ОЛЕГ

1.Вступ

RESTful сервіси є одним з ключових концептів сучасних веб-розробок, відкриваючи шлях до побудови динамічних та ефективних веб-додатків. REST (Representational State Transfer) визначає архітектурний стиль для створення розподілених систем, що базується на принципах спрощення взаємодії між клієнтами та серверами. Цей підхід покладається на використання стандартних протоколів, таких як HTTP, і забезпечує однорівневий доступ до ресурсів за допомогою уніфікованих ідентифікаторів ресурсів (URI).

У цій доповіді ми розглянемо основні концепції та практики реалізації RESTful сервісів з використанням фреймворку ASP .net core.

Крім теоретичного викладу, ми також продемонструємо практичний приклад реалізації RESTful сервісу.

Що таке REST?

REST (з англійської Representational State Transfer, «передача репрезентативного стану») — підхід до архітектури мережевих протоколів, які надають доступ до інформаційних ресурсів. Був описаний і популяризований 2000 року Роєм Філдінгом, одним із творців протоколу HTTP та був розроблений паралельно і у зв'язці з HTTP 1.1.

Дані повинні передаватися у вигляді невеликої кількості стандартних форматів (наприклад, HTML, XML, JSON). Будь-який REST протокол (HTTP в тому числі) повинен підтримувати кешування, не повинен залежати від мережевого шару, не повинен зберігати інформацію про стан між парами «запит-відповідь». Такий підхід забезпечує масштабованість системи і дозволяє їй еволюціонувати щоб відповідати новим вимогам. Однак це породжує свої недоліки - при підході REST кількість методів і складність протоколу суворо обмежені, що призводить до суттєвого збільшення кількості окремих ресурсів.

Ось стислий перелік основних особливостей архітектури REST:

- Клієнт-серверна архітектура
- Відсутність стану
- Кешування
- Однозначна ідентифікація ресурсів
- Маніпуляція ресурсами через представлення
- Поділ на шари абстракції

Відповідно RESTful сервіс - це додаток, який відповідає переліченим обмеженням архітектури REST.

MVC

На практиці для реалізації RESTful сервісів, особливо серверної частини, зазвичай використовують шаблон MVC - Model, View, Controller. Він передбачає поділ застосунку на 3 відповідні шари.

Model репрезентує дані та бізнес-логіку застосунку, відповідає на запити на отримання даних від Controller

View відображає дані з Model користувачу та надсилає користувацькі запити до controller

Controller виступає посередником між іншими двома компонентами передає запити від View до Model та повертає дані від Model до View.

Таким чином цей шаблон дозволяє ізолювати шари застосунку, та унеможливити пряму взаємодію користувача з даними, що полегшує тестування та розробку: база даних спілкується тільки з Model, Model та View - тільки з Controller.

Наведені особливості MVC створюють чимало переваг у розробці додатку:

- Розділення обов'язків - MVC розділяє різні аспекти додатка (дані, інтерфейс користувача та логіку), що робить код більш зрозумілим, зручним для підтримки та модифікації.
- Модульність - Кожен компонент (Модель, Вид, Контролер) може бути розроблений та протестований окремо, що сприяє перевикористанню коду та масштабованості.
- Гнучкість - Оскільки компоненти є незалежними, зміни в одному компоненті не впливають на інші, що дозволяє легше вносити оновлення та модифікації.
- Паралельний розвиток - Кілька розробників можуть працювати над різними компонентами одночасно, що прискорює процес розробки.
- Перевикористання коду - Компоненти можна перевикористовувати в інших частинах додатка або в різних проектах, що скорочує час і зусилля, витрачені на розробку.

Але також мають певні недоліки:

- Складність - Реалізація патерну MVC може ускладнити код, особливо для простіших додатків, що може призвести до додаткових витрат у часі розробки.
- Додаткові витрати апаратних ресурсів - Взаємодія між компонентами (Модель, Вид, Контролер) може призвести до додаткових витрат, що впливає на продуктивність додатка, особливо в умовах обмежених ресурсів.
- Потенціал для оверінжинірингу: У деяких випадках розробники можуть занадто ускладнити додаток, додаючи непотрібні абстракції та шари, що призводить до розпухлого та важкого для підтримки коду.
- Збільшена кількість файлів: MVC може призвести до більшої кількості файлів та класів порівняно з простішими архітектурами, що може зробити структуру проекту складнішою та важчою для розуміння.

