатку

Цей розділ надає стислий огляд аспектів, частково висвітлених у попередньому розділі, присвяченому лабораторній роботі №5 та архітектурному паттерну MVC. Також, при виконанні цієї лабораторної роботи ви познайомитеся з іншим архітектурним паттерном – Controller Service Repository (CSR), який допомагає розділити відповідальність між шарами додатку та дотримуватися принципів SOLID.

Моделі

У шарі моделей знаходитимуться сутності, які за допомогою Entity Framework будуть перетворені на таблиці в базі даних.

Для опису моделі в додатку достатньо визначити C# клас, з потрібними вам полями. Ці поля автоматично будуть перетворені в стовпці таблиці, а назва таблиці буде відповідати назві класу. Так задано за замовчуванням, але є спеціальні атрибути, які дають змогу гнучкіше налаштовувати зберігання даних у БД:

Перша модель, яка знадобитися для опису сервісу з ремонту - модель *співробітника*. Що вона буде собою являти?

- Унікальний ідентифікатор співробітника
- Ім'я співробітника
- Посада співробітника
- Номер телефону для зв'язку зі співробітником

Наступна модель для опису сервісу - *автомобілі*, які надходитимуть на ремонт.

- Унікальний ідентифікатор автомобіля
- Назва автомобіля
- Номер автомобіля

І остання модель, яку ми вже будемо відсилати – це документ (виписка) з ремонту.

- Унікальний ідентифікатор документа
- Співробітник, який обслуговував автомобіль
- Автомобіль, який був на ремонті

Щоб моделі потрапили в базу даних, необхідно створити міграцію.

Міграція – це опис того, як і що буде записано в базу даних. За допомогою Entity Framework міграції можна генерувати автоматично. Для цього в пакетному менеджері треба прописати команду `Add-Migration`. Після цього Entity Framework згенерує міграцію за вашими моделями, які зазначені в класі, успадкованому від `DbContext`. Щоб застосувати міграцію (тобто виконати), треба виконати команду `Update-Database`, після цього ваші дані потраплять у базу даних (як це використовувати буде описано далі).

Контролери

Контролер – це посередник між бізнес-логікою, або базою даних та front-end частиною додатку. Отримуючи запит від front-end частини додатку, контролер обробляє його - викликає необхідні сервіси для виконання якоїсь бізнес-логіки та надсилає отримані дані назад до front-end частини як відповідь.

У якості даних, які повертається до front-end частини додатку, в контролерах буде використовуватися тип `json`. Для цього достатньо в операторі `return` прописати наступний код: `new JsonResult(ваш_об'єкт)`

У цій лабораторній роботі ми розглянемо обробку HTTP-запитів таких типів як `GET`, `POST`, `PUT` та `DELETE` на конкретному прикладі. Під час обробки GET-запиту ми вибиратимемо всі наявні документи та відсилатимемо їх до фронтенд частини додатку. Для POST-запитів будемо активувати сервіс з ремонту автомобіля та повертати детальну виписку зі звітом про виконаний ремонт. Запити PUT відповідатимуть за оновлення існуючих документів, тоді як DELETE дозволить видаляти документи.

Подання

У цьому проекті подання відсутні, на відміну від проекту, який ми розглядали у лабораторній роботі №5. У тому проекті серверна частина генерувала відповіді на запити у вигляді HTML-сторінок, які потім відображалися у браузері. У поточному проекті, натомість, відповіді сервера складатимуться виключно з даних, які будуть надсилатися клієнту у форматі JSON.

DAO (Репозиторії)

Репозиторії слугують як посередники для взаємодії з базою даних, що дозволяє уникнути безпосередньої взаємодії користувача з даними. Це важливо для приховування деталей внутрішньої логіки обробки даних, автоматизації ключових процесів роботи з БД та забезпечення безпеки даних.

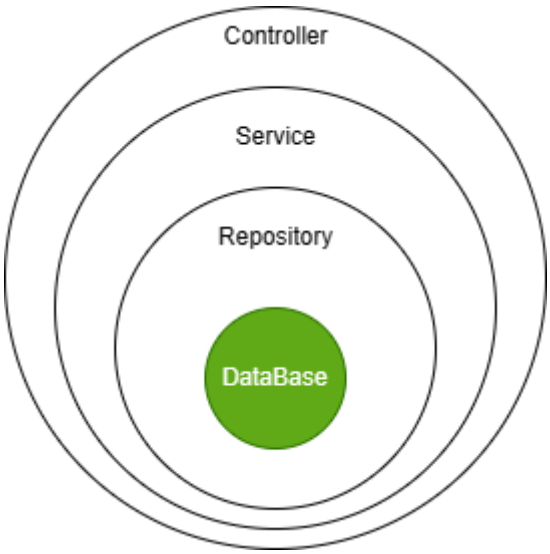
У рамках цього застосунку реалізований універсальний репозиторій, який може працювати з будь-якою моделлю. Цей репозиторій надає базові операції, такі як отримання (get), отримання всіх даних (get all), оновлення (update), створення (create) та видалення (delete) записів.

Сервіси

Сервіси – це такі класи, які містять у собі *бізнес-логіку* додатку. Являють собою клас із методами для вирішення того чи іншого завдання.

Як приклад сервісу, реалізовано клас лише з одним методом `Work()`. Цей метод імітує роботу сервісу з ремонту машин. У цьому методі призначається робітник для ремонту певного автомобілю і заповнюється документ (звіт) про виконаний ремонт.

Модель архітектурного патерну CSR зручно уявляти у вигляді сфери з різних шарів, кожен з яких по ланцюжку знаходиться "ближче до даних" (поступове зменшення рівня абстракції над взаємодією з базою даних):



Архітектурні патерни Controller Service Repository (CSR) та Model-View-Controller (MVC) часто використовуються разом, оскільки обидва включають компонент контролера, але виконують різні функції у розробці програмного забезпечення.

Спільний Компонент - Контролер: У обох паттернах, контролер відповідає за приймання запитів від користувача та направлення їх до відповідних сервісів або моделей. Це ключовий елемент у обох архітектурах, який сприяє розділенню відповідальності та забезпечує централізоване управління вхідними даними.

Взаємодія з MVC: У паттерні MVC, контролер спрямовує дані від користувача до моделі та вибирає відповідне подання (View) для відображення результату. MVC фокусується на розділенні бізнес-логіки (Model), інтерфейсу користувача (View) та обробки користувацьких запитів (Controller).

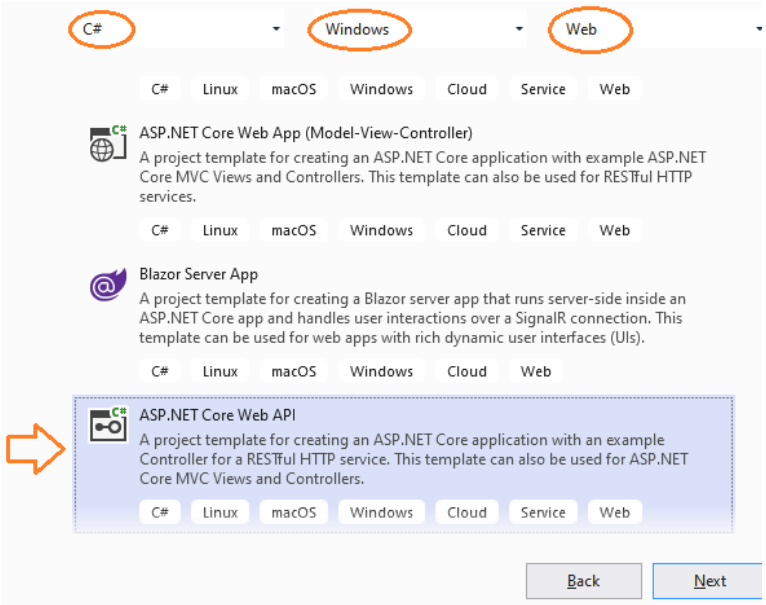
Взаємодія з CSR: У CSR, контролер відіграє роль посередника, передаючи дії від користувачів до сервісного шару, який обробляє бізнес-логіку, а репозиторії займаються взаємодією з базою даних. Тут основний акцент робиться на чіткому розділенні бізнес-логіки (Service) та взаємодії з даними (Repository).

Використання CSR разом з MVC є ефективним, оскільки об'єднує переваги обох паттернів: чітке розділення логіки обробки даних, бізнес-логіки та подання. Це сприяє створенню добре структурованого, здатного до масштабування та пфдтримки програмного забезпечення.

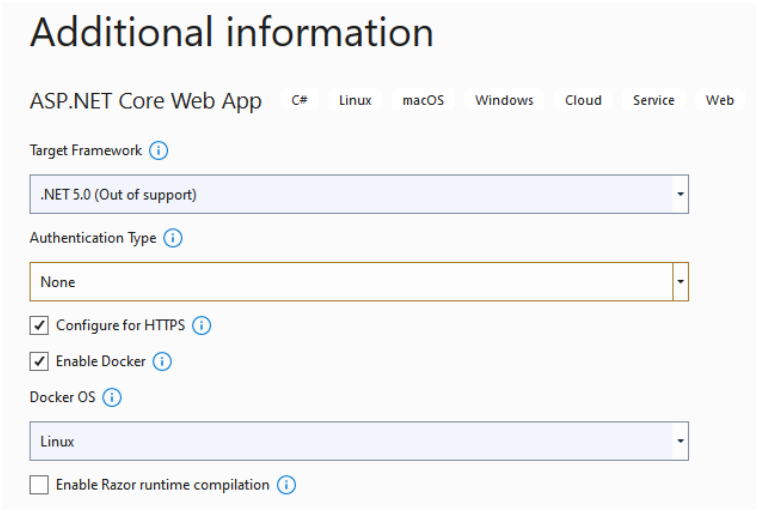
Тепер, з огляду на описані компоненти, можна приступати до реалізації додатку.

1.10 Створення проекту

При створенні нового проєкту, як і в попередній роботі, обираємо веб-додаток, тільки тепер необхідно вибрати шаблон ASP.NET Core Web API (у попередній роботі ми використовували шаблон ASP.NET Core App MVC):

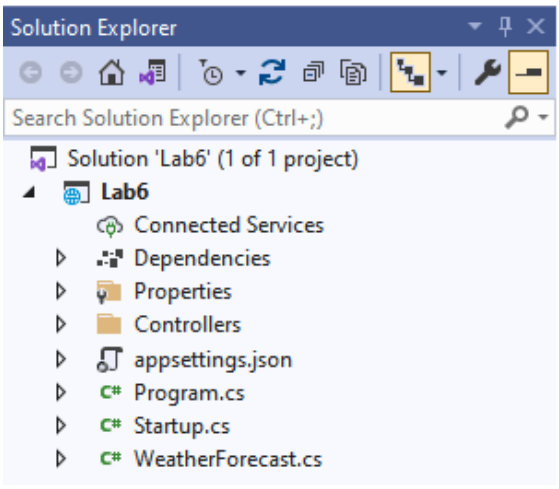


Вигадуємо назву проєкту, та обираємо папку, де він зберігатиметься:



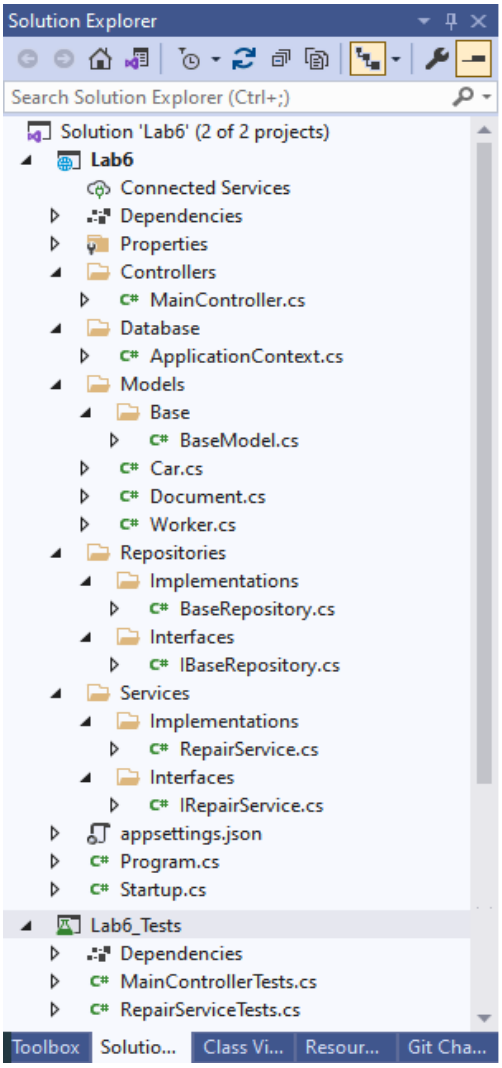
Далі приступимо до створення структури додатка.

Весь код розподілений на окремі шари, організовані в різні папки, а unit-тести винесені в окремий проект. Тож, після створення основного проєкту, вашим завданням буде створити запропоновану нижче структуру проєкту, а саме — створити всі необхідні папки та cs-файли класів.



Структура проєкту після його створенняв IDE.

Треба привезти структуру до вигляду, наведеного праворуч, тобто треба створити всі відсутні папки та файли.



1.11 Моделі

Для реалізації моделей зроблено абстрактний клас BaseModel. Він знадобиться в майбутньому для коректного успадкування, а також у ньому прописано Id кожної моделі (це допомагає не дублювати код):

```
using System;

public abstract class BaseModel {
    public Guid Id { get; set; }
}
```

Далі код моделей:

```
using Lab6.Models.Base;

public class Worker : BaseModel
{
    public string Name { get; set; }
    public string Position { get; set; }
    public string Telephone { get; set; }
}
```

```
using Lab6.Models.Base;

public class Car : BaseModel
{
    public string Name { get; set; }
    public string Number { get; set; }
}
```

```
using System;
using Lab6.Models.Base;

public class Document : BaseModel
{
    public Guid CarId { get; set; }
    public Guid WorkerId { get; set; }
    public virtual Car Car { get; set; }
    public virtual Worker Worker { get; set; }
}
```

1.12 Application Context

ApplicationContext є класом, що успадковується від DbContext, який є ключовим для Entity Framework, яка використовується для взаємодії з базою даних. Усередині ApplicationContext визначаються властивості типу DbSet для кожної моделі домену. Ці DbSet визначають, які саме C# класи представляють моделі предметної області та як вони мають відображатися у базі даних. Використання DbSet дозволяє Entity Framework точно знати, які сутності мають бути включені до бази даних.

```
using Lab6.Models;
using Microsoft.EntityFrameworkCore;

public class ApplicationContext : DbContext
{
    public DbSet<Car> Cars { get; set; }
    public DbSet<Document> Documents { get; set; }
    public DbSet<Worker> Workers { get; set; }

    public ApplicationContext(DbContextOptions<ApplicationContext> options)
        : base(options)
    {
        Database.EnsureCreated();
    }
}
```

1.13 Репозиторій

Як було зазначено раніше, у нашій системі передбачено універсальний репозиторій, спроможний працювати з будь-якою моделлю. Для забезпечення гнучкості та інкапсуляції логіки роботи репозиторію, реалізуємо низку відповідних інтерфейсів.

