КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА
Факультет радіофізики, електроніки та комп'ютерних систем
Кафедра комп'ютерної інженерії

Створення факультативного курсу лабораторних робіт з розробки веб-служб стилю REST на платформі контейнеризації Docker

Випускна кваліфікаційна робота
студентки 4 року навчання
спеціальність: 123 «Комп'ютерна інженерія»
Оксани ЛУЖНЯК

Науковий керівник:
доцент кафедри комп'ютерної інженерії
Сергій ЗАГОРОДНЮК

Рецензент:
канд. фіз.-мат. наук,
доцент кафедри геоінформатики
ННІ «Інститут геології» Всеволод ДЕМИДОВ

До захисту допускаю: Завідувач кафедри

Юрій БОЙКО

Ухвалено на засіданні кафедри "____"_____2022 р., протокол № ____

Київ - 2022

РЕФЕРАТ

Обсяг роботи 56 сторінок, 45 ілюстрацій, 8 джерел посилань.
ВЕБ-СЛУЖБА, ОБРАЗ КОНТЕЙНЕРА, ВЕБ-ЗАПИТ, ВІРТУАЛЬНА МАШИНА, DOCKER, АВТОРИЗАЦІЯ, КОЛЕКЦІЯ ОБ'ЄКТІВ.
Об'єктом роботи процес створення факультативного курсу з розробки веб-служб на платформі контейнеризації Docker. Предметом роботи є мова програмування C#, інструментарій управління Linux-контейнерами Docker.
Метою роботи є створити покрокові інструкції та варіанти завдань до факультативного курсу лабораторних робіт з розробки веб-служб на платформі контейнеризації Docker. Курс повинен містити необхідний теоретичний матеріал, а також три лабораторні роботи за тематикою адміністрування і програмування веб-служб стилю REST.

Перша лабораторна робота повинна бути присвячена налаштуванню платформи Docker на операційній системі Linux, створення веб-служби з веб-функцією додавання простий об'єктів даних та веб-функцією перевірки існування доданих об'єктів.

Друга лабораторна робота повинна бути присвячена розробці веб-функцій, які дозволяють накопичувати і змінювати складні об'єкти даних.

Третя лабораторна робота повинна бути присвячена авторизованому доступу до веб-служби за механізмом автентифікації JWT (JSON Web Token).
Методи розроблення: вивчення теоретичного матеріалу, узагальнення і структуризація досвіду налаштування веб-служб, розробка завдань та тематики лабораторних робіт. Інструменти розроблення: середовище розробки Visual Studio 2019, фреймоворк Asp.Net Core, мова програмування C#, інструментарій Docker.
Результати роботи: виконано огляд використання служби контейнеризації Docker, проаналізовано переваги і недоліки веб-служб стилю REST, створено курс лабораторних робіт, який дозволяє студентам опанувати технологію контейнеризації та розробку веб-служб.

Створений в кваліфікаційні роботі курс лабораторних робіт в перспективі може розширити зміст нормативного курсу «Системне програмне забезпечення» та інших дисциплін кафедри комп’ютерної інженерії.

ЗМІСТ

ВСТУП 5
ТЕОРЕТИЧНА ЧАСТИНА 7
РОЗДІЛ 1. ОСНОВНІ ТЕОРЕТИЧНІ ВІДОМОСТІ 7
1.1 Загальні відомості про Docker 7
1.1.1 Що таке Docker? 7
1.1.2 Архітектура Docker 8
1.1.3 Переваги використання Docker 9
1.2 Загальні положення про веб-служба Restful 9
1.2.1 Що таке Restful веб-служби? 9
1.2.2 Архітектура Restful 10
ПРАКТИЧНА ЧАСТИНА 12
РОЗДІЛ 2. ЛАБОРАТОРНІ РОБОТИ. ЗАВДАННЯ ТА ОПИС 12
2.1 Перша лабораторна робота 12
2.1.1 Мета першої лабораторної роботи 12
2.1.2 Завдання першої лабораторної роботи 12
2.1.3 Покрокове виконання першої лабораторної роботи 12
2.2 Друга лабораторна робота 34
2.2.1 Мета другої лабораторної роботи 34
2.2.2 Завдання до другої лабораторної роботи 34
2.2.3 Приклад виконання другої лабораторної роботи 35
2.3 Третя лабораторна робота 43
2.3.1 Мета роботи третьої лабораторної роботи 43
2.3.2 Завдання лабораторної роботи 44
2.3.3 Приклад виконання 44
ВИСНОВКИ 55
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 56

ВСТУП

Оцінка сучасного стану об’єкта розробки. Одним з найбільш популярних інструментів для програмної віртуалізації є Docker – автоматизований засіб для керування віртуальними контейнерами. Він вирішує багато завдань, пов’язаних з створенням контейнерів, розміщенням в них додатків, керуванням процесами, а також тестуванням ПЗ та його окремих компонентів.