ASP.NET

ASP.NET - фреймворк для створення веб-застосунків на платформі .NET. Він був розроблений компанією Microsoft із випуском першої стабільної версії у 2002 році. Він розроблявся разом із платформою .NET як заміна застарілому фреймворку ASP. Остання версія - мультиплатформенний ASP.NET Core була випущена у 2016 році. Вона є суттєво переробленою версією, яка об'єднує до цього окремі модулі ASP.NET MVC, ASP.NET Web API. Саме цю версію ми будемо розглядати сьогодні.

СЕРГІЙ

2. Створення RESTful-сервісів з ASP.NET Core

Створення нового ASP.NET Core Web API проекту

ASP.NET Core — це сучасний фреймворк мови C#, який підтримує кросплатформенність та високу продуктивність, що робить його ідеальним вибором для різноманітних створення веб-сервісів. Створення буд-якої програми починається із створення нового проєкту .net

Якщо ви використовуєте Visual Studio, то можете скористатися вбудованим шаблоном для швидкого створення структури проєкту:

- У меню «File» виберіть «Create» > «Project».
- Введіть Web API у вікні пошуку.

- Виберіть шаблон ASP.NET Core Web API і виберіть Далі.
- У діалоговому вікні «Configure your new project dialog» введіть назву проєкта та виберіть «Next».
- У діалоговому вікні Additional information:

    - Переконайтеся, що платформа .NET 8.0 (long-term support).
    - Переконайтеся, що встановлено прапорець Use controllers (зніміть прапорець, щоб використовувати мінімальний API).
    - Переконайтеся, що встановлено прапорець «Enable OpenAPI support».
    - Виберіть Create.

Часто створення RESTful сервіса передбачає використання сторонніх бібілотек, наприклад. Для підключення бази даних, тому вам може бути потрібно додати необхідний пакет NuGet. NuGet – це пакетний менеджер для бібліотек .NET

- У меню «Tools» виберіть «NuGet Packet Manager» > «Manage NuGet packages for solution».
- Виберіть вкладку Browse.
- Введіть назву потрібного пакета у поле пошуку щоб знайти його
- Установіть прапорець Project на правій панелі, а потім виберіть Install.

Для інших IDE, таких як VS code, потрібно буде використовувати CLI утиліту:

- Відкрийте термінал та перейдіть до директорії проєкту
- Виконайте наступні команди:

    dotnet new webapi --use-controllers -o “YourProjectName”

    cd “YourProjectName”

Для встановлення NuGet пакетів:

    dotnet add package “PackageName”

В результаті буде створено шаблонний Web API проєкт для прогнозу погоди.

Після створення проекту ви побачите стандартну структуру проекту ASP.NET Core, яка містить каталоги "Controllers", "Models", "Views" та інші. Контролери в каталозі "Controllers" відповідають за обробку HTTP-запитів та відправку відповідей. Моделі в каталозі "Models" представляють дані, з якими працює ваше API.

Для запуску проєкту використовуйте зелену кнопку "Start" у Visual Studio або виконайте команду dotnet run з командного рядка у каталозі вашого проекту. Після цього створене API буде доступне за вказаною URL-адресою, наприклад, https://localhost:5001/api/weatherforecast

Це лише початок у створенні веб-сервісу з використанням ASP.NET Core Web API. Ви можете додавати нові контролери, розширювати функціонал та налаштовувати ваше API відповідно до ваших потреб.

Струкура проєкту

У ASP.NET Core використовується патерн реалізації MVC (Model-View-Controller) для створення RESTful сервісів. Давайте розглянемо кожен його компонент і його реалізацію:

1. Модель (Model): Модель представляє дані додатку і зазвичай міститься у класах. У ASP.NET Core моделі можуть бути представлені класами C#, які описують потрібну структуру даних. Наприклад, модель користувача може виглядати так:

2. Контролер (Controller): Контролер в ASP.NET Core відповідає за обробку HTTP-запитів та взаємодію з моделями. Контролери у ASP.NET Core це класи, які успадковують *ControllerBase* та містять методи для обробки різних типів запитів. Маршрутизація налаштовується за допомогою атрибутів, які вказують, який URL повинен відповідати кожному методу контролера. Це виглядає наступним чином:

3. Представлення (View): У RESTful сервісах представлення використовується для передачі даних між клієнтом та сервером у вигляді JSON. У ASP.NET Core можна повертати дані у вигляді об'єктів C# або серіалізувати дані у JSON форматі, що є більш поширеною практикою.