Інтерфейс IBaseRepository визначає основні операції, які можна виконувати з моделями:

```
using Lab6.Models.Base;

public interface IBaseRepository<TDbModel> where TDbModel : BaseModel
{
    public List<TDbModel> GetAll();
    public TDbModel Get(Guid id);
    public TDbModel Create(TDbModel model);
    public TDbModel Update(TDbModel model);
    public void Delete(Guid id);
}
```

Реалізація цього інтерфейсу забезпечує стандартні CRUD-операції (створення, читання, оновлення, видалення) для кожної моделі:

```
public class BaseRepository<TDbModel>
    : IBaseRepository<TDbModel> where TDbModel : BaseModel
{
    protected ApplicationContext Context { get; set; }

    public BaseRepository(ApplicationContext context) {
        Context = context;
    }
```

(продовження)

```
public TDbModel Create(TDbModel model) {
        Context.Set<TDbModel>().Add(model);
        Context.SaveChanges();
        return model;
    }

    public void Delete(Guid id) {
        var toDelete = Context.Set<TDbModel>().FirstOrDefault(m => m.Id == id);
        Context.Set<TDbModel>().Remove(toDelete);
        Context.SaveChanges();
    }

    public List<TDbModel> GetAll() {
        return Context.Set<TDbModel>().ToList();
    }

    public TDbModel Update(TDbModel model) {
        var toUpdate = Context.Set<TDbModel>().FirstOrDefault(m => m.Id ==
model.Id);
        if (toUpdate != null)
            toUpdate = model;
        Context.Update(toUpdate);
        Context.SaveChanges();
        return toUpdate;
    }

    public virtual TDbModel Get(Guid id) {
        return Context.Set<TDbModel>().FirstOrDefault(m => m.Id == id);
    }
}
```

1.14 Сервіс

Сервіс у нашій системі, подібно до репозиторію, включає в себе інтерфейс та його реалізацію. Це дозволяє абстрагувати логіку роботи сервісу від його конкретної імплементації, забезпечуючи гнучкість та можливість заміни реалізації без змін у інших частинах системи.

Інтерфейс IRepairService визначає базовий контракт для сервісу ремонту:

```
public interface IRepairService {
    public void Work();
}
```

Реалізація IRepairService:

```
using System;
using Lab6.Models;
using Lab6.Repositories.Interfaces;
using Lab6.Services.Interfaces;

public class RepairService : IRepairService
{
    private IBaseRepository<Document> Documents { get; set; }
    private IBaseRepository<Car> Cars { get; set; }
    private IBaseRepository<Worker> Workers { get; set; }

    public RepairService(IBaseRepository<Document> documents,
                         IBaseRepository<Car> cars,
                         IBaseRepository<Worker> workers) {
        Documents = documents;
        Cars = cars;
        Workers = workers;
    }

    public void Work()
    {
        var rand = new Random();
        var carId = Guid.NewGuid();
        var workerId = Guid.NewGuid();

        Cars.Create(new Car {
            Id = carId,
            Name = String.Format($"Car{rand.Next()}"),
            Number = String.Format($"{rand.Next()}")
        });

        Workers.Create(new Worker {
            Id = workerId,
            Name = String.Format($"Worker{rand.Next()}"),
            Position = String.Format($"Position{rand.Next()}"),
            Telephone = String.Format(
                $"8916{rand.Next()}{rand.Next()}{rand.Next()}" +
                $"{rand.Next()}{rand.Next()}{rand.Next()}{rand.Next()}")
        });

        var car = Cars.Get(carId);
        var worker = Workers.Get(workerId);

        Documents.Create(new Document {
            CarId = car.Id,
            WorkerId = worker.Id,
            Car = car,
            Worker = worker
        });
    }
}
```

1.15 Контролер

У цьому додатку використовується лише один контролер, проте його шаблон можна застосувати для створення будь-якої кількості контролерів. Коли застосунок запущено, для звернення до методу контролера з фронтенд-частини додатку достатньо відправити запит наступного вигляду:

ДоменнеІм'я/НазваКонтролера/НазваМетоду?Параметри(якщо вони є)

Шляхи до методів контролерів гнучко налаштовуються за допомогою спеціальних атрибутів маршрутизації.

```
[ApiController]
MainController.cs
[Route("[controller]")]
public class MainController : ControllerBase
{
    private IRepairService RepairService { get; set; }
    private IBaseRepository<Document> Documents { get; set; }

    public MainController(IRepairService repairService,
                          IBaseRepository<Document> document) {
        RepairService = repairService;
        Documents = document;
    }

    [HttpGet]
    public JsonResult Get() {
        return new JsonResult(Documents.GetAll());
    }

    [HttpPost]
    public JsonResult Post() {
        RepairService.Work();
        return new JsonResult("Work was successfully done");
    }

    [HttpPut]
    public JsonResult Put(Document doc) {
        bool success = true;
        var document = Documents.Get(doc.Id);
        try {
            if (document != null)
                document = Documents.Update(doc);
            else
                success = false;
        }
        catch (Exception) {
            success = false;
        }
        return success
            ? new JsonResult($"Update successful {document.Id}")
            : new JsonResult("Update was not successful");
    }
```

(продовження)

```
    [HttpDelete]
    public JsonResult Delete(Guid id) {
        bool success = true;
        var document = Documents.Get(id);
        try {
            if (document != null)
                Documents.Delete(document.Id);
            else
                success = false;
        }
        catch (Exception) {
            success = false;
        }
        return success
            ? new JsonResult("Delete successful")
            : new JsonResult("Delete was not successful");
    }
}
```

1.16 Налаштування залежностей та їх впровадження

Тепер перейдемо до обговорення процесу впровадження залежностей. Адекватне налаштування залежностей в проекті спрощує його архітектуру та допомагає уникнути написання зайвого коду. Усі залежності визначаються у файлі Startup.cs.

Які залежності слід визначити? Можна створити зв'язок між інтерфейсом репозиторію та його конкретними реалізаціями для кожної моделі. Також можна зв'язати інтерфейси сервісів з їхніми реалізаціями.

Крім того, у файлі Startup.cs прописуються параметри (пам’ятаєте рядок підключення, згаданий на початку) для з’єднання з СУБД.

Зауваження: Перед запуском додатку не забудьте створити базу даних. Для цього в консолі диспетчера пакетів потрібно виконати наступні команди:

Add-Migration init (або будь-яке інше ім'я)
Update-Database

Ось і все, якщо всі кроки виконані коректно, то Ви створили свою базу даних. Також ці команди використовуються і для її оновлення, якщо ваші моделі зміняться.

Файл Startup.cs має мати наступний вигляд:

```
using Lab6.Database;
using Lab6.Models;
using Lab6.Repositories.Implementations;
using Lab6.Repositories.Interfaces;
using Lab6.Services.Implementations;
using Lab6.Services.Interfaces;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.OpenApi.Models;

namespace Lab6 {

public class Startup {

    public Startup(IConfiguration configuration) {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    // This method gets called by the runtime.
    // Use this method to add services to the container.
    public void ConfigureServices(IServiceCollection services) {
        string connection = Configuration.GetConnectionString("DefaultConnection");
        services.AddMvc();
        services.AddDbContext<ApplicationContext>(options => {
            options.UseSqlServer(connection);
        });
        services.AddTransient<IRepairService, RepairService>();
        services.AddTransient<IBaseRepository<Document>, BaseRepository<Document>>();
        services.AddTransient<IBaseRepository<Car>, BaseRepository<Car>>();
        services.AddTransient<IBaseRepository<Worker>, BaseRepository<Worker>>();
        services.AddSwaggerGen(c => {
            c.SwaggerDoc("v1", new OpenApiInfo { Title = "Lab6", Version = "v1" });
        });
    }

    // This method gets called by the runtime.
    // Use this method to configure the HTTP request pipeline.
    public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
        if (env.IsDevelopment()) {
            app.UseDeveloperExceptionPage();
            app.UseSwagger();
            app.UseSwaggerUI(c => c.SwaggerEndpoint("/swagger/v1/swagger.json", "Lab6
v1"));
        }
        app.UseHttpsRedirection();
        app.UseRouting();
        app.UseAuthorization();
        app.UseEndpoints(endpoints => {
            endpoints.MapControllers();
        });
    }}}
```

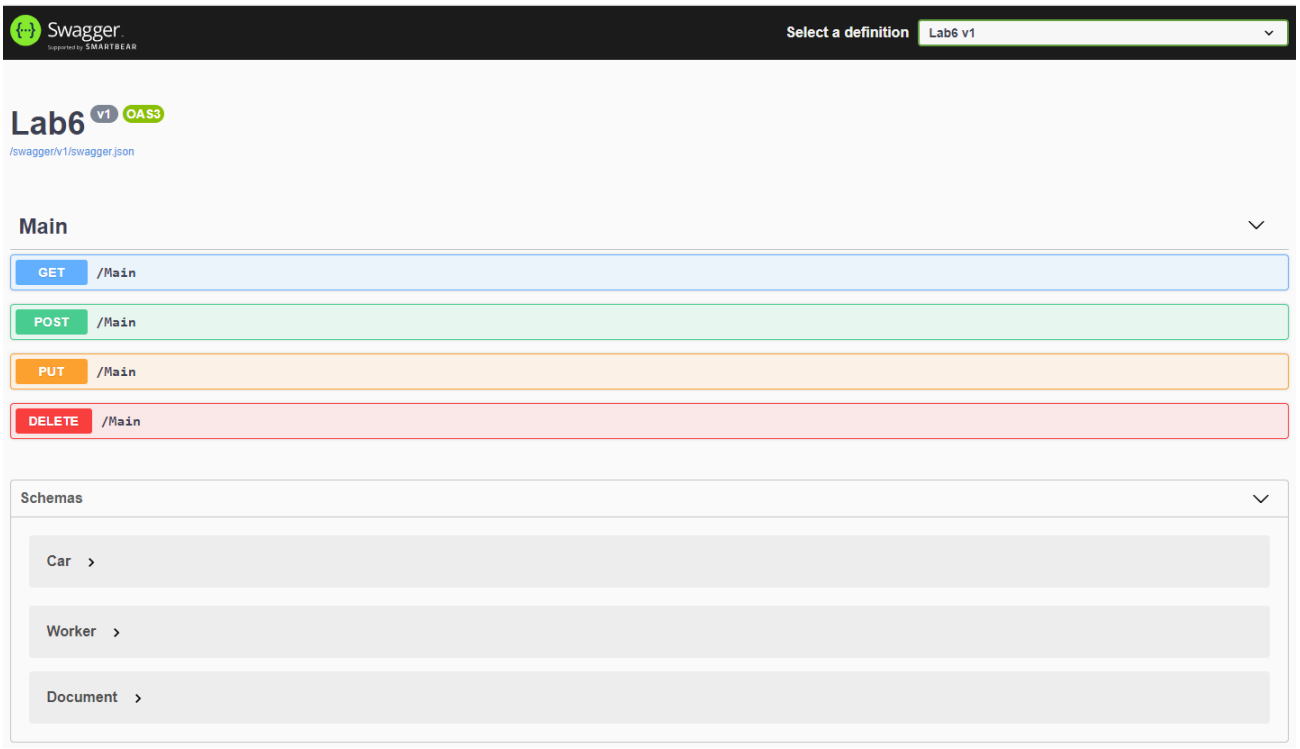
1.17 Перевірка працездатності додатку

Як же тестувати API, якщо у нас відсутній користувацький інтерфейс? На щастя, існує інструмент, який дозволяє ефективно вирішити це питання. Swagger UI – це потужний інструмент, який інтегрований у .NET додатки за замовчуванням при використанні шаблону проекту ASP.NET Core Web API. Він створює інтерактивну консоль API, яка дозволяє проводити експерименти з запитами в реальному часі.

Завдяки Swagger UI, можна детально описати параметри запитів для кожного з HTTP-методів, таких як GET, POST, PUT та DELETE, а також визначити очікуваний формат відповідей. Наприклад, для методу GET можна описати параметри запиту, як-от рядок запиту (query string), заголовки (headers) або параметри шляху (path parameters), і вказати формат відповіді, такий як JSON або XML. Swagger UI забезпечує ясність і послідовність у документації API, що дозволяє розробникам ефективно використовувати API у своїх додатках.

Використання Swagger UI не лише спрощує тестування API, але й актуалізує документацію, оскільки ви можете легко оновлювати її з урахуванням змін у вашому API. Це робить Swagger UI незамінним інструментом для розробки та тестування сучасних веб-додатків.