REST являє собою архітектурний стиль для забезпечення стандартів між комп’ютерними системами в Інтернеті, що спрощує взаємодію систем між собою. Rest-сумісні системи, які часто називають RESTful, характеризуються тим, що не мають стану і розділяють проблеми клієнта і сервера. Рой Філдінг класифікував існуючу архітектуру поточної реалізації, так і визначив, які аспекти слід вважати центральними для поведінкових і продуктивних вимог Інтернету.

Актуальність роботи та підстави для її виконання. Контейнеризація є чудовою альтернативою апаратної віртуалізації. Всі процеси в ній протікають на рівні операційної системи, що дозволяє суттєво економити ресурси та збільшувати ефективність роботи з додатками. Саме тому студентам необхідно у процесі вивчення мережевих технологій отримати навички практичного застосування контейнеризації під час створення веб-служб. Тому актуальним є створення курсу лабораторних робіт по розробці веб-служб за допомогою платформи контейнеризації Docker, під час виконання яких студенти можуть ознайомитись з контейнеризацією.

Мета і завдання роботи. Метою кваліфікаційної роботи є створення завдань та покрокових інструкції до лабораторних робіт. Для досягнення цієї мети створено такі завдання.

- Розробити опис та завдання до лабораторних робіт.
- Розробити веб-служби, які працюють в операційній системі GNU Linux, а також створенню клієнтів до цієї служби
- Розгорнути веб-служби за допомогою Docker

Об'єкт, методи й засоби розроблення. Об’єктом роботи процес створення факультативного курсу з розробки веб-служб на платформі контейнеризації Docker.

Під час розробки лабораторних робіт використовувалась еволюційна модель. Це означає, що спочатку створена пробна версія, яка передана викладачу для оцінки. Після цього, робимо аптейдт відповідно до рекомендацій викладача. Це відбувається стільки разів, аж до поки вона лабораторна робота не починає відповідати вимогам.

Інструментом розроблення веб-служб є середовище розробки Visual Studio 2019, мова програмування C#, фреймворк Asp.Net Core та служба контейнеризації Docker

C# — це сучасна, об'єктно-орієнтована мова програмування загального призначення. Вона була розроблена Microsoft під керівництвом Андерса Хейлсберга та його команди в рамках ініціативи .NET і схвалена Європейською асоціацією виробників комп’ютерів (ECMA) та Міжнародною організацією зі стандартів (ISO). C# є однією з мов для загальномовної інфраструктури. Синтаксично C# дуже схожий на Java і є простим для користувачів, які знають C, C++ або Java.

Можливі сфери застосування. Курс лабораторних робіт може розширити зміст нормативного курсу «Системне програмне забезпечення» та інших спецкурсів.

ТЕОРЕТИЧНА ЧАСТИНА

РОЗДІЛ 1. ОСНОВНІ ТЕОРЕТИЧНІ ВІДОМОСТІ

1.1 Загальні відомості про Docker.

1.1.1 Що таке Docker?

Docker – це ПЗ з відкритим вихідним код, яке застосовується для розробки, тестування, доставки і запуску веб-додатків в середовищах з підтримкою контейнеризації. Він потрібен для більш ефективного використання системи і ресурсів, швидкого розгортання готових програмних продуктів, а також для їх масштабування і переносу в інші середовища з гарантійним збереженням стабільної роботи.

Основний принцип роботи Docker – контейнеризація Цей тип віртуалізації дозволяє упаковувати програмне забезпечення по ізольованим середовищам-контейнерам. Кожен з цих віртуальних блоків отримає всі необхідні елементи для роботи програми. Це дає можливість одночасного запуску великої кількості контейнерів на одному хості.

Docker-контейнери працюють у різних середовищах: локальному центрі обробки інформації, хмарі, персональних комп'ютерах тощо.

Docker допомагає викладати ваш код швидше, швидше тестувати, швидше викладати програми та зменшити час між написанням коду та запуску коду. Платформа та засоби контейнерної віртуалізації можуть бути корисними у таких випадках:

- пакування додатка (і так само компонентів, що використовуються) в контейнери docker;
- роздача та доставка цих контейнерів командам для розробки та тестування;
- викладання цих контейнерів на продакшени, як у датацентри, так і в хмари.