==========

Фреймворк ASP .NET оптимізований для використання разом з Microsoft Visual Studio Code, тому додавати основні компоненти RESTful додатку тут можна використовуючи готові шаблони, які мінімізують написання стандартного коду.

Створення моделей

- У Solution Explorer клацніть проект правою кнопкою миші. Виберіть Add > New Folder. Назвіть папку Models.
- Клацніть правою кнопкою миші папку «Models» та виберіть «Add» > «Class». Оберіть назву класу і натисніть «Add».
- Замініть код шаблону там, де необхідно.

Створення контроллерів

- Клацніть правою кнопкою миші папку Controllers.
- Виберіть «Add» > «New Scaffolded Item».
- Виберіть API controller with actions за допомогою Entity Framework, а потім виберіть Add.
- У діалоговому вікні «Додати контролер API з діями» за допомогою Entity Framework:

    - Виберіть необхідну модель (“ProjectName”.Models) у класі Model.
    - Виберіть Context (“ProjectName”.Models) у Data Context Class.
    - Виберіть Додати.

У згенерованому коді:

- Класс позначається атрибутом [ApiController]. Цей атрибут вказує, що контролер відповідає на запити веб-API.

- Використовується Dependency Injection для введення контексту бази даних (TodoContext) у контролер. Контекст бази даних використовується в кожному з методів CRUD у контролері.

ASP.net Core підтримує два типи шаблонів для контролерів. Шаблони ASP.NET Core для контролерів з представленнями включають [action] у шаблон маршруту, а шаблони для контролерів API - ні.

Якщо маркера [action] немає в шаблоні маршруту, ім’я дії (ім’я методу) не включається в кінцеву точку. Тобто пов’язане з дією ім’я методу не використовується у відповідному маршруті.

Робота з HTTP-методами (GET, POST, PUT, DELETE)

Згідно зі стандартами RESTful .net ASP для маніпуляцій даними використовує наступні HTTP запити:

POST - метод POST найчастіше використовується для **створення** нових ресурсів. Зокрема, він використовується для створення ресурсів, підпорядкованих якомусь іншому батьківському ресурсу. Після успішного створення повертає статус HTTP 201, повертаючи заголовок Location із посиланням на щойно створений ресурс із статусом HTTP 201.

GET - Метод HTTP GET використовується для **читання** (або отримання) представлення ресурсу. За відсутності помилок GET повертає представлення в XML або JSON і код відповіді HTTP 200 (OK). У разі помилки він найчастіше повертає 404 (НЕ ЗНАЙДЕНО) або 400 (НЕВДАЛИЙ ЗАПИТ).

Відповідно до специфікації HTTP запити GET (разом із HEAD) використовуються лише для читання даних, а не для їх зміни. Тому при такому застосуванні вони вважаються безпечними. Тобто їх можна викликати без ризику модифікації чи пошкодження даних — один виклик матиме той самий ефект кожен раз, або не матиме жодного ефекту.

Не виконуйте небезпечні операції через GET — він ніколи не повинен змінювати будь-які ресурси на сервері.

PUT - найчастіше використовується для можливостей **оновлення** відомого URI ресурсу та передає оновлену інформацію ресурсу у тілі запиту.

Однак PUT також можна використовувати для створення ресурсу у випадку, коли ідентифікатор ресурсу вибирає клієнт, а не сервер, або якщо PUT робить запит до URI, який містить ідентифікатор неіснуючого ресурсу. Це може ускладнити та заплутати роботу системи, тому цю можливість варто використовувати обережно, або взагалі уникати її.

Після успішного оновлення ресурсу PUT повертає статус 200 (або 204, якщо не повертається вміст у тілі). Тіло у відповіді є необов’язковим. Немає необхідності повертати клієнту дані, які він сам щойно надіслав.

PUT не є безпечною операцією, оскільки вона змінює стан на сервері, але вона є ідемпотентною. Тобто кожне надсилання одного і того ж запиту матиме однаковий результат.

Але якщо, наприклад, виклик PUT збільшує лічильник у ресурсі він перестає бути ідемпотентим. Таких випадків варто уникати.

PATCH - використовується для можливостей **модифікації**. Запит PATCH має містити лише зміни до ресурсу, а не весь ресурс.