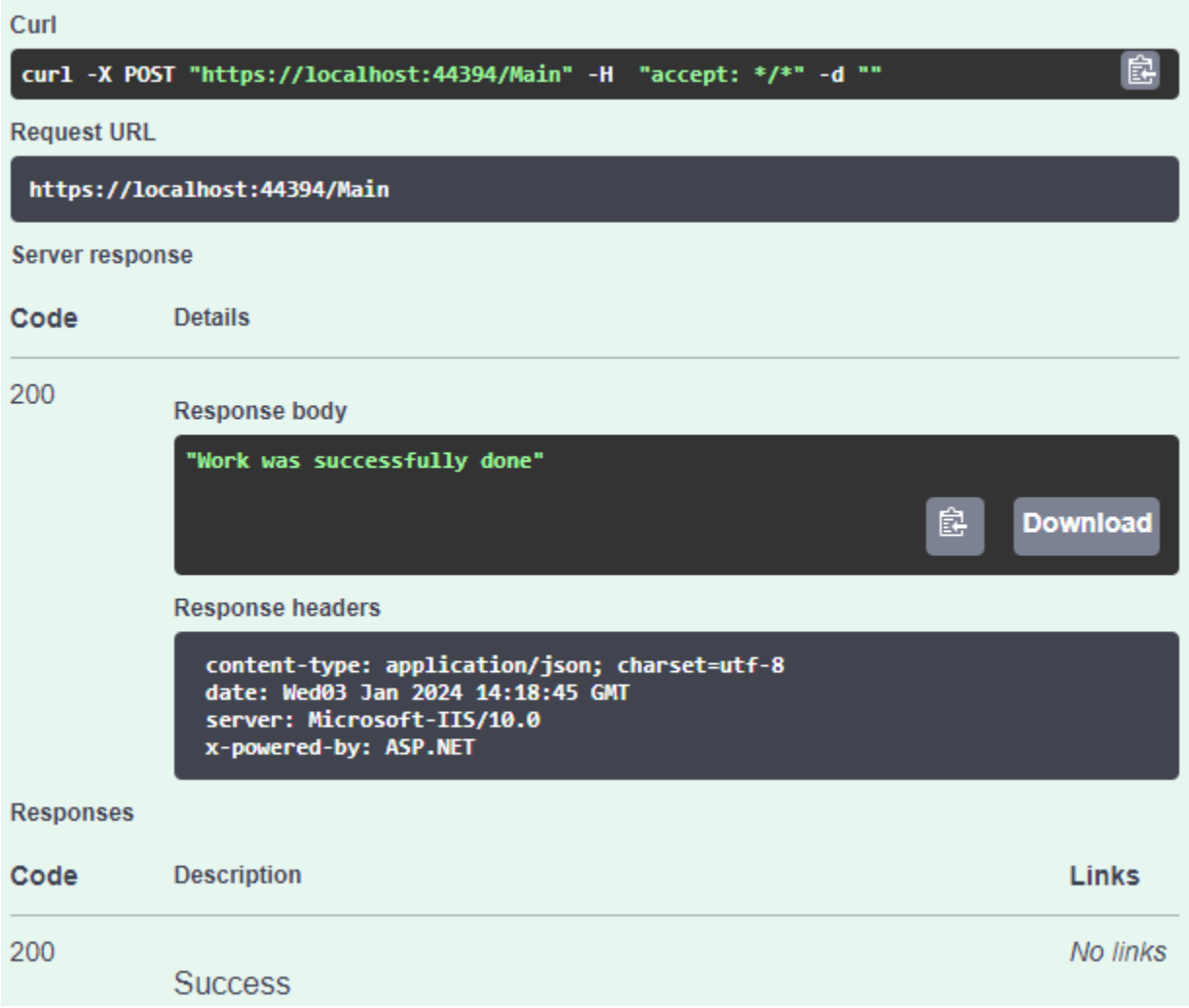
Натисніть **Ctrl+F5** для запуску сервера. Через кілька секунд відкриється веб-браузер і початкова сторінка додатку, це буде сторінка Swagger UI:



Тепер натисніть кнопку Post, а у відкрившийся секції натисніть Try It Out. Після цього з'явиться кнопка Execute, яку також потрібно натиснути. Буде виконано POST-запит, який у нашому випадку не передбачає передачі параметрів. Метод контролера, що обробляє цей запит, виглядає наступним чином:

```
[HttpPost]
public JsonResult Post() {
    RepairService.Work();
    return new JsonResult("Work was successfully done");
}
```

У інтерфейсі Swagger UI ви побачите результат виконання запиту:

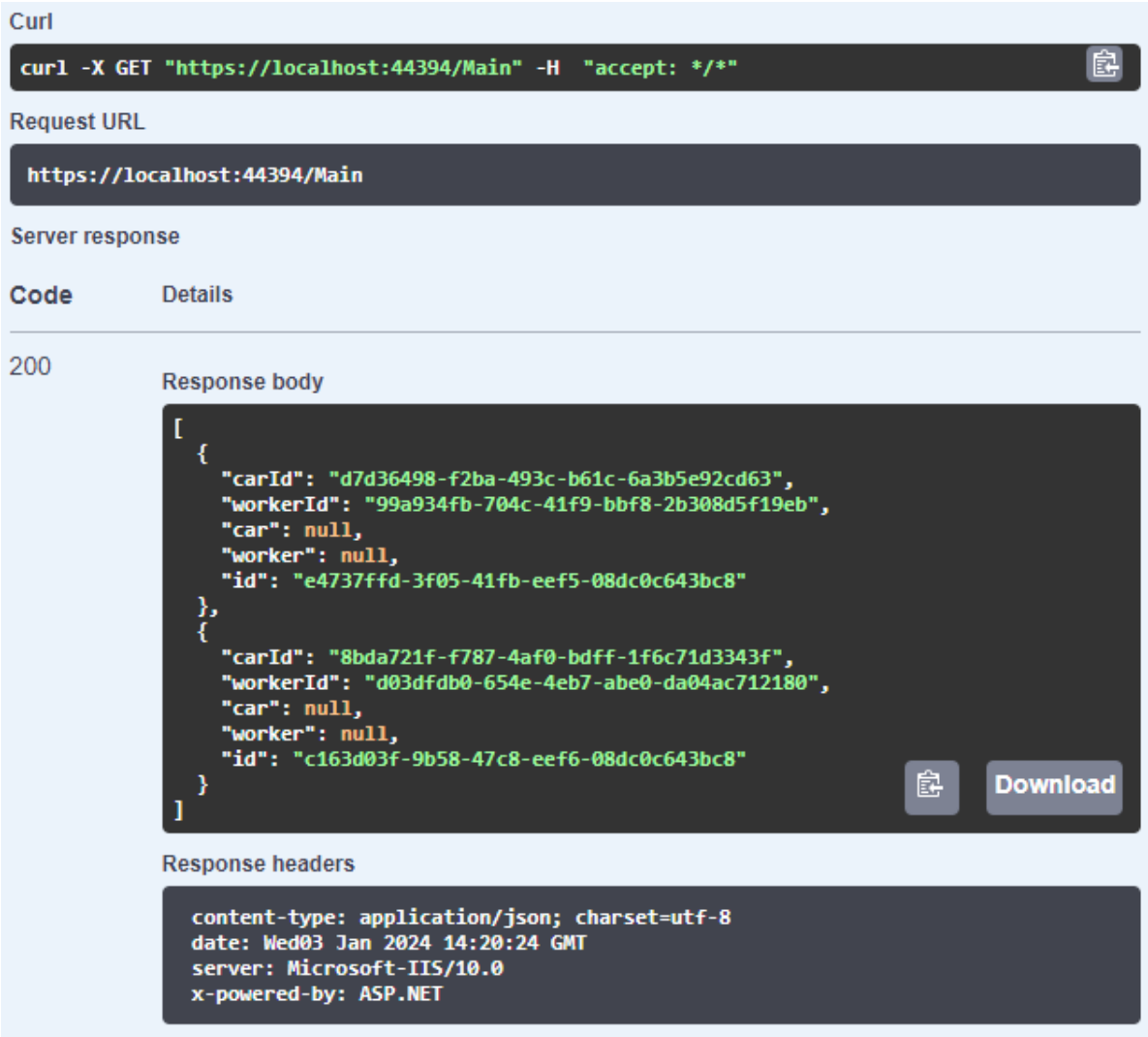


Спробуйте повторити цю операцію, відправивши POST-запит ще раз. У нашому випадку, при кожному надходженні POST-запиту умовно виконується ремонт автомобіля, інформація про який зберігається у базі даних.

Тепер спробуйте отримати всі звіти про виконані ремонти. Оскільки ми відправили POST-запит двічі, в базі даних має бути збережено два звіти. Для цього надішліть GET-запит. У коді обробки цього запиту ми не передбачали обробку додаткових параметрів, хоча це можна зробити, наприклад, для фільтрації списку звітів за певною ознакою:

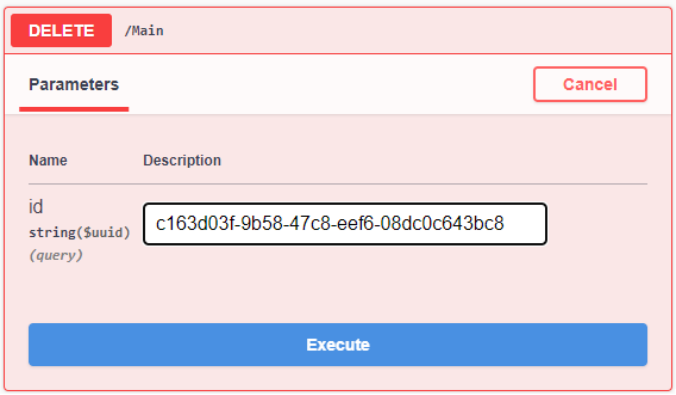
```
[HttpGet]
public JsonResult Get() {
    return new JsonResult(Documents.GetAll());
}
```

Знов таки після натискання кнопки Execute ми побачимо результат виконання запиту:

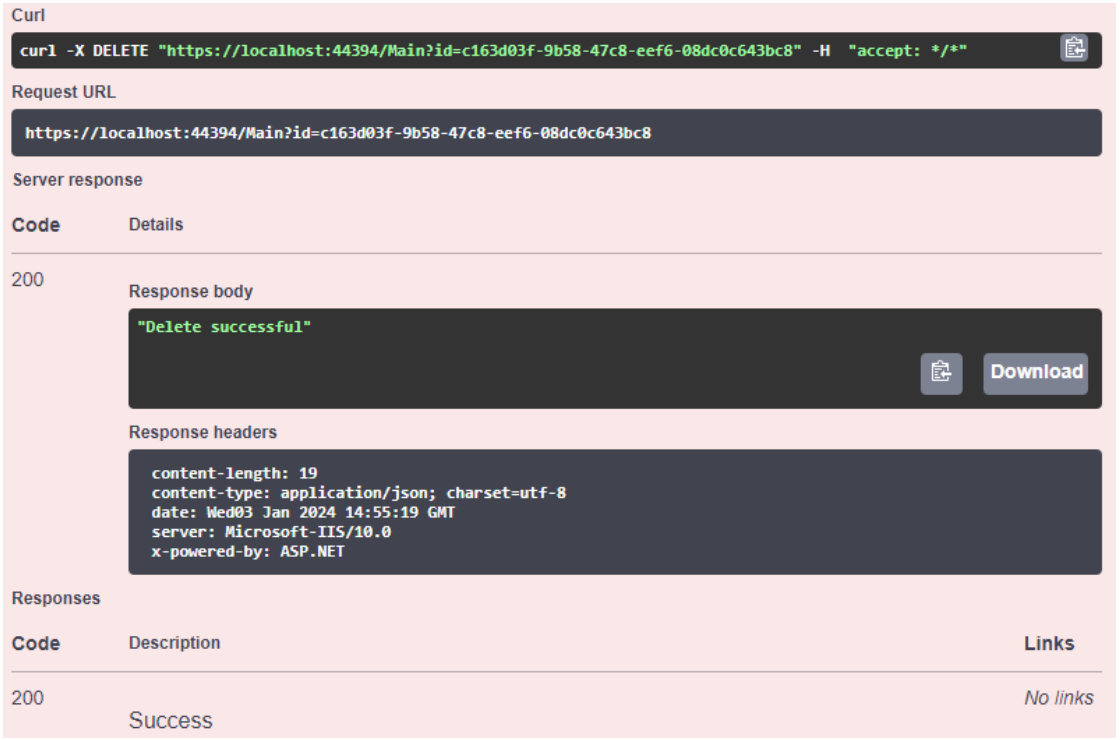


Як бачимо, у відповіді ми отримали перелік звітів про виконані ремонти у форматі JSON. Таким чином, ми отримали тільки дані, а як їх буде візуалізовано або використано — залежить від клієнта, який користується нашим REST API. Тобто це може бути додаток, розроблений на будь-якій мові програмування і технології.

Далі можемо перевірити функціональність обробки DELETE-запиту. У цьому випадку необхідно передати один параметр — ID документа (звіту про виконаний ремонт), який треба видалити. Як бачимо з результату виконання запиту GET в нас збережено два звіти з ідентифікаторами e4737ffd-3f05-41fb-eef5-08dc0c643bc8 та c163d03f-9b58-47c8-eef6-08dc0c643bc8. Спробуємо видалити перший:



Результат виконання запиту успішний:



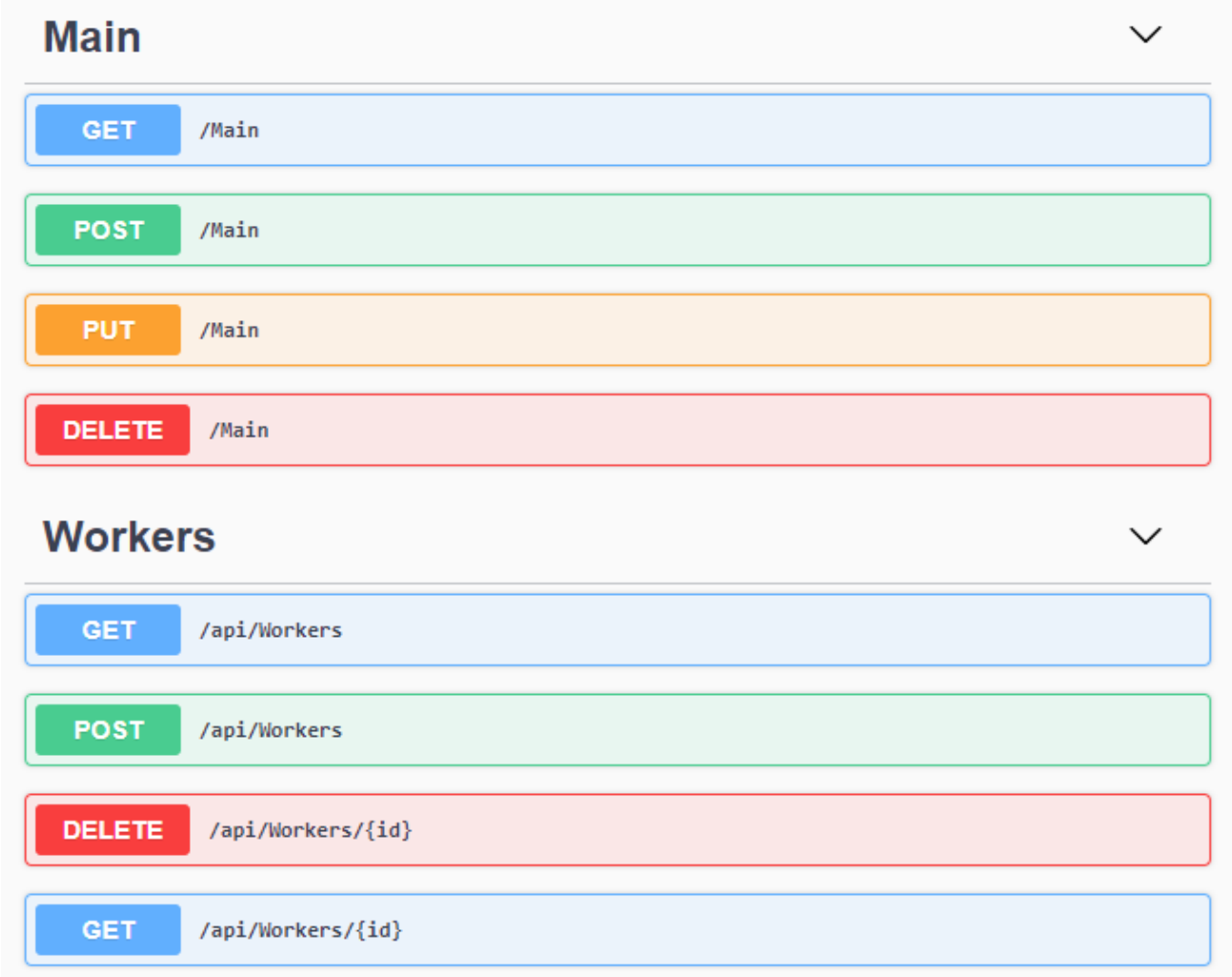
1.18 Закінчення проекту

Отже, ми успішно запустили та протестували наш веб-сервіс. Розглядаючи результати GET-запиту, можна помітити, що в звітах про виконані ремонти відсутня повна інформація про конкретні автомобілі та працівників, хоча ідентифікатори цих сутностей присутні. Така ситуація виникає через те, що Entity Framework за замовчанням не завантажує пов'язані сутності з бази даних автоматично.