Docker чудово підходить для організації циклу розробки. Docker дозволяє розробникам використовувати локальні контейнери з програмами та сервісами. Що згодом дозволяє інтегруватися з процесом постійної інтеграції та викладання (continuous integration and deployment workflow).

1.1.2 Архітектура Docker

Docker використовує архітектуру клієнт-сервер. Docker-клієнт спілкується з демоном Docker, який бере на себе створення, запуск, розподіл контейнерів. Клієнт та сервер можуть працювати на одній системі, можна підключити клієнт до віддаленого демона docker. Клієнт та сервер спілкуються через сокет або через RESTful API.

Docker складається з трьох компонентів:

- Образи (images)
- Регістри (registries)
- Контейнери (containers)

Образи:

Docker-образ – це read-only шаблон. Наприклад, образ може містити ОС Ubuntu з Apache і додатком на ній. Образи використовуються для створення контейнерів. Docker дозволяє легко створює нові образи, обновляти вже створені, або завантажувати образи, які були створені іншими людьми. Образи – це компонента збірки docker-a.

Регістри:

Docker-регістри зберігає образи. Є публічні і приватні регістри, з яких можна завантажити або загрузити образи. Регістри – це компонента поширення.

Контейнери:

Контейнери схожі на директорії. В контейнерах міститься все, що потрібно для роботи програми. Кожен контейнер створюється з образу. Контейнери можуть бути створені, запущені, зупинені, перенесені або видалені. Кожен контейнер ізольований і є безпечною платформою для програми. Контейнери – це компонента роботи.

1.1.3 Переваги використання Docker.

1. Мінімальне використання ресурсів – контейнери не віртуалізують всю операційну систему, а використовують ядро хоста і ізолюють програму на рівні процесу. Останній використовує набагато менше ресурсів, ніж віртуальна машина.
2. Швидкісне розгортання – допоміжні компоненти можна не встановлювати, а використовувати вже готові docker-образи. Наприклад, немає сенсу постійно встановлювати і налаштовувати Linux Ubuntu. Достатньо один раз її інсталювати, створити образ і постійно його використовувати, обновляючи версію при необхідності.
3. Зручне приховування процесів – для кожного контейнера можна використовувати різні методи обробки даних, приховуючи фонові процеси.
4. Робота з небезпечним кодом – технологія ізоляції контейнерів дозволяє запускати будь-який код без шкоди для ОС.
5. Просте масштабування – будь-який проект можна розширити, впровадивши нові контейнери.
6. Зручний запуск – програму, яка знаходиться всередині контейнера можна запустити на будь-якому docker-хості.
7. Оптимізація – файлової системи – образ складається з шарів, які дозволяють дуже ефективно використовувати файлову систему.

1.2 Загальні положення про веб-служба Restful

1.2.1 Що таке Restful веб-служби?

Restful веб-служба – це веб-служба побудована на архітектурі REST. Веб-служба Restful, надає клієнту API з додатку безпечним, одноманітним способом без збереження стану. Клієнт може виконувати певні операції, використовуючи сервіс Restful. Основним протоколом для REST є HTTP.

REST – це спосіб доступу до ресурсів, які знаходяться в певнім середовищі. Наприклад, є сервер, на якому розміщуються важливі документи, зображення чи відео. Все це приклад ресурсів. Якщо клієнт, тобто, веб-браузер потребує будь-який із цих ресурсів, він повинен відправити запит на сервер для доступу до цих ресурсів. Тепер REST визначає спосіб доступу до цих ресурсів.

1.2.1 Ключові елементи реалізації Restful

- Ресурси – це ключова абстракція, на якій концентрується протокол HTTP.
- Методи запитів – вказують, на дію, яка буде виконуватись над запитом.

Основні методи:

- GET: отримати детальну інформацію про ресурс
- POST: створити новий ресурс
- PUT: обновити існуючий ресурс
- DELETE: видалити ресурс

- Заголовки запита – це додаткові інструкції, які надсилаються разом з запитом.
- Тіло запита – дані, які відправляються разом з запитом. Дані зазвичай відправляються в запиті, коли POST-запит зроблений до веб-службу REST. При виклику POST клієнт фактично повідомляє веб-службу, що він хоче додати ресурс на сервер. Звідси, тіло запита буде містити інформацію про ресурс, який потрібно додати на сервер.
- Тіло відповіді – це основна частина відповіді.
- Коди стану відповіді – це коди, які повертаються разом з відповіддю веб-служба.