Це схоже на функціонал PUT, але тут тіло запиту містить набір інструкцій, що описують, як потрібно змінити ресурс, який зараз знаходиться на сервері, а не просто нову версію ресурсу, як у PUT.

PATCH не є ні безпечним, ні ідемпотентним. Однак він може бути поданий таким чином, щоб бути ідемпотентним, що також допомагає запобігти поганим результатам через одночасне надсилання двох запитів PATCH на один ресурс. Колізії від кількох запитів PATCH можуть бути більш небезпечними, ніж колізії PUT, можуть пошкодити дані та мати побічні ефекти на інші ресурси. Тому перед модифікацією даних потрібно перевіряти, чи не було їх змінено з моменту останнього запиту клієнта.

DELETE - використовується для **видалення** ресурсу, визначеного URI.

У разі успішного видалення повертає HTTP-статус 200 (OK) разом із тілом відповіді, можливо, представленням видаленого елемента. Або повертає статус HTTP 204 (БЕЗ ВМІСТУ) без тіла відповіді.

З точки зору специфікації HTTP, операції DELETE є ідемпотентними. Якщо ви ВИДАЛИТЕ ресурс, він зникне. Повторний виклик DELETE для цього ресурсу закінчується тим самим: ресурс зник.

У ASP описані HTTP запити вказуються при створенні методів контролера, таким чином кожен контролер може описувати різні операції, які можна виконувати над ресурсом, і кожен метод представляє одну з них.

Повернення даних в JSON форматі

Згідно із специфікацією RESTful додатків, кожен запит, який повертає дані, повертає їх у форматі JSON.

JSON (JavaScript Object Notation) це - простий формат обміну даними, що є зручним як для читання та написання людиною, так і для парсінгу та генерації комп'ютером. Він базується на підмножині мови програмування JavaScript стандарту ECMA-262 3rd Edition від грудня 1999 року. JSON - це текстовий формат, повністю незалежний від мови реалізації, але він використовує конвенції, знайомі програмистам С-подібних мов. Ці властивості роблять JSON ідеальною мовою для обміну даними.

JSON базується на двох структурах даних:

- Колекція пар ключ/значення. У різних мовах ця концепція реалізована як *об'єкт*, запис, структура, словник, хеш, іменований список або асоціативний масив.
- Упорядкований список значень. У більшості мов це реалізовано як *масив*, вектор, список або послідовність.

Це універсальні структури даних. У тому чи іншому вигляді їх підтримують майже усі сучасні мови програмування, у тому числі C#. Є сенс будувати формат даних, що є незалежним від мови програмування, саме на цих структурах.

У нотації JSON це виглядає так:

Об'єкт - це невпорядкований набір пар ключ/значення. Об'єкт починается з {відкриваючої фігурної дужки і закінчується }закриваючою фігурною дужкою. Кожне ім'я супроводжують :двокрапкою, а пари ключ/значення відокремлюють ,комою.

Масив - це впорядкована колекція значень. Масив починається з [відкриваючої квадратної дужки і закінчується]закриваючою квадратною дужкою. Значення відокремлюють ,комою.

Значення може бути *строкою* у подвійних лапках, *числом*, true, false, null, *об'єктом* або *масивом*. Ці структури можуть бути вкладеними.

Строка - це послідовність з нуля або декількох символів Unicode, загорнута в подвійні лапки, що використовує \зворотню косу риску задля екранування. Символ являє собою односимвольну строку. Строка у JSON дуже схожа до строк у C та Java.

Число зображується так само, як у C або Java, за виключенням того, що використовується тільки десяткова система.

ОЛЕГ

3. Практика: створення простого RESTful сервісу

Створення простого RESTful-сервісу на ASP.NET може бути досить прямолінійним завданням, особливо з використанням ASP.NET Web API або новішої технології ASP.NET Core. Ось кілька кроків для створення такого сервісу:

1. Створення нового проекту ASP.NET: Відкрийте Visual Studio і створіть новий проект ASP.NET. Виберіть тип проекту, який підтримує створення веб-служб, наприклад, ASP.NET Web API або ASP.NET Core Web API.

2. Визначення моделей даних: Визначте моделі даних, які будуть використовуватися вашим сервісом. Це може бути клас C#, що відображає сутності вашого додатка.