Наступним кроком буде створення додаткового контролера, який дозволить нашому веб-API реєструвати співробітників, які виконуватимуть ремонт автомобілів. Його код виглядатиме так:

```
[Route("api/[controller]")]
[ApiController]
public class WorkersController : ControllerBase
{
    private static List<Worker> workers = new List<Worker>();

    [HttpGet]   // GET: api/Workers
    public ActionResult<IEnumerable<Worker>> GetWorkers() {
        return workers;
    }

    [HttpPost]   // POST: api/Workers
    public ActionResult<Worker> PostWorker(Worker worker) {
        worker.Id = Guid.NewGuid();
        workers.Add(worker);
        return CreatedAtAction("GetWorker", new { id = worker.Id }, worker);
    }

    [HttpDelete("{id}")]   // DELETE: api/Workers/5
    public ActionResult<Worker> DeleteWorker(Guid id) {
        var worker = workers.FirstOrDefault(w => w.Id == id);
        if (worker == null)
            return NotFound();
        workers.Remove(worker);
        return worker;
    }

    [HttpGet("{id}")]   // GET: api/Workers/5
    public ActionResult<Worker> GetWorker(Guid id) {
        var worker = workers.FirstOrDefault(w => w.Id == id);
        if (worker == null)
            return NotFound();
        return worker;
    }
}
```

Якщо зараз запустити проект на виконання, у веб-інтерфейсі Swagger з’явиться нова секція, яка відповідає маршрутам, зв’язаним з методами дій нового контролера WorkersController:

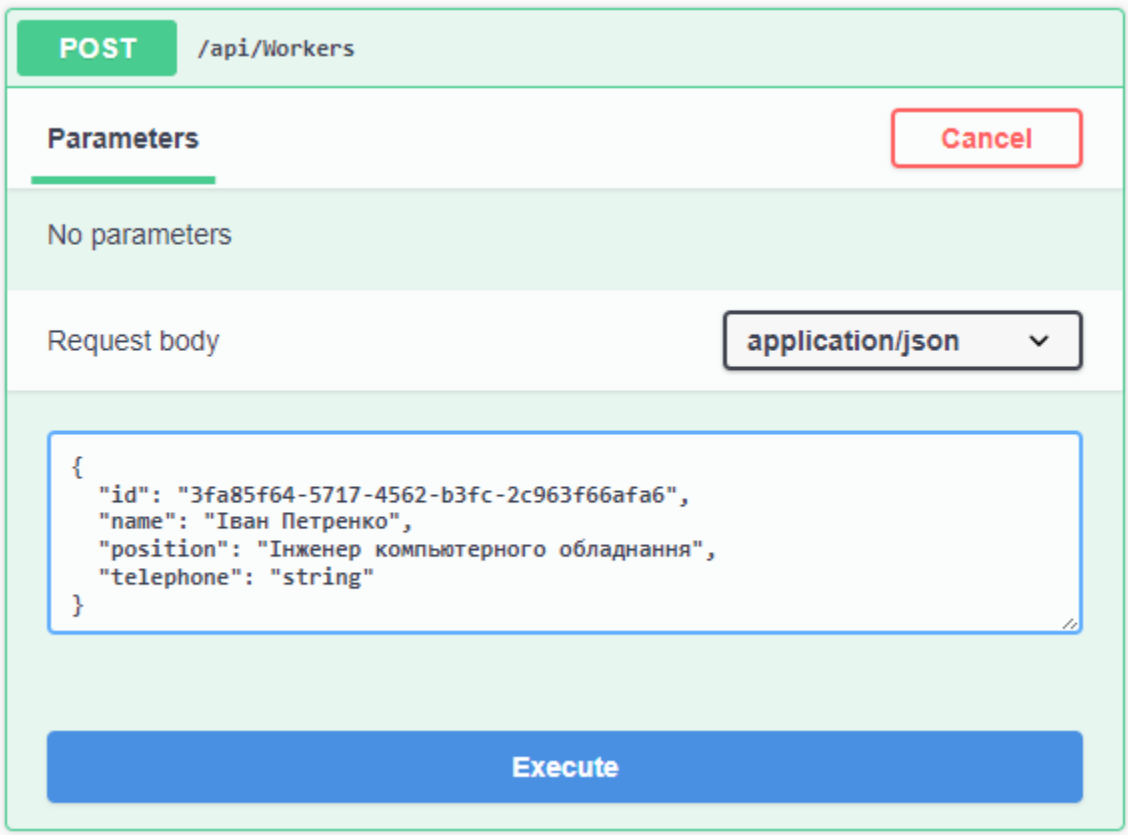


Можна помітити, що в порівнянні з секцією для запитів, які обробляються контролером MainController, у секції, яка відповідає контролеру WorkersController, праворуч від кнопок з назвами HTTP-методів, з’явилися шаблони URL-адрес. З цих шаблонів можна зрозуміти, як саме повинні передаватися параметри для певного запиту. Ці деталізовані описи з'явились завдяки коментарям XML-документації, які ми додали до кожного з методів дій відповідного контролера.

Документування коду вкрай важливе для розробників, які використовують ваш API. Чіткі та інформативні коментарі в коді не лише допомагають іншим розробникам зрозуміти ваш код, але й автоматично використовуються Swagger для генерації відповідної документації API. Це робить процес інтеграції з вашим API більш зрозумілим та простим, забезпечуючи кращий досвід розробки.

Спробуйте виконати GET-запит до контролера `WorkersController` і ви побачите результат – перелік співробітників, які вже зареєстровані в системі.

Далі відкрийте секцію POST і створіть нового працівника, внісши відповідні дані в тіло запиту та виконавши його:



Після цього знову виконайте GET-запит для контролера `WorkersController` і переконайтеся, що інформація про новоствореного працівника була успішно збережена у системі.

В поточній версії нашого додатку при формуванні звіту про виконаний ремонт кожного разу створюються нові сутності співробітника та автомобіля з випадково згенерованими даними. Проте, для більш реалістичної моделі роботи, ми можемо передбачити можливість передачі цих даних через API, над яким працюємо.

В контролері `MainController` є метод `Post` який викликає метод `Work` з сервису `IRepairService`, який фактично імітує виконання ремонту авто:

```
[HttpPost]  // POST: api/Main
public JsonResult Post([FromBody] RepairRequest request) {
    RepairService.Work();
    return new JsonResult("Work was successfully done");
}
```

Так як звіт формується певним працівником, інформація про нього повинна вже існувати в БД у відповідній таблиці и прив'язуватись до документу який створюється методом Work. По-друге - також повинна створюватись сутність машини у відповідповідній таблиці, при чому хотілось би щоб якщо певна машина вже колись була в ремонті то вдруге щоб така ж сама запис в таблиці машин не створювалась (перевіряти можна по реєстраційному номеру, наприклад).

Для виправлення і адаптації коду методу Work, нам потрібно змінити логіку таким чином, щоб він приймав ID існуючого працівника, реєстраційний номер та назву автомобіля. Якщо автомобіль із заданим реєстраційним номером вже існує в БД, то ми повинні використати існуючий запис, інакше - створити новий. Ось приклад, як можна змінити код методу Work():

```
public void Work(Guid workerId, string carName, string carRegistrationNumber)
{
    var rand = new Random();

    // перевіряємо, чи існує працівник
    var worker = Workers.Get(workerId);
    if (worker == null)
        throw new Exception("Worker not found");

    // перевіряємо, чи існує автомобіль з заданим номером
    var car = Cars.GetAll().FirstOrDefault(c => c.Number == carRegistrationNumber);
    if (car == null) {
        // якщо автомобіль не знайдено, створюємо новий
        car = new Car {
            Id = Guid.NewGuid(),
            Name = carName,
            Number = carRegistrationNumber
        };
        Cars.Create(car);
    }

    // створюємо документ про ремонт
    Documents.Create(new Document {
        CarId = car.Id,
        WorkerId = worker.Id,
        Car = car,
        Worker = worker
    });}
```

Примітка: так як сігнатура методу Work змінилась – тепер він приймає два параметри, необхідно також оновити визначення інтерфейсу IRepairService .

Слід також адаптувати метод Post() у контролері MainController щоб викликався оновлений метод Work(). Потрібно змінити його таким чином, щоб він приймав необхідні параметри для ремонту автомобіля. В прикладі ціми параметрами є ідентифікатор працівника та ім'я (реєстраційний номер) автомобіля, але за необхідності можна додати й іншу інформацію наприклад про модель та колір, додаткові нотатки, тощо. Тож оновлений код методу Post() у контролері MainController виглядатиме так:

```
[HttpPost]
public JsonResult Post([FromBody] RepairRequest request) {
    RepairService.Work(rq.WorkerId, rq.CarName, rq.CarRegistrationNumber);
    return new JsonResult("Work was successfully done");
}
```

Передача усіх необхідних даних буде відбуватись в тілі запита. Ми можемо використати спеціальну модель, щоб інкапсулювати ці дані; назвемо цей клас моделі RepairRequest, а його визначення буде виглядати так:

```
public class RepairRequest {
    public Guid WorkerId { get; set; }
    public string CarName { get; set; }
    public string CarRegistrationNumber { get; set; }
    // можна додати і інші поля, наприклад:
    //   public string CarModel { get; set; }
    //   public string CarColor { get; set; }
    //   public string Comment { get; set; }
}
```

Клас RepairRequest є моделлю даних, яка використовується для передачі параметрів у методи API. Цей клас слугує як контейнер для даних, які отримує API, і зазвичай включає в себе властивості, що відповідають полям у запиті. Можна помітити, що така модель дуже схожа на модель-представлення, які ми використовували у проекті попередньої лабораторної роботи. Але в даному проекті у нас немає пред-ставлень, а подібні класи моделей прийнято називати Data Transfer Object (DTO).

Запустіть проект на виконання та скористайтесь інтерфейсом Swagger - перевірте функціонування оновленої версії метода реєстрації нового ремонту.

Далі ми розробимо просту програму-кліент для нашого API, але перед цим нам потрібно додати в MainController новий обробник GET-запиту, який дозволить отримувати детальну інформацію про виконаний ремонт разом з усіма пов’язаними даними. Це необхідно для відображення деталізованої інформації про ремонт, включаючи дані про виконавця та автомобіль, у клієнтському додатку.

Почнемо з файлу визначення контролера MainController.cs, в який потрібно додати наступний метод:

```
[HttpGet("{id}")]   // GET: /Main/{id}
public ActionResult<Document> GetById(Guid id) {
    var report = Documents.Get(id);
    if (report == null)
        return NotFound();
    return report;
}
```

У нашому додатку визначено базовий клас репозиторію, який використовується для всіх моделей. Він містить метод Get(id), що отримує сутність з БД за вказаним ідентифікатором. Однак цей метод не витягує пов'язані сутності з БД, а в об'єкті документа існують дві такі сутності: Car та Worker. Щоб вирішити цю проблему, ми створимо спеціалізовану реалізацію репозиторію для документів, а щоб не писати реалізації методів для всіх CRUD операцій ми успадкуємо наш новий клас DocumentRepository від BaseRepository<Document> та перевизначимо лише метод Get(id); код класу DocumentRepository наведено нижче:

```
using Lab6.Database;
using Lab6.Models;
using Microsoft.EntityFrameworkCore;
using System;
using System.Linq;

namespace Lab6.Repositories.Implementations {
    public class DocumentRepository : BaseRepository<Document> {
        public DocumentRepository(ApplicationContext context) : base(context) { }
        public override Document Get(Guid id) {
            return Context.Documents
                          .Include(d => d.Worker)
                          .Include(d => d.Car)
                          .FirstOrDefault(m => m.Id == id);
        }
    }
}
```

Примітка: Метод Get(id) в класі BaseRepository<Document> потрібно зробити віртуальним, щоб він міг бути перевизначений у похідних класах:

```
public virtual TDbModel Get(Guid id) {
    return Context.Set<TDbModel>().FirstOrDefault(m => m.Id == id);
}
```

На завершення, нам потрібно оновити налаштування контейнера залежностей (DI), щоб використовувати нашу нову реалізацію для IBaseRepository<Document>. Відкриваємо файл Startup.cs та у методі ConfigureServices змінюємо реєстрацію сервісу:

```
services.AddTransient<IBaseRepository<Document>, BaseRepository<Document>>();
```

на

```
services.AddTransient<IBaseRepository<Document>, DocumentRepository>();
```

1.19 Javascript клієнт

Тепер давайте розробимо клієнтську частину для нашого Web API. Головна перевага використання RESTful API полягає в їх універсальності: ви можете взаємодіяти з ними, використовуючи широкий спектр технологій та мов програмування. Це може бути веб-додаток, консольний застосунок, або навіть класичний додаток на базі Windows Forms або WPF. Для нашого прикладу ми оберемо JavaScript, який чудово підходить для створення веб-клієнтів.

Для спрощення розробки використаємо той самий веб-сервер, на якому розгорнуто наше Web API. Для початку створимо в проекті папку wwwroot – ця папка традиційно використовується для розміщення статичного контенту. Потім, у методі Configure класу Startup, додамо виклик методу розширення UseStaticFiles(). Це дозволить серверу обслуговувати статичні файли:

```
// This method gets called by the runtime.
// Use this method to configure the HTTP request pipeline.
public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
    ...
    app.UseStaticFiles();
    app.UseHttpsRedirection();
    app.UseRouting();
    ...
}
```

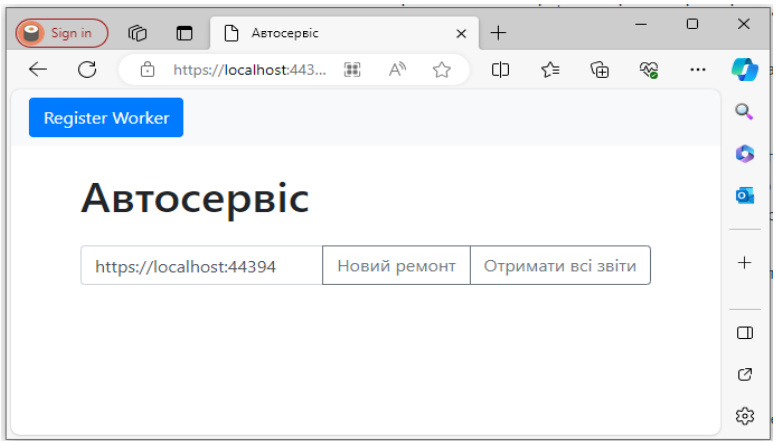
Далі, у папці wwwroot, створимо файл client.html із вмістом, наведеним у **Додатку Б**.

Я не став приводити лістінг коду html-сторінки безпосередньо тут за причини його великого об’єму. Код цього файлу як і код всього розглянутого проекту також доступен в репозіторіі за адресою:

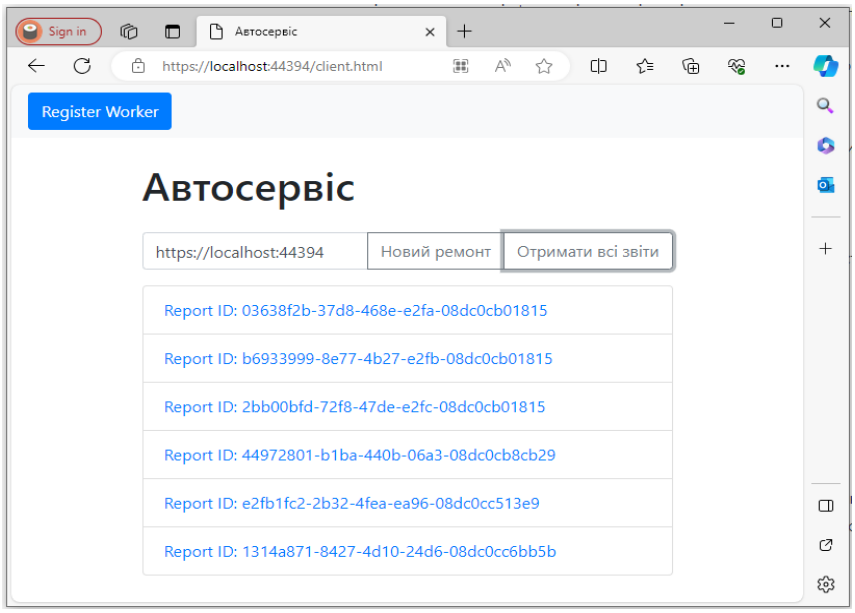
https://github.com/alysenko4317/ntu.net/tree/master/Lab_6_REST/2.machineService

Цей HTML-файл і буде нашим простим клієнтом для взаємодії з Web API. Він містить Javascript-код для виконання запитів до сервера і відображення отриманих даних.

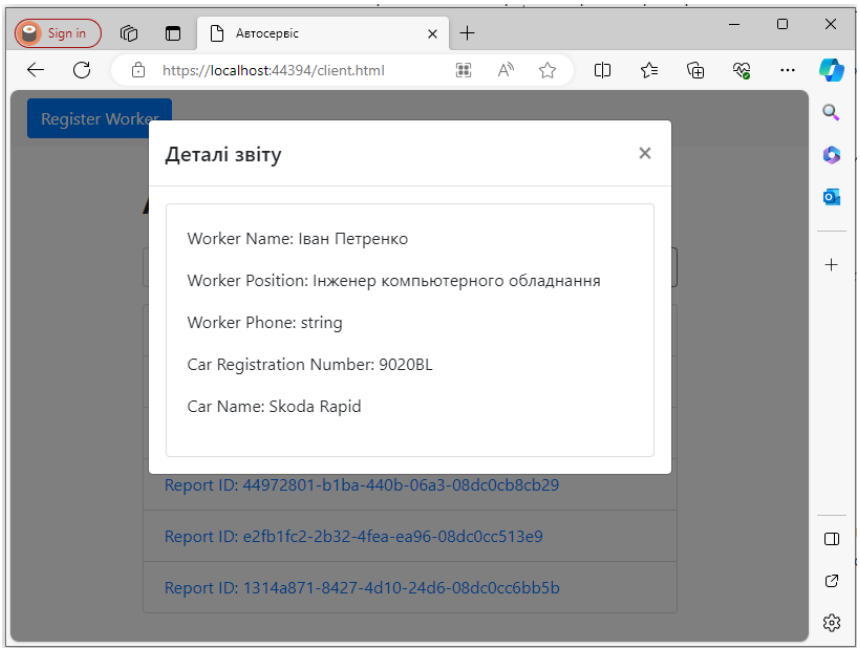
Запустіть проект на виконання, після відкриття браузера перейдіть за URL-адресою https://localhost:44394/client.html (номер tcp-порта у вас буде інший). Ви побачите наступну сторінку:



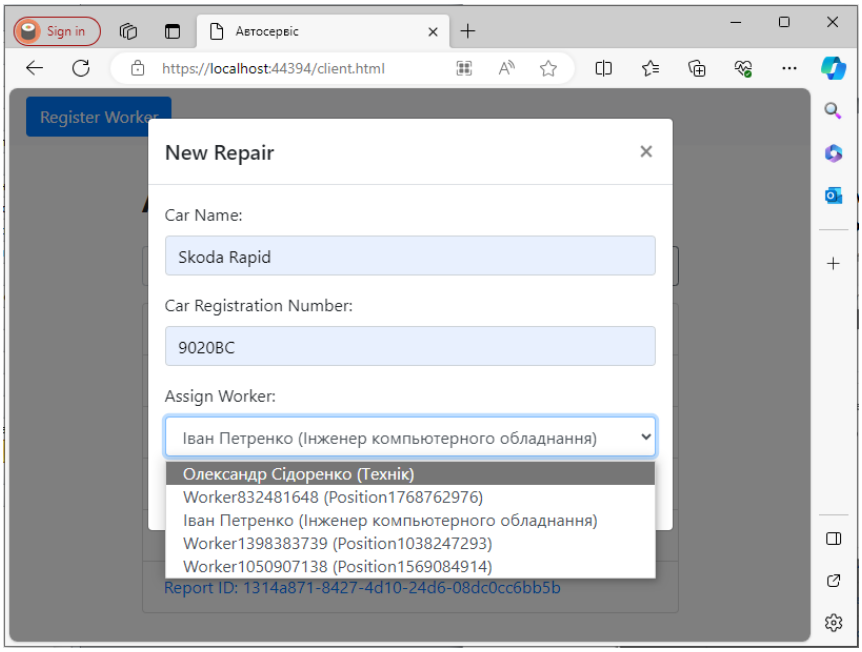
Натисніть Отримати всі звіти, через недовгу затримку ви побачите перелік всіх створених на даний час звітів, збережених у БД:



Тепер можете натиснути на будь-якому зі з'явившихся посилань, буде відображено деталі виконаного ремонту:



А натиснувши на кнопку Новий ремонт можна сформувати новий звіт



Якщо подивитись на сирцевий код цієї сторінки, можна помітити, що дані для відображення отримуються за допомогою HTTP-запитів (AJAX),, що дозволяє взаємодіяти з API без необхідності перезавантаження сторінки, а HTML розмітка сторінки оновлюється динамічно виключно на стороні клієнта.

2.1 Тестування додатку

Природний поділ різних відповідальності програми по незалежних одна від іншої частинам програмного забезпечення, яке підтримується архітектурою MVC, дозволяє будувати додатки, які порівняно легко супроводжуються і тестуються. Проте, проектувальники ASP.NET MVC на цьому не зупинилися. Для кожного фрагмента компонентно-орієнтованого проекту інфраструктури вони забезпечили структурованість, необхідну для задоволення вимог модульного тестування та засобів імітації.

У середу Visual Studio додано набір майстрів для автоматизованого створення проектів модульного тестування, які можуть бути інтегровані з інструментами модульного тестування з відкритим сирцевим кодом, такі як NUnit та xUnit. Навіть якщо вам ніколи раніше не доводилося створювати модульні тести, ви матимете все необхідне для успішного старту.

У широкому значенні сучасні розробники веб-додатків зосереджують свою увагу на двох видах автоматизованого тестування. Перший вид – це модульне тестування, що є способом перевірки поведінки окремих класів (або інших невеликих блоків коду) в ізоляції від решти додатка. Другий вид – інтеграційне тестування, який є способом перевірки поведінки безлічі компонентів, що працюють спільно, аж до цілого веб-додатку.

У веб-застосунках можуть бути корисні обидва види тестування. Модульні тести, які легко створювати та запускати, гарно підходять під час роботи з алгоритмами, бізнес-логікою та іншими частинами внутрішньої інфраструктури. Цінність інтеграційного тестування полягає в тому, що воно може моделювати взаємодію користувача з інтерфейсом і покривати весь стек технологій, що застосовуються в додатку, включаючи веб-сервер і базу даних.

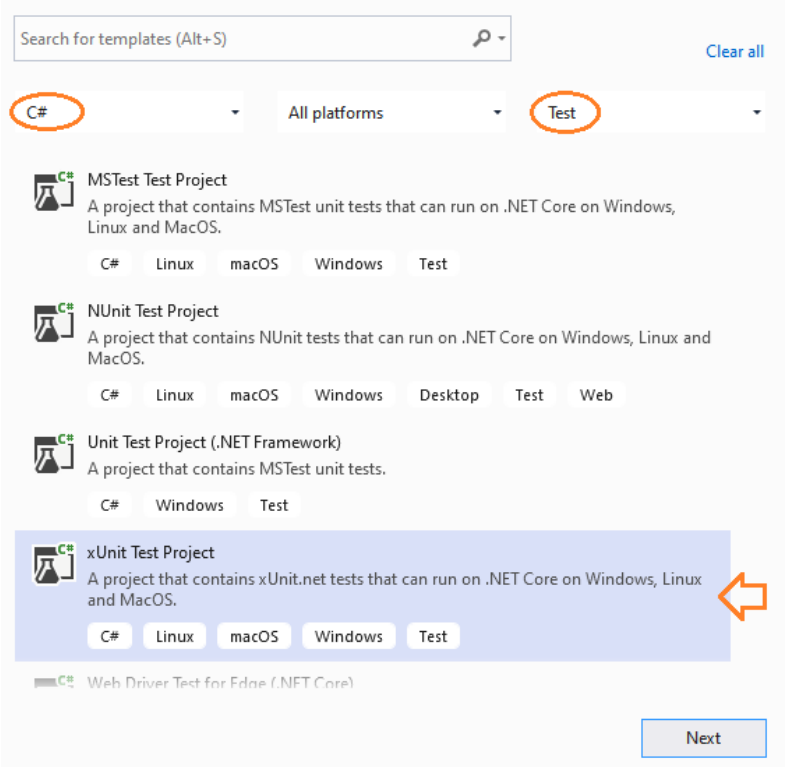
Як правило, інтеграційне тестування краще підходить для виявлення нових програмних помилок, що виникають у готових функціональних засобах; таке тестування називають регресійним тестуванням.

Ми розглянемо приклади розробки простих модульних тестів для контролерів та методів дій ASP.NET MVC, що надають фіктивні або імітовані реалізації компонентів інфраструктури для емуляції будь-якого сценарію з використанням різноманітних стратегій тестування та імітації.

2.2 Створення проекту та розробка модульних тестів

У цьому розділі показано як створити unit-тести для контролера та сервісу. Як було зазначено на початку цього посібника, для тестів слід створити окремий проєкт:

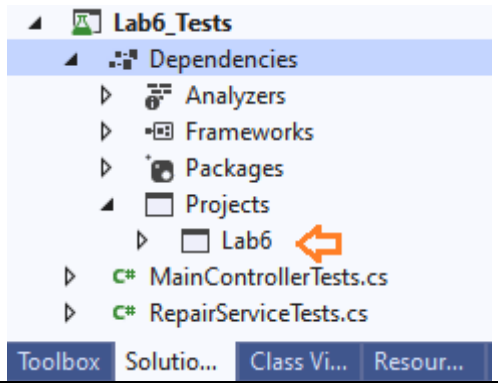
Усі нижче описані файли мають відноситься до цього нового проєкту (я назвав його Lab6_Tests).



Тести контролера слід реалізувати у файлі MainControllerTests.cs.

Насамперед підключаємо всі необхідні простори імен і додаємо залежність від основного проєкту (у мене він називається Lab6):

```
using Moq;
using Xunit;
using System;
using System.Collections.Generic;

using Lab6.Models;
using Lab6.Controllers;
using Lab6.Repositories.Interfaces;
using Lab6.Services.Interfaces;
using Microsoft.AspNetCore.Mvc;  // for
JsonResult
```



Усі тести будуть визначені в просторі імен Lab6_Tests, а тести контролера в класі MainControllerTests:

```
namespace Lab6_Tests {
    public class MainControllerTests {
    }
}
```

Почнемо з розробки тесту для методу Post, його код виглядатиме так:

```
[Fact]
public void CreateDocument()
{
    // setup test data for repair request
    var repairRequest = new RepairRequest() {
        WorkerId = Guid.NewGuid(),
        CarName = "Skoda Rapid",
        CarRegistrationNumber = "AX5060AX"
    };

    var mockDocs = new Mock<IBaseRepository<Document>>();
    var mockService = new Mock<IRepairService>();

    // setup mock service to ensure the correct parameters are passed to Work() method
    mockService.Setup(service =>
        service.Work(It.IsAny<Guid>(), It.IsAny<string>(), It.IsAny<string>()))
               .Callback<Guid, string, string>((wId, carName, carNumber) => {
                   // check actually passed values
                   Assert.Equal(wId, repairRequest.WorkerId);
                   Assert.Equal(carName, repairRequest.CarName);
                   Assert.Equal(carNumber, repairRequest.CarRegistrationNumber);
               });

    // creating the controller and invoking the Post method
    MainController controller = new MainController(mockService.Object, mockDocs.Object);
    JsonResult result = controller.Post(repairRequest) as JsonResult;
    Assert.Equal("Work was successfully done", result?.Value);
}
```

Розглянемо, як працює тест для методу Post() у класі MainController. Цей тест має на меті перевірити, як метод обробляє POST-запит і чи адекватно він додає документ до репозиторію з вказаними параметрами.