3. Створення контролерів: Створіть контролери, які будуть обробляти HTTP-запити. У ASP.NET Web API або ASP.NET Core це може бути клас, який наслідується від ApiController або ControllerBase відповідно. У цих контролерах ви описуєте методи, які відповідають на різні HTTP-запити (GET, POST, PUT, DELETE тощо) та виконують відповідні дії.

4. Встановлення маршрутів: Визначте маршрути для вашого сервісу, які вказують, які URL-адреси будуть відповідати різним діям. У ASP.NET Web API це можна зробити за допомогою атрибутів маршрутизації, а у ASP.NET Core це може бути встановлено в методі ConfigureServices() з використанням методу UseEndpoints().

5. Обробка запитів і відповідей: У кожному методі контролера ви реалізуєте логіку для обробки вхідних запитів і створення відповідей. Ви зазвичай використовуєте об'єкти моделі даних для передачі даних між клієнтом і сервером.

6. Тестування: Переконайтеся, що ваш RESTful-сервіс працює належним чином, тестуючи його з використанням різних інструментів для тестування API, таких як Postman, Swagger тощо.

7. Документування: Додайте документацію до вашого сервісу, яка пояснює, як використовувати його. У ASP.NET Core ви можете використовувати Swagger або другі інструменти для автоматичної генерації документації API.

8. Розгортання: Після успішного тестування розгорніть ваш сервіс на відповідному сервері або хмарному середовищі.

Зараз я надам загальний опис того, як можна створити простий RESTful-сервіс для роботи з користувачами на ASP.NET Core. Він буде включати операції створення, читання, оновлення та видалення користувачів. Ось кроки, які вам слід виконати:

1. Відкрийте Visual Studio і створіть новий проект ASP.NET Core Web Application.

2. Створіть клас C#, який представлятиме модель користувача.

3. Створіть контролер.

4. Конфігурація маршрутизації: У файлі Startup.cs у методі Configure перевірте налаштування маршрутизації. Додайте маршрутизацію для вашого

5. Тестування: Запустіть програму та використайте інструменти, такі як Postman, для тестування HTTP-запитів до вашого сервісу. Виконайте операції створення, читання, оновлення та видалення користувачів.

6. Документація і розгортання: Додайте документацію до вашого API, можливо, використовуючи Swagger. Потім розгорніть свій сервіс на відповідному сервері або хмарному середовищі.

СЕРГІЙ

4. Тестування RESTful-сервісів

(Приклад Р3)

Поняття unit-тестів та інтеграційних тестів

Написання тестів вже давно стало невід'ємною частиною написання якісного коду. Найбільш широко використовують два типи тестів – unit-тести та інтеграційні тести.

Unit-тести зосереджуються на одному компоненті програми в повній ізоляції, як правило, на одному класі або функції. В ідеалі перевірений компонент не повинен мати побічних ефектів, тому його максимально легко виділити та протестувати.

Тестування стає складнішим, коли неможливо досягти такого рівня ізоляції. Інші фактори також можуть обмежувати можливості модульного тестування. Наприклад, у мовах програмування з такими модифікаторами доступу, як private або public, ви не можете тестувати приватні функції. Спеціальні інструкції компілятора або позначки іноді допомагають обійти ці обмеження. В іншому випадку вам потрібно заздалегідь планувати код так, щоб для нього можна було легко написати тести.

Ключовим фактором, який робить модульне тестування хорошим вибором, є швидкість його виконання. Оскільки ці тести не повинні мати побічних ефектів, ви можете запустити їх безпосередньо, не залучаючи жодної пов'язаної системи. В ідеалі це дозволяє виключити усі залежності від основної операційної системи, як-от доступ до файлової системи чи мережеві можливості. На практиці можуть існувати певні залежності. Інші можна замінити, створивши mock'и

Модульне тестування також є основою принципу розробки програмного забезпечення, що називається розробкою, керованою тестуванням (Test Driven Development). Вона передбачає написання тестів перед рештою коду. Мета полягає в тому, щоб повністю створити специфікацію окремої одиниці до її реалізації.

Дотримання такого принципу може бути привабливим, він також має помітні недоліки. Специфікація об'єктів має бути дуже точною, а автори тестів мають розуміти принципі створення самих об'єктів. Ця вимога суперечить деяким принципам Agile.

Інтеграційні тести перевіряють, як частини програми працюють разом як єдине ціле.

На відміну від модульного тестування, інтеграційне тестування враховує усі побічні ефекти та залежності.