Для тестування ми використовуємо імітацію сховища – створюються mock-об'єкти для репозиторію документів (mockDocs) та сервісу ремонту (mockService). Ці mock-об'єкти дозволяють нам імітувати поведінку реальних компонентів системи без залучення реальної бази даних або інших зовнішніх залежностей.

Mock-об'єкт сервісу виконання ремонтних робіт налаштовується так, що при виклику методу IRepairService.Work() активується функція зворотного виклику (Callback), яка імітує створення нового документа з заданими параметрами. Однак, оскільки метою тесту є перевірка правильності передачі параметрів від контролера до методу Work(), то ми фокусуємося саме на перевірці, що параметри, які передаються в метод, відповідають очікуваним.

Щодо mock-об'єкту репозиторію документів, то він використовується як пустушка і потрібен лише для виконання контракту конструктора. В реальній системі об'єкт типу Document створюється та додається до репозиторію сервісом RepairService, але в нашому тесті перевіряється лише поведінка контролера, який при обробці POST-запиту не здійснює безпосередніх дій із репозиторіями.

Після налаштування mock-об'єктів, створюється екземпляр класу MainController із використанням цих mock-об'єктів. Викликається метод Post() цього контролера з об'єктом RepairRequest. Як ми вже знаємо – це так званий Data Transfer Object який містить дані про ідентифікатор працівника, назву та реєстраційний номер автомобіля. У результаті виконання методу Post() очікується, що буде повернуто повідомлення про успішне виконання роботи.

Таким чином, цей тест імітує реальну взаємодію з методом API, перевіряючи його здатність коректно обробляти та передавати наступним компонентам вхідні дані.

Далі створимо тести для різних сценаріїв методу Get класу MainController. Кожен із цих тестів демонструє важливі аспекти роботи з даними через API, переконуючись у правильності обробки запитів та відповідей.

Перший тест GetAllDocuments перевіряє, чи метод Get() повертає всі документи з репозиторію. Для цього ми налаштовуємо mock-об'єкт репозиторію так, щоб метод GetAll() повертав список, який містить один документ. Після виконання методу Get() контролера перевіряємо, чи значення, повернуте методом, відповідає очікуваному списку документів:

```
[Fact]
public void GetAllDocuments()
{
    var mockDocs = new Mock<IBaseRepository<Document>>();
    var mockService = new Mock<IRepairService>();
    var document = GetDoc();
    // setup to return the list that contains the document
    mockDocs.Setup(x => x.GetAll()).Returns(new List<Document> { document });
    MainController controller = new MainController(mockService.Object, mockDocs.Object);
    JsonResult result = controller.Get() as JsonResult;
    Assert.Equal(new List<Document> { document }, result?.Value);
}
```

Другий тест GetDocumentById перевіряє, чи метод GetById(Guid id) повертає конкретний документ за його ідентифікатором. Ми налаштовуємо mock-об'єкт так, щоб при виклику методу Get() з певним ID він повертав відповідний документ. Потім перевіряємо, що результат, який повернув метод GetById(), відповідає очікуваному документу:

```
[Fact]
public void GetDocumentById() {
    var mockDocs = new Mock<IBaseRepository<Document>>();
    var mockService = new Mock<IRepairService>();
    var document = GetDoc();
    // setup to return the document when Get is called with the ID
    mockDocs.Setup(x => x.Get(document.Id)).Returns(document);
    MainController controller = new MainController(mockService.Object, mockDocs.Object);
    JsonResult result = new JsonResult(controller.GetById(document.Id).Value);
    Assert.Equal(document, result?.Value);
}
```

Третій тест GetDocumentById_NotFound перевіряє ситуацію, коли документ за певним ID не знайдено в репозиторії. Ми налаштовуємо mock-об'єкт репозиторію так, щоб при виклику методу Get() з будь-яким ID він повертав null. Це імітує сценарій, коли документ відсутній у базі даних. У тесті ми перевіряємо, що метод GetById() в цьому випадку повертає результат типу NotFoundResult, що свідчить про відсутність документа:

```
[Fact]
public void GetDocumentById_NotFound() {
    var mockDocs = new Mock<IBaseRepository<Document>>();
    var mockService = new Mock<IRepairService>();
    // assuming no documents are found
    mockDocs.Setup(x => x.Get(It.IsAny<Guid>())).Returns((Document)null);
    MainController controller = new MainController(mockService.Object, mockDocs.Object);
    var result = controller.GetById(Guid.NewGuid()).Result as NotFoundResult;
    Assert.NotNull(result);  // ensuring that the result is a NotFoundResult
}
```

Можна помітити, що ці тести залежать від допоміжного методу GetDoc(), який створює об’єкт документу з випадково заповненими значеннями; ось його код:

```
public Document GetDoc() {
    var carId = Guid.NewGuid(); var workerId = Guid.NewGuid();
    var mockCars = new Mock<IBaseRepository<Car>>();
    var mockWorkers = new Mock<IBaseRepository<Worker>>();
    mockCars.Setup(x => x.Create(new Car() {
        Id = carId, Name = "car", Number = "123"
    }));
    mockWorkers.Setup(x => x.Create(new Worker() {
        Id = workerId,
        Name = "worker",
        Position = "manager",
        Telephone = "89165555555"
    }));
    return new Document {
        Id = Guid.NewGuid(),
        CarId = carId,
        WorkerId = workerId
    };
}
```

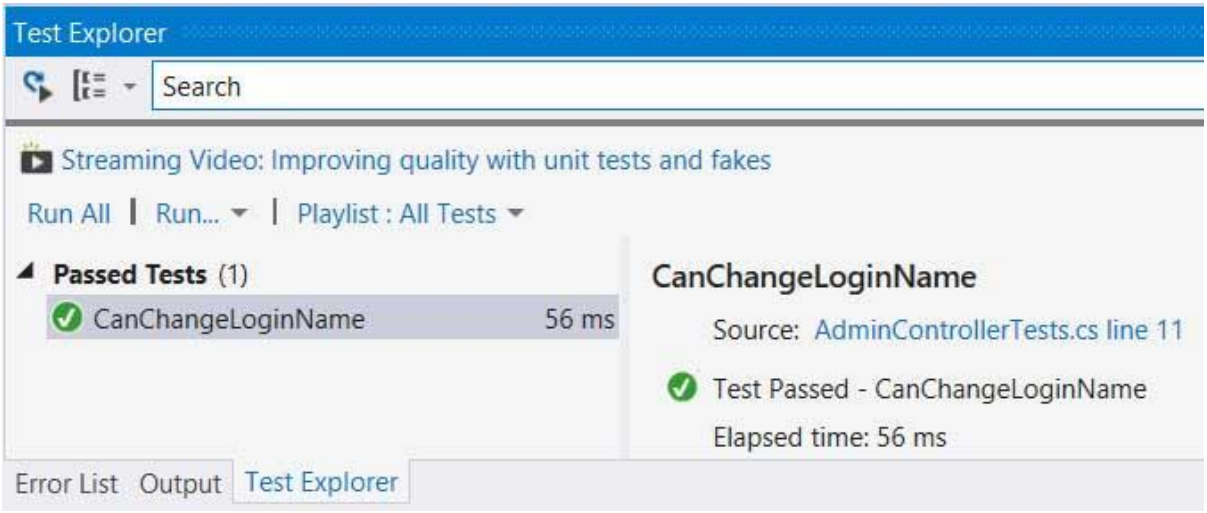
Ці три тести разом забезпечують глибоке розуміння того, як контролер MainController обробляє різні сценарії запитів на отримання даних, і демонструють важливість правильної налаштування mock-об'єктів для імітації поведінки системи.

Далі наведено код для перевірки запитів PUT та DELETE:

```
[Fact]
public void ModifyDocument()
{
    var originalDocument = GetDoc();
    var updatedDocument = GetDoc();
    updatedDocument.Id = originalDocument.Id; // assuming other properties are
updated

    var mockDocs = new Mock<IBaseRepository<Document>>();
    var mockService = new Mock<IRepairService>();

    // setup to return the original document when Get is called with the ID
    mockDocs.Setup(x => x.Get(originalDocument.Id)).Returns(originalDocument);

    // setup to capture the updated document and verify it
    mockDocs.Setup(x => x.Update(It.IsAny<Document>()))
            .Callback<Document>(d => {
                // to compare objects by values convert them to JSON
                Assert.Equal(JsonConvert.SerializeObject(updatedDocument),
                             JsonConvert.SerializeObject(d));
            })
            .Returns(updatedDocument); // return the updated document
```

```
    // creating the controller and invoking the Put method
    MainController controller = new MainController(mockService.Object,
mockDocs.Object);
    JsonResult result = controller.Put(updatedDocument);
    Assert.Equal($"Update successful {originalDocument.Id}", result?.Value);
}

[Fact]
public void DeleteDocument()
{
    var mockDocs = new Mock<IBaseRepository<Document>>();
    var mockService = new Mock<IRepairService>();
    var document = GetDoc();

    // setup to return the document when Get is called with the ID
    mockDocs.Setup(x => x.Get(document.Id)).Returns(document);

    // setup to ensure that correct document ID was passed to the Delete method
    mockDocs.Setup(x => x.Delete(It.IsAny<Guid>()))
            .Callback<Guid>(id => {
                Assert.Equal(id, document.Id);
            });

    // creating the controller and invoking the Delete method
    MainController controller = new MainController(mockService.Object,
mockDocs.Object);
    JsonResult result = controller.Delete(document.Id) as JsonResult;
    Assert.Equal("Delete successful", result?.Value);
}
```

Запуск тестів здійснюється через меню Test середовища Visual Studio. У міру виконання тестів ми отримуємо візуальний відгук, як показано на малюнку нижче:



Якщо тестове оснащення запускається без створення будь-яких необроблених винятків і всі оператори виклику методів класу Assert виконуються без проблем, вікно Test Explorer (Провідник тестів) відображає зелений маркер. В іншому випадку ми побачимо червоний маркер і подробиці про проблему.

Як бачите, застосування DI (яке ми обговорювали раніше) полегшило модульне тестування. Ми змогли створити фіктивну реалізацію сховища та впровадити її у контролер для створення специфічного сценарію. Це одна із причин популярності шаблону проектування DI.

Може здатися, що тестування простого методу зажадало неспівмірно великих зусиль, проте обсяг коду для тестування чогось значно складнішого виявився б не набагато більшим. Якщо виникає бажання відмовитися від дрібних тестів, аналогічних описаним вище, врахуйте, що такі тестові оснащення допомагають виявити помилки, які іноді залишаються непоміченими при виконанні більш складних тестів.

2.3 Фреймворк Moq

Крім власне фреймворків для створення та виконання юніт-тестів часто виникає необхідність імітувати чи емулювати певні компоненти системи, які можуть бути зовнішніми залежностями або мати складну логіку. В таких випадках використовуються фреймворки для створення mock-об'єктів.

Подібних фреймворків існує безліч, і одним із найпопулярніших є Moq. Для доступу до функціональності Moq спочатку підключається відповідний простір імен:

```
using Moq;
```

Moq дозволяє легко створювати імітації об'єктів, особливо корисних при тестуванні взаємодії між різними частинами системи. Для створення mock-об'єкта використовується синтаксис, де mock-об'єкт типізується відповідним інтерфейсом або класом:

```
var mock = new Mock<IBaseRepository<TypeName>>();
```

Після створення mock-об'єкта, можна налаштовувати його поведінку за допомогою методу `Setup()`. Цей метод дозволяє визначити, як повинен реагувати об'єкт на певні виклики методів або властивостей. Наприклад, можна налаштувати

mock-об'єкт таким чином, щоб при виклику методу Get() він повертав певний об'єкт або генерував виключення.

Окрім методу Setup(), фреймворк Moq пропонує й інші корисні інструменти, зокрема метод Verify(), який дозволяє перевірити, чи був викликаний конкретний метод з певними параметрами, що є важливим при тестуванні взаємодій між компонентами системи. Зокрема, Verify() використовується для переконання в тому, що певні методи були викликані всередині інших методів, що дозволяє гарантувати правильність взаємодій та потоку даних у програмі.

Як було продемонстровано у попередніх тестах, Moq дозволяє ефективно тестувати складні взаємодії в програмах, забезпечуючи гнучкість та контроль над тестованим середовищем.

Давайте тепер розробимо юніт-тести для другого компоненту нашого додатку – реалізації IRepairService. Ми розпочали розробку цього проекту з методу Work(), який не приймав жодних параметрів. Надалі, адаптувавши функціонал до більш реалістичних сценаріїв використання, ми розширили метод Work(), додавши передачу конкретних параметрів перед розробкою клієнта на базі JavaScript.

У фінальній реалізації сервісу я залишив обидва варіанти методу, тому давайте створимо юніт-тести для обох варіантів методу Work().

Тестовий клас RepairServiceTests виглядає так:

```
public class RepairServiceTests
{
    [Fact]
    public void WorkSuccessTest()
    {
        var (mockCars, mockWorkers, mockDocs) = SetupMocks();
        IRepairService service
            = new RepairService(mockDocs.Object, mockCars.Object, mockWorkers.Object);
        service.Work();  // test the logic
    }

    [Fact]
    public void WorkSuccessTest_withParameters()
    {
        var (mockCars, mockWorkers, mockDocs) = SetupMocks();
        // additional setup specific to this test
        mockCars.Setup(x => x.GetAll()).Returns(new List<Car> { CreateCar() });
        IRepairService service
            = new RepairService(mockDocs.Object, mockCars.Object, mockWorkers.Object);
        service.Work(CreateWorker().Id, CreateCar().Name, CreateCar().Number);
    }
   // це ще не повний код цього класу
}
```

Так як конфігурація mock-об’єктів для обох тестів є схожою, ми можемо винести цей код в окремий метод SetupMocks():

```
private (Mock<IBaseRepository<Car>>,
         Mock<IBaseRepository<Worker>>,
         Mock<IBaseRepository<Document>>) SetupMocks()
{
    var mockCars = new Mock<IBaseRepository<Car>>();
    var mockWorkers = new Mock<IBaseRepository<Worker>>();
    var mockDocs = new Mock<IBaseRepository<Document>>();

    var car = CreateCar();
    var worker = CreateWorker();
    var doc = CreateDoc(worker.Id, car.Id);

    mockCars.Setup(x => x.Get(It.IsAny<Guid>())).Returns(car);
    mockWorkers.Setup(x => x.Get(It.IsAny<Guid>())).Returns(worker);
    mockCars.Setup(x => x.Create(It.IsAny<Car>())).Returns((Car c) => c);
    mockWorkers.Setup(x => x.Create(It.IsAny<Worker>())).Returns((Worker w) => w);
    mockDocs.Setup(x => x.Create(
        It.Is<Document>(d => d.WorkerId == worker.Id && d.CarId ==
car.Id))).Returns(doc);

    return (mockCars, mockWorkers, mockDocs);
}
```

А цей метод, в свою чергу, залежить від допоміжних методів:

```
private Car CreateCar() {
    return new Car() {
        Id = Guid.NewGuid(),
        Name = "Skoda Rapid",
        Number = "AX1234AX"
    };
}

private Worker CreateWorker() {
    return new Worker() {
        Id = Guid.NewGuid(),
        Name = "worker",
        Position = "manager",
        Telephone = "89165555555"
    };
}

private Document CreateDoc(Guid workerId, Guid carId) {
    return new Document {
        Id = Guid.NewGuid(),
        CarId = carId,
        WorkerId = workerId
    };
}
```

Важливо підкреслити, що ретельно розроблені юніт-тести є ключовою складовою якісного програмного забезпечення. Через метод SetupMocks, який ми використовуємо у класі RepairServiceTests, ми забезпечуємо ефективну організацію тестового коду, уникаючи повторення та спрощуючи процес додавання нових тестів у майбутньому. Це не тільки покращує читабельність нашого тестового коду, але й сприяє легшому управлінню та модифікації тестів відповідно до змін у бізнес-логіці чи вимогах до програми.

Підготовлені нами тести для методу Work у обох його варіантах демонструють, як можна ефективно перевірити різні аспекти бізнес-логіки в рамках одного сервісу, гарантуючи надійність та стійкість реалізованих функцій. Завдяки модульному підходу до тестування та використанню mock-об'єктів, ми маємо змогу точно імітувати зовнішні залежності та зосередитись на перевірці логіки нашого сервісу, що є важливим аспектом розробки надійного програмного забезпечення.

Закінчуючи, хочу наголосити на значенні постійного розвитку та вдосконалення навичок юніт-тестування. Це не тільки підвищує якість кінцевого продукту, але й сприяє кращому розумінню власного коду та його потенційних слабких місць.

Використання TDD (розробки через тестування) та робочого потоку типу "червоний-зелений-рефакторинг" (red-green-refactor)

Під час розробки через тестування модульні тести дозволяють допомогти у проектуванні коду. Тим, хто звик проводити тестування після завершення кодування, ця концепція може бути дивною, але в цьому підході закладено величезний сенс. Ключовою концепцією є робочий потік розробки, який називається "червоний-зелений-рефакторинг" (red-green-refactor).

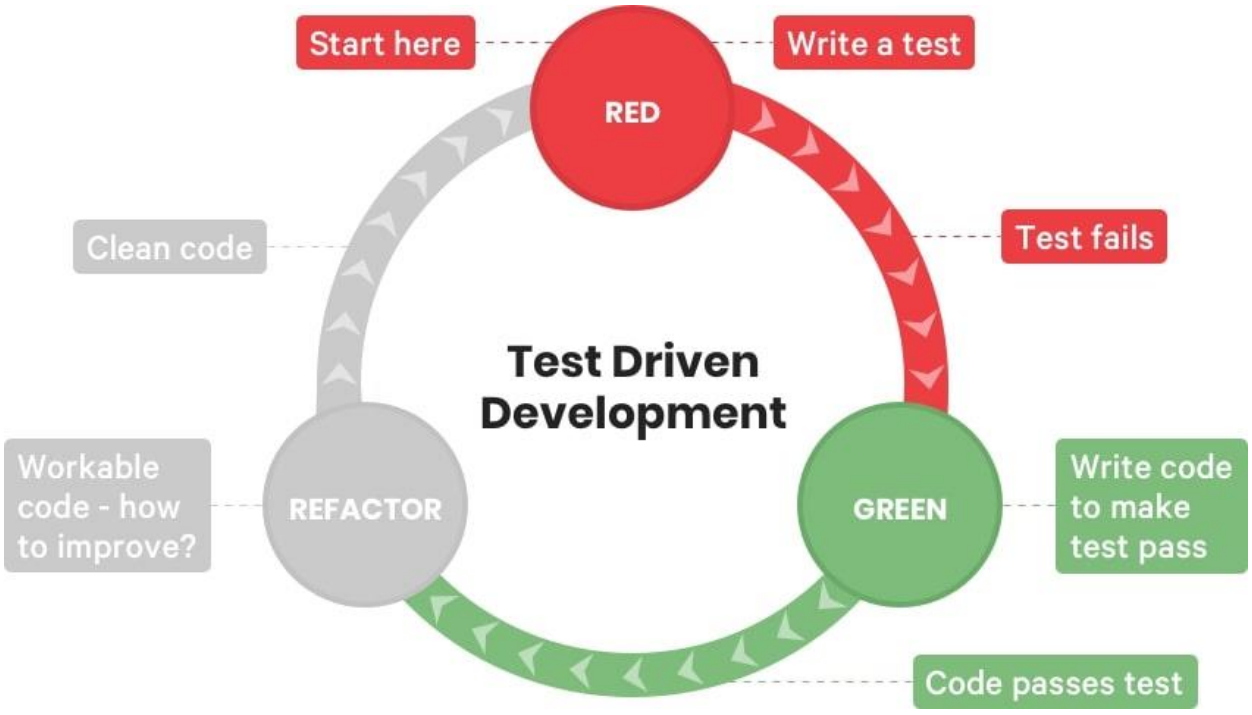
Нижче описано відповідну послідовність дій:

1. Визначте, що програма потребує додавання нової функціональної можливості або методу.
2. Створіть тест, який перевірить поведінку нової функціональної можливості після її реалізації.
3. Виконайте тест та отримайте "червоне світло".
4. Напишіть код, який реалізує нову можливість.

5. Знову запустіть тест і коригуйте код доти, доки не отримаєте "зелене світло".

6. За потреби проведіть рефакторинг коду – наприклад, реорганізуйте оператори, перейменуйте змінні тощо.

7. Виконайте тест для того, щоб переконатися, що рефакторинг не змінив поведінку програми.



Цей робочий потік повторюється для кожної функціональної можливості, що додається. У TDD традиційний процес розробки інвертується: ми починаємо з написання тестів для коректної реалізації того чи іншого функціонального засобу, знаючи, що тести не пройдуть. Потім ми реалізуємо нову функціональну можливість, створюючи всі її аспекти по черзі для проходження все більшої кількості тестів.

Такий цикл і є сенсом розробки через тестування. Є маса вагомих причин, щоб рекомендувати її як стиль розробки пріоритетного вибору, і не в останню чергу тому, що вона змушує програміста подумати про те, як повинна поводитися зміна або вдосконалення, ще до початку написання коду. У вас завжди буде чітка кінцева мета та спосіб перевірки того, що ця мета досягнута. А за наявності

модульних тестів, які покривають решту програми, можна мати впевненість у тому, що внесені зміни не модифікують якісь інші аспекти поведінки.

Спочатку розробка TDD виглядає дещо дивною, але вона має дивовижну міць, і написання тестів першими ясно встановлює припущення про те, що саме повинна робити ідеальна реалізація, перш ніж ви почнете занурюватися в технології, які використовуються для написання коду.

Недолік розробки TDD полягає в тому, що вона потребує суворої дисципліни. Коли наближаються терміни здачі робіт, завжди виникає спокуса відмовитися від TDD і просто приступити до написання коду або, як мені доводилося неодноразово бути свідком, крадькома відкинути проблемні тести, щоб зробити код виглядаючим краще оформленим, ніж він є насправді.

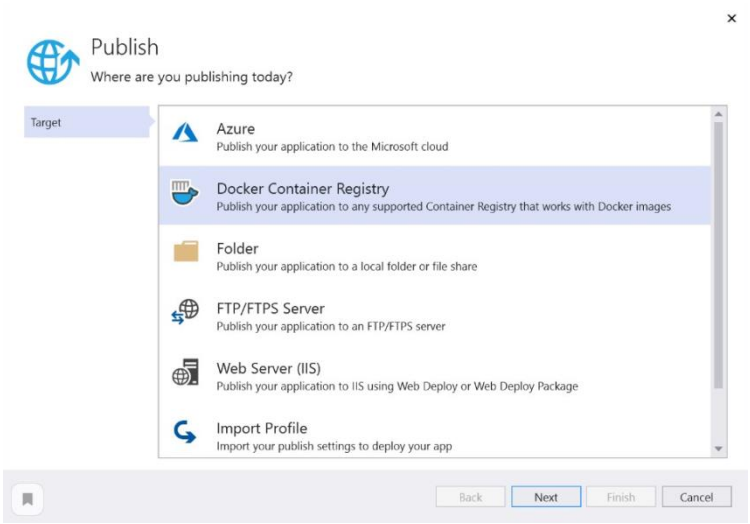
З вказаних причин розробка TDD повинна застосовуватися в згуртованих і зрілих командах розробників, де в цілому є високий рівень майстерності та дисципліни, або ж у командах, керівники яких здатні примусово змусити користуватися рекомендованими прийомами навіть в умовах обмежень у часі.

2.4 Публікація додатка в Docker

Наостанок розглянемо як викласти створений додаток у Docker Hub. У Visual Studio це зробити вкрай просто, але врахуйте, що у вас уже має бути профіль у Docker та створено репозиторій у Docker Hub.

Натискаєте ПКМ на вашому проект і вибираєте пункт Publish....

Далі вибираємо Docker Container Registry.



У наступному вікні, треба обрати Docker Hub:

Далі введіть свої облікові дані Docker.

Якщо все пройшло успішно, то залишилося зробити останню річ – натиснути кнопку Publish.

Готово, ви опублікували свій додаток у Docker Hub!

ЗАВДАННЯ ДЛЯ САМОСТІЙНОГО ВИКОНАННЯ

Розробка REST API (серверна частина)**:**

1. Створіть контролери ASP.NET, які реалізують функціональність REST API.
2. Запити та відповіді повинні містити дані у форматі JSON. Оберіть предметну область на ваш розсуд (наприклад, інформацію про погоду, курси валют, статистику користувачів, навантаження на CPU, тощо).
3. Забезпечте автоматичне оновлення значень даних через задані проміжки часу (наприклад, щосекундно).

Реалізація клієнтської частини**:**

1. Розробіть клієнтський додаток, який використовує будь-яку технологію на ваш вибір (не обов'язково JavaScript) для взаємодії з API.
2. Інтерфейс має включати функціонал для періодичного запиту та оновлення даних, отриманих від API.
3. Розгляньте використання додаткових бібліотек для поліпшення візуалізації даних (наприклад, для створення графіків або діаграм).

Додаткові рекомендації**:**

1. Переконайтеся, що API коректно обробляє запити та помилки.
2. Підготуйте коротку документацію API з описом ендпойнтів, прикладами запитів та відповідей.
3. Розгляньте можливість додавання базових юніт-тестів для вашого API.
4. Документуйте ваш клієнтський додаток, особливо щодо його взаємодії з API.

Приклади предметних областей (варіантів завдання) наведено на наступній сторинці. Номер варіанту завдання визначається за останньою цифрою залікової книжки або узгоджується з викладачем індівідуально.

1. Сервіс відстеження погодних умов: Використання датчиків для збору даних про погоду, як-от температура, вологість, тиск. REST API може обробляти дані з датчиків, а клієнт відображатиме поточні погодні умови а також історію за попередній період.

2. Система відстеження витрат на електроенергію: Розробка сервісу для відстеження споживання електроенергії у домогосподарствах. REST API зберігатиме дані з лічильників енергії, а клієнт дозволить користувачам переглядати статистику та аналізувати споживання.

3. Моніторинг рівня забруднення повітря: Сервіс для відстеження якості повітря за допомогою датчиків, які вимірюють рівні забруднюючих речовин. REST API може збирати дані з датчиків, а клієнт відображатиме ці дані для користувачів.

4. Система керування запасами продуктів харчування: Простий сервіс для відстеження запасів продуктів у домашніх умовах. Користувачі через клієнт можуть додавати, видаляти або переглядати продукти, а REST API забезпечуватиме зберігання та управління даними.

5. Система резервування кімнат у готелі: Простий інтерфейс для резервування кімнат, де клієнт дозволяє користувачам переглядати доступні кімнати та робити бронювання, а REST API веде облік бронювань та доступності кімнат.

6. Менеджер заміток: Сервіс для створення та управління персональними замітками. Користувачі можуть додавати, редагувати та видаляти замітки через графічний клієнт, а REST API забезпечує зберігання та синхронізацію даних.

7. Сервіс відстеження фітнес-активності: Програма для відстеження фізичних вправ, кроків, пробіжок тощо. Дані можуть збиратися з фітнес-пристроїв або вводитися вручну через графічний клієнт, а REST API оброблятиме та зберігатиме цю інформацію.

8. Система управління освітленням у будинку: Сервіс для керування розумними лампами або освітлювальними системами. Клієнт дозволяє користувачам керувати освітленням, а REST API взаємодіє з обладнанням.

9. Сервіс планування подорожей: Простий додаток для планування маршрутів подорожей, з можливістю збереження місць та маршрутів. REST API управляє даними маршрутів, а клієнт надає інтерфейс користувача.

КОНТРОЛЬНІ ЗАПИТАННЯ
до лабораторних робіт №5 та №6

1. Що таке ASP.NET Core та які її основні переваги порівняно з попередніми версіями ASP.NET?

2. Як функціонує паттерн MVC у контексті ASP.NET Core?

3. Які основні компоненти MVC, які їхні ролі у додатку?

4. Що таке Razor у контексті ASP.NET Core і як він використовується?

5. Як створити просту веб-сторінку з використанням Razor синтаксису?

6. Що таке маршрутизація в ASP.NET Core MVC та як вона налаштовується?

7. Як реалізувати авторизацію та аутентифікацію в ASP.NET Core?

8. Що таке Dependency Injection (DI) і як це використовується в ASP.NET Core?

9. Як працює паттерн "Repository" і чому він важливий в контексті ASP.NET Core?

10. Як використовувати Entity Framework Core для роботи з базами даних?

11. Перерахуйте принципи RESTful API і як їх імплементувати в ASP.NET Core?

12. Як створити та тестувати REST API в ASP.NET Core?

13. Що таке ViewData, ViewBag, TempData та їх використання?

14. Як налаштувати і використовувати майстер-сторінки для дизайну у ASP.NET Core?

15. Як організувати валідацію даних на стороні сервера у ASP.NET Core?

16. Що таке мідлваре (middleware) і як це використовується в ASP.NET Core?

17. Як налаштувати обробку помилок в ASP.NET Core?

18. Які основні різниці між синхронними та асинхронними методами в ASP.NET Core?

19. Що таке Tag Helpers і як їх використовувати?

20. Як працюють сесії та кукіси в ASP.NET Core?