Наприклад, інтеграційний тест може використовувати з'єднання з базою даних (залежність у модульному тестуванні) для запиту та зміни бази даних, як за нормального використання системи. Вам потрібно буде підготувати базу даних і потім правильно до неї звернутися. Це дозволяє виявити проблеми, які могло приховати unit-тестування із створенням моків.

Хоча межі між різними категоріями тестів розмиті, ключовою властивістю інтеграційного тесту є те, що він має справу з кількома окремими частинами програми. У той час як модульні тести завжди беруть результати з одного блоку, наприклад виклику функції, інтеграційні тести можуть об'єднувати результати з різних частин і джерел.

Тестування RESTful-сервісів за допомогою Postman

Часто при розробці REST API виникає потреба перевірити роботу одного або декількох компонентів, які ще не покриті тестами, або для яких повне покриття неможливе. Для цього часто використовують Postman.

Postman - це платформа для розробки, тестування та взаємодії з API. Вона дозволяє легко відправляти різні HTTP-запити до API, перевіряти його відповіді, налаштовувати параметри запитів, автоматизувати тестування API та багато іншого. Postman надає зручний інтерфейс для роботи з API, що робить процес розробки та тестування API більш ефективним.

Для тестування одного ендпойнта достатньо створити новий запит у Postman:

1. **Створення нового запиту**: Створіть новий запит, вказавши метод (GET, POST, PUT, DELETE тощо) та URL вашого REST API.
2. **Додавання параметрів**: Якщо ваш запит вимагає параметрів, ви можете додати їх у вкладці "Params" або "Body" у разі передачі даних у тілі запиту (для POST та PUT запитів).
3. **Налаштування заголовків**: Додайте необхідні заголовки у вкладці "Headers", наприклад, Content-Type або Authorization.
4. **Відправлення запиту**: Натисніть кнопку "Send", щоб відправити запит на сервер.
5. **Перевірка відповіді**: Перевірте отриману відповідь в розділі "Body" внизу сторінки. Тут ви побачите статус-код відповіді та саму відповідь у форматі JSON або іншому, в залежності від налаштувань.
6. **Тестування різних сценаріїв**: Використовуйте Postman для тестування різних сценаріїв, таких як успішні запити, невірні запити (наприклад, неправильний URL або недійсний токен авторизації.

Для більш складних тестів Postman підтримує створення колекцій запитів, які запускатимуться разом. Щоб створити колекцію:

- Виберіть API на бічній панелі та оберіть потрібне API
- Виберіть «Test and Automation»
- Поруч із Collections натисніть + і виберіть опцію:

    - Add new collection – цей параметр створює нову порожню колекцію в API. Ви можете додати свої тести на вкладку Tests.
    - Copy existing collection – виберіть доступну колекцію зі списку. Копія колекції додається до API.
    - Create from definition: змініть будь-які параметри, щоб налаштувати нову колекцію, і виберіть «Create collection».

SWAGGER

У сучасній практиці для створення серверної частини RESTful сервісу - REST API - часто використовується Swagger. Swagger - це набір інструментів для розробки, створення та документування API. Основний компонент Swagger - це специфікація OpenAPI, яка визначає формат для опису API, включаючи доступні ендпоінти, формати даних та інші деталі.

Swagger надає інструменти для генерації специфікації OpenAPI на основі наявного коду (code-first підхід), автоматичної генерації документації API з використанням специфікації OpenAPI та створення клієнтських SDK на основі OpenAPI специфікації. Swagger підтримує понад 50 різних мов програмування, що робить його дійсно універсальним інструментом та дозволяє уніфікувати підходи до розробки API.

ОЛЕГ

5. Висновок

Архітектура REST - це основа сучасних WEB технологій. Вона дозволила стандартизувати структуру сотень веб-додатків та суттєво спростити взаємодію між клієнтом та сервером. Популярність REST суттєво вплинула на розвиток сучасних web-фреймворків, більшість з яких надають багато засобів для зручного створення RESTful сервісів. Одним з них є ASP.NET Core, про який ми сьогодні розповідали. Він надає численні інструменти та шаблони для швидкого створення для створення RESTful веб-додатків із застосуванням паттерну проєктування MVC, які особливо зручно використовувати разом із Microsoft Visual Studio. MVC - дуже розповсюджений підхід до створення RESTful додатків, який дозволяє розбити їх на незалежні шари, що суттєво спрощує розробку та тестування.

6. Питання