21. Які стратегії кешування можна використовувати в ASP.NET Core?

22. Як реалізувати підтримку багатомовності в ASP.NET Core?

23. Що таке SignalR і як він використовується в ASP.NET Core?

24. Як налаштувати та використовувати логування в ASP.NET Core?

25. Які основні принципи праці з файлами та вивантаженням даних у ASP.NET Core?

26. Що таке Blazor і як його використовувати разом з ASP.NET Core?

27. Які існують способи оптимізації продуктивності ASP.NET Core додатків?

28. Як реалізувати підтримку мобільних пристроїв у ASP.NET Core?

29. Які заходи безпеки слід враховувати при розробці додатків на ASP.NET Core?

30. Що таке сервісний робітник (Service Worker) і як його використовувати в ASP.NET Core?

31. Як інтегрувати сторонні сервіси, такі як соціальні мережі, в ASP.NET Core додатки?

32. Що таке CORS і як правильно його налаштувати у ASP.NET Core?

33. Як реалізувати масштабованість та балансування навантаження в ASP.NET Core?

34. Як використовувати WebSockets в ASP.NET Core?

35. Які патерни проектування найбільш поширені та корисні в ASP.NET Core?

36. Як налаштувати та використовувати автоматизоване тестування в ASP.NET Core?

37. Як налаштувати та використовувати Continuous Integration та Continuous Deployment для ASP.NET Core проектів?

38. Які основні кроки для міграції існуючого ASP.NET додатку до ASP.NET Core?

39. Як реалізувати мікросервісну архітектуру в ASP.NET Core?

40. Що таке Docker і як його використовувати для розгортання ASP.NET Core додатків?

41. Які типи веб-застосунків можна створювати у MS Visual Studio, чим вони відрізняються?

42. Які види html-запитів використовувалися у додатку, чим вони відрізняються?

РЕКОМЕНДОВАНІ ДЖЕРЕЛА

1. Andrew Troelsen, Philip Japikse. Pro C# 7: With .NET and .NET Core, 8th edition – Apress, 2017. - 2077 p.

2. Jeffrey Richter. CLR via C# (Developer Reference) 4th edition – Microsoft Press, 2012. - 896 p.

3. James Chambers, David Paquette, Simon Timms. ASP.NET Core Application Development: Building an application in four sprints (Developer Reference), 1st edition – Microsoft Press, 2016. – 432 с.

4. Ian Miell, Aidan Hobson Sayers. Docker in Practice, 2nd Edition – Manning, 2019. - 384 p.

5. Adrian Mouat. Using Docker: Developing and Deploying Software with Containers, 1st edition – O'Reilly Media, 2016. - 354 p.

6. Joseph Albahari. C# 9.0 in a Nutshell: The Definitive Reference. — O'Reilly Media, 2021. – 1058 с.

7. Mark J. Price. C# 8.0 and .NET Core 3.0 – Modern Cross-Platform Development: Build applications with C#, .NET Core, Entity Framework Core, ASP.NET Core, and ML.NET using Visual Studio Code, 4th edition – Packt Publishing, 2019. - 818 p.

8. Joseph Rattz, Adam Freeman. Pro LINQ: Language Integrated Query in C# 2010 (Expert's Voice in .NET), 1st edition – Apress, 2010. - 862 p.

9. Itzik Ben-Gan. T-SQL Fundamentals (Developer Reference) 3rd edition – Microsoft Press, 2016. - 464 p.

10. Karl Seguin, Perry Neal. The Little Redis Book – Self Published, 2012. - 32 p.

11. Методичні вказівки до лабораторних робіт «Основи паралельного програмування мовою C#» з дисципліни «Технології програмування» / Ю. Н. Кожин, О. Н. Малих, В. Ф. Прокопенко. – Харків: НТУ «ХПІ», 2018. – 64 с.

12. Автоматизоване тестування веб-додатків з різнорівневою архітектурою // Г.М. Кодола, Н.С. Волинець, І.В. Сербулова. – Вісник НТУ "ХПІ" №5, 2019

13. Esposito, Dino. Programming Microsoft® ASP.NET 4, 2-е изд. — Redmond, Washington 98052-6399: Microsoft Press, 2011. — 966 с.

14. Jeffrey Palermo, Jimmy Bogard, Eric Hexter, Matthew Hinze, and Jeremy Skinner. ASP.NET MVC 4 in Action. – M.: Manning, 2012. – 408 с.

15. Adam Freeman, Matthew MacDonald, Mario Szpuszta. Pro ASP.NET 4.5 in C#, Apress, 2013. — 1620 с.

ДОДАТОК А

Приклад оформлення звіту з лабораторної роботи

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«Харківський політехнічний інститут»
Кафедра стратегічного управління

ЗВІТ

з лабораторної роботи №5
по курсу «Стек технології»

ВИКОНАВ(ЛА):
Ст. гр. _____
_____
ПІБ

ПЕРЕВІРИВ(ЛА):
доцент каф. СУ
_____
ПІБ

Харків 2023

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Автосервіс</title>
       <style>
          .modal-lg { -width: 80%; /* adjust the width of large modals */ }
          .modal-body .row { padding: 5px 0; /* spacing within each row */ }
          .font-weight-bold { color: #0056b3; /* custom color for the key labels */ }
       </style>
    <link href=https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css
          rel="stylesheet">
    <script src="https://code.jquery.com/jquery-3.5.1.min.js"></script>
    <script src="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/js/bootstrap.min.js"></script>
    <script>
    $(document).ready(function() {

    // set default value for the server IP and port textbox
    var defaultUrl = window.location.protocol + '//' + window.location.hostname;
    if (window.location.port)
        defaultUrl += ':' + window.location.port;
    $('#urlInput').val(defaultUrl); // do not append '/Main'

    $('#newRepair').click(function() {   // Open New Repair modal dialog
        // fetch workers to populate the dropdown
        $.get($('#urlInput').val() + '/api/Workers', function(workers) {
            var workerSelect = $('#workerSelect');
            workerSelect.empty();
            workers.forEach(function(worker) {
                workerSelect.append(
                   new Option(worker.name + ' (' + worker.position + ')', worker.id));
            });
        });

        $('#newRepairModal').modal('show');  // show the dialog
    });

    $('#newRepairForm').on('submit', function(e) {   // Handle New Repair form submission
        e.preventDefault();

        var repairData = JSON.stringify({
            carRegistrationNumber: $('#carRegistrationNumber').val(),
            carName: $('#carName').val(),
            workerId: $('#workerSelect').val()
        });

        $.ajax({
            url: $('#urlInput').val() + '/Main',   // adjust URL if necessary
            type: 'POST',
            contentType: 'application/json',
            data: repairData,
            success: function(response) {   // handle the response
                alert("New repair registered successfully!");
                $('#newRepairModal').modal('hide');
                $('#newRepairForm').trigger("reset");
            },
            error: function() {
                alert("Error registering new repair.");
            }
        });    });
```

```
$('#getAllReports').click(function() {
    $.get($('#urlInput').val() + '/Main', function(data, status) {
        let reportsList = $('#reportsList');
        reportsList.empty();
        data.forEach(function(report) {
            reportsList.append(
                '<li class="list-group-item">' +
                '<a href="#" data-toggle="modal"' +
                'data-target="#reportModal" data-report-id="' + report.id + '">' +
                'Report ID: ' + report.id +
                '</a>' + '</li>'
            );
        });
    });
});

$('#reportModal').on('show.bs.modal', function(event) {
    var button = $(event.relatedTarget);
    var report = button.data('report');
    var modal = $(this);
    var modalBody = modal.find('.modal-body');
    modalBody.empty();
    var contentHtml = '<div class="card"><div class="card-body">';
    for (var key in report) {
        if (report.hasOwnProperty(key)) {
            contentHtml += '<div class="row mb-3">' +
                           '<div class="col-sm-4 font-weight-bold">' + key + ':</div>' +
                           '<div class="col-sm-8">' + report[key] + '</div>' +
                           '</div>';
        }
    }
    contentHtml += '</div></div>';
    modalBody.html(contentHtml);
});

$('#reportsList').on('click', 'a', function(e) {
    e.preventDefault();
    var reportId = $(this).data('report-id');
    // fetch the detailed report data from the API
    $.get($('#urlInput').val() + '/Main/' + reportId, function(detailedReport) {
        var modal = $('#reportModal');
        var modalBody = modal.find('.modal-body');
        modalBody.empty();
        // construct the modal content with detailed report data
        var contentHtml = '<div class="card"><div class="card-body">';
        contentHtml += '<p>Worker Name: ' + detailedReport.worker.name + '</p>';
        contentHtml += '<p>Worker Position: ' + detailedReport.worker.position + '</p>';
        contentHtml += '<p>Worker Phone: ' + detailedReport.worker.telephone + '</p>';
        contentHtml += '<p>Car Registration Number: ' + detailedReport.car.number + '</p>';
        contentHtml += '<p>Car Name: ' + detailedReport.car.name + '</p>';
        // ... add more details as needed ...
        contentHtml += '</div></div>';
        modalBody.html(contentHtml);
        modal.modal('show');
    });
});

// Toggle Worker Registration Form
$('#toggleWorkerForm').click(function() {
    $('#workerRegistrationForm').toggle();
});
```

```
    // Handle the worker registration form submission
    $('#registerWorkerForm').on('submit', function(e) {
        e.preventDefault();

        var workerData = JSON.stringify({
            name: $('#workerName').val(),
            position: $('#workerPosition').val(),
            telephone: $('#workerTelephone').val()
        });

        $.ajax({
            url: $('#urlInput').val() + '/api/Workers',
            type: 'POST',
            contentType: 'application/json',
            data: workerData,
            success: function(response) {
                // Handle the response
                alert("Worker registered successfully!");
                // Close the modal and clear the form
                $('#registerWorkerModal').modal('hide');
                $('#registerWorkerForm').trigger("reset");
            },
            error: function() {
                alert("Error registering worker.");
            }
        });
    });

    }); </script>
</head>
<body>

<!-- Navbar Admin Menu -->
<nav class="navbar navbar-expand-lg navbar-light bg-light">
    <button class="btn btn-primary" data-toggle="modal" data-target="#registerWorkerModal">
        Register Worker
    </button>
</nav>

<!-- Modal for Report Details -->
<div class="modal fade" id="reportModal" tabindex="-1" aria-labelledby="reportModalLabel"
     aria-hidden="true">
    <div class="modal-dialog">
        <div class="modal-content">
            <div class="modal-header">
                <h5 class="modal-title" id="reportModalLabel">Деталі звіту</h5>
                <button type="button" class="close" data-dismiss="modal" aria-label="Close">
                    <span aria-hidden="true">&times;</span>
                </button>
            </div>
            <div class="modal-body">
                <!-- Report details will be added here -->
            </div>
        </div>
    </div>
</div>
```

(продовження)

```
<!-- Modal for Registering a Worker -->
<div class="modal fade" id="registerWorkerModal" tabindex="-1" role="dialog" aria-
labelledby="registerWorkerModalLabel" aria-hidden="true">
    <div class="modal-dialog" role="document">
        <div class="modal-content">
            <div class="modal-header">
                <h5 class="modal-title" id="registerWorkerModalLabel">Register New Worker</h5>
                <button type="button" class="close" data-dismiss="modal" aria-label="Close">
                    <span aria-hidden="true">&times;</span></button>
            </div>
            <div class="modal-body">
                <form id="registerWorkerForm">
                    <!-- Form fields for worker registration -->
                    <div class="form-group">
                        <label for="workerName">Name:</label>
                        <input type="text" class="form-control" id="workerName" required>
                    </div>
                    <div class="form-group">
                        <label for="workerPosition">Position:</label>
                        <input type="text" class="form-control" id="workerPosition" required>
                    </div>
                    <div class="form-group">
                        <label for="workerTelephone">Telephone:</label>
                        <input type="text" class="form-control" id="workerTelephone" required>
                    </div>
                    <button type="submit" class="btn btn-primary">Register Worker</button>
                </form>
            </div>
        </div>
    </div>
</div>

<!-- New Repair Modal -->
<div class="modal fade" id="newRepairModal" tabindex="-1" role="dialog" aria-
labelledby="newRepairModalLabel" aria-hidden="true">
    <div class="modal-dialog" role="document">
        <div class="modal-content">
            <div class="modal-header">
                <h5 class="modal-title" id="newRepairModalLabel">New Repair</h5>
                <button type="button" class="close" data-dismiss="modal" aria-label="Close">
                    <span aria-hidden="true">&times;</span></button>
            </div>
            <div class="modal-body">
                <form id="newRepairForm">
                    <div class="form-group">
                        <label for="carName">Car Name:</label>
                        <input type="text" class="form-control" id="carName" required>
                    </div>
                    <div class="form-group">
                        <label for="carRegistrationNumber">Car Registration Number:</label>
                        <input type="text" class="form-control" id="carRegistrationNumber" required>
                    </div>
                    <div class="form-group">
                        <label for="workerSelect">Assign Worker:</label>
                        <select class="form-control" id="workerSelect" required></select>
                    </div>
                    <button type="submit" class="btn btn-primary">Submit</button>
                </form>
            </div>
        </div>
    </div></div>
```

(продовження)

```
<div class="container mt-4">
    <h1 class="mb-4">Автосервіс</h1>
    <div class="input-group mb-3">
        <input type="text" class="form-control"
               placeholder="Enter server IP and TCP-port (e.g., http://localhost:5000)" id="urlInput">
        <div class="input-group-append">
            <button class="btn btn-outline-secondary" type="button" id="newRepair">
                Новий ремонт</button>
            <button class="btn btn-outline-secondary" type="button" id="getAllReports">
                Отримати всі звіти</button>
        </div>
    </div>
    <ul class="list-group" id="reportsList">
        <!-- Reports will be added here dynamically -->
    </ul>
</div>

</body>
</html>
```

Навчальне видання

Методичні вказівки
для лабораторних робіт
«Стек технології»
з дисципліни «Стек технології»
ЧАСТИНА ІІ

для студентів
122 спеціальності – комп'ютерні науки

Укладачі:
ЛИСЕНКО Антон Олександрович
ЛИСЕНКО Олександр Олександрович

Відповідальний за випуск (завідувач кафедри) Марина ГРИНЧЕНКО
Роботу рекомендував до друку (експерт РВР) Ігор ГАМАЮН
Комп'ютерна верстка _____
В авторській редакції

План 2023 р., поз. __

Підп. до друку (дата підпису проректора)_____.
Гарнітура Times New Roman.

Видавничий центр НТУ «ХПІ».
Свідоцтво про державну реєстрацію ДК № 5478 від 21.08.2017 р.
61002, Харків, вул. Кирпичова, 2