1.2.2 Архітектура Restful

Додаток або архітектура вважається RESTful, якщо їй притаманні такі характеристики:

- Стан та функціональність представлені у вигляді ресурсів – це означає, що кожен ресурс має бути доступним через звичайні HTTP-запити GET, POST, PUT або DELETE. Так, якщо хтось хоче отримати файл на сервері, у них має бути можливість відправити GET запит і отримати файл. Якщо він хоче завантажити файл на сервер, то він повинен мати можливість використовувати POST або PUT-запит. Нарешті, якщо він хоче видалити файл, має бути можливість надіслати запит DELETE.
- Архітектура клієнт-сервер, відсутність стану (stateless) та підтримка кешування:

- Клієнт-сервер – звичайна архітектура, де сервером може бути веб-сервер, на якому розміщено програму, а клієнтом – звичайний веб-браузер;
- Архітектура без збереження стану означає, що стан програми не зберігається у REST. Наприклад, якщо ви видалили ресурс з сервера командою DELETE, навіть при отриманні позитивного коду відповіді немає гарантій, що він дійсно був видалений. Щоб переконатися, що ресурс видалений, необхідно надіслати запит GET. З його допомогою можна запросити ресурси, щоб подивитися, чи є там віддалений.

ПРАКТИЧНА ЧАСТИНА

РОЗДІЛ 2. ЛАБОРАТОРНІ РОБОТИ. ЗАВДАННЯ ТА ОПИС

2.1 Перша лабораторна робота

2.1.1 Мета першої лабораторної роботи

Навчитись розгортати середовище розробки .Net Core на ОС Linux. Налаштовувати службу контейнеризації Docker в ОС Linux. Створити веб-службу та створити клієнтську програму для неї. Навчитись користуватись низькорівневим відлагоджувачем Postman.

2.1.2 Завдання першої лабораторної роботи

Створіть веб-службу, яка за допомогою HTTP-запитів перевіряє наявність імені/виду птахів/сорту квітів у таблиці MySQL/текстовому файлі/бінарному файлі, виводить список усіх об’єктів та додає нове об’єкт у таблицю MySQL/ текстовий файл/бінарний файл.

Перевірте роботу служби за допомогою HTTP-клієнта Postman. Зробіть скріншоти з результатами перевірки.

Створіть клієнтську частину у вигляді застосунку Windows Form, у якій три кнопки виконують три функції веб-служби.

2.1.3 Покрокове виконання першої лабораторної роботи.

Насамперед, використовуючи ПЗ VirtualBox, створимо віртуальну машину на якій інсталюємо ОС Centos 7:

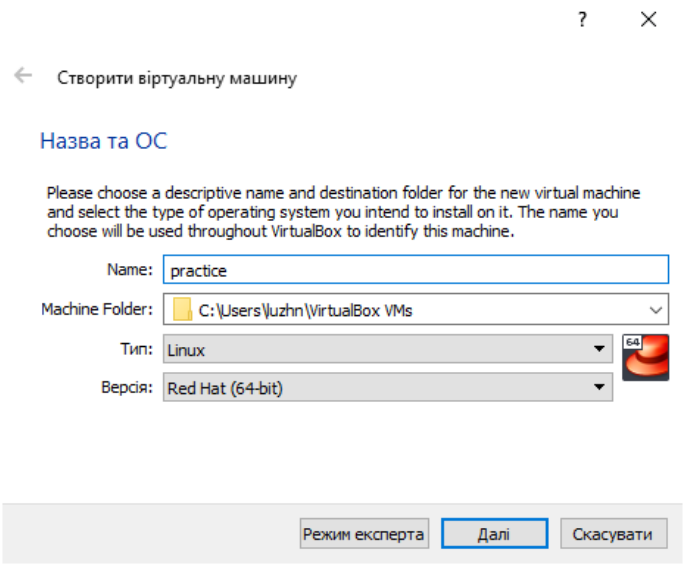

Рис. 1 Створення віртуальної машини

Проінсталюємо операційну систему:

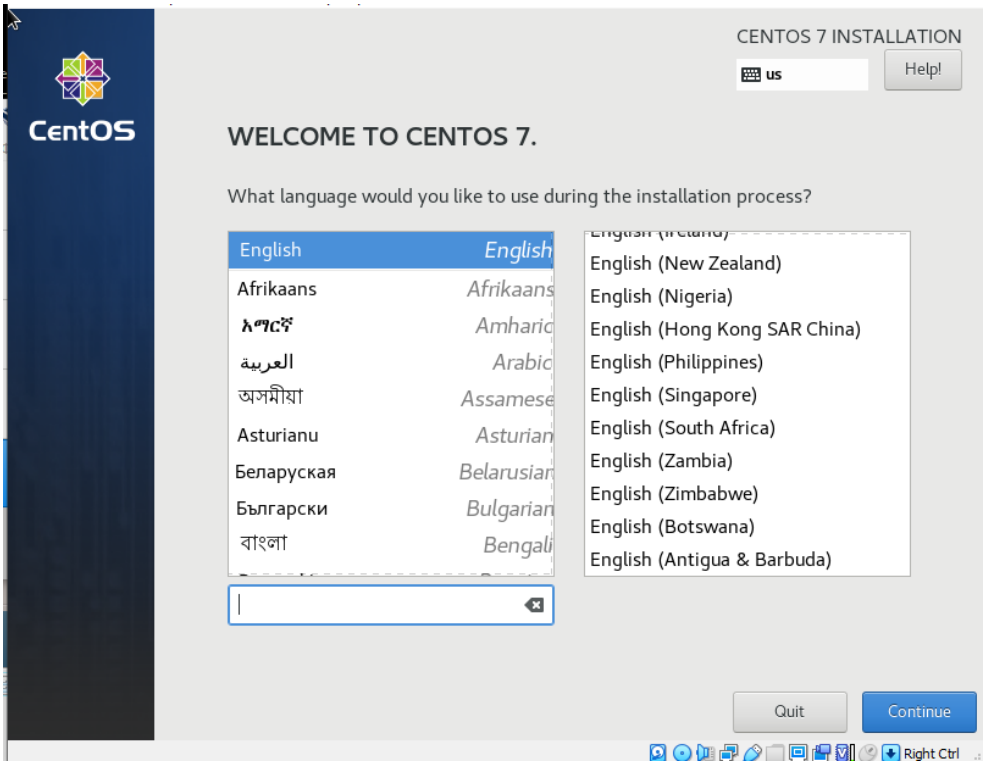

Рис. 2 Вибір мови системи

Для зручності встановимо Gnome Desktop:

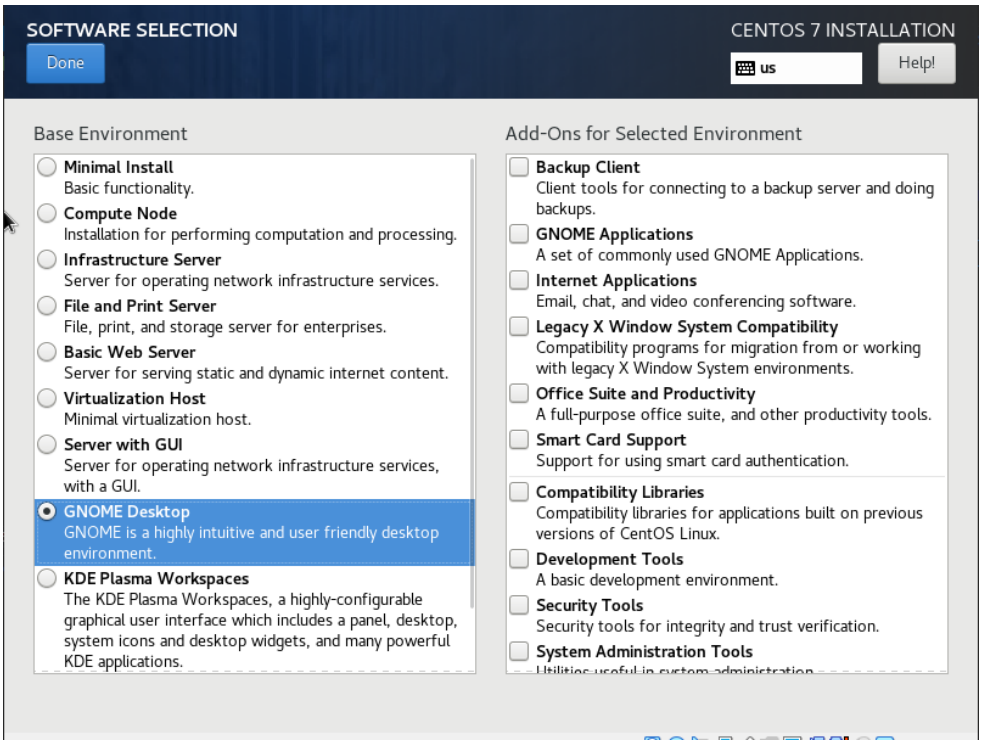

Рис. 2 Вибір графічного інтерфейсу

Обов’язково вказуємо пароль для користувача root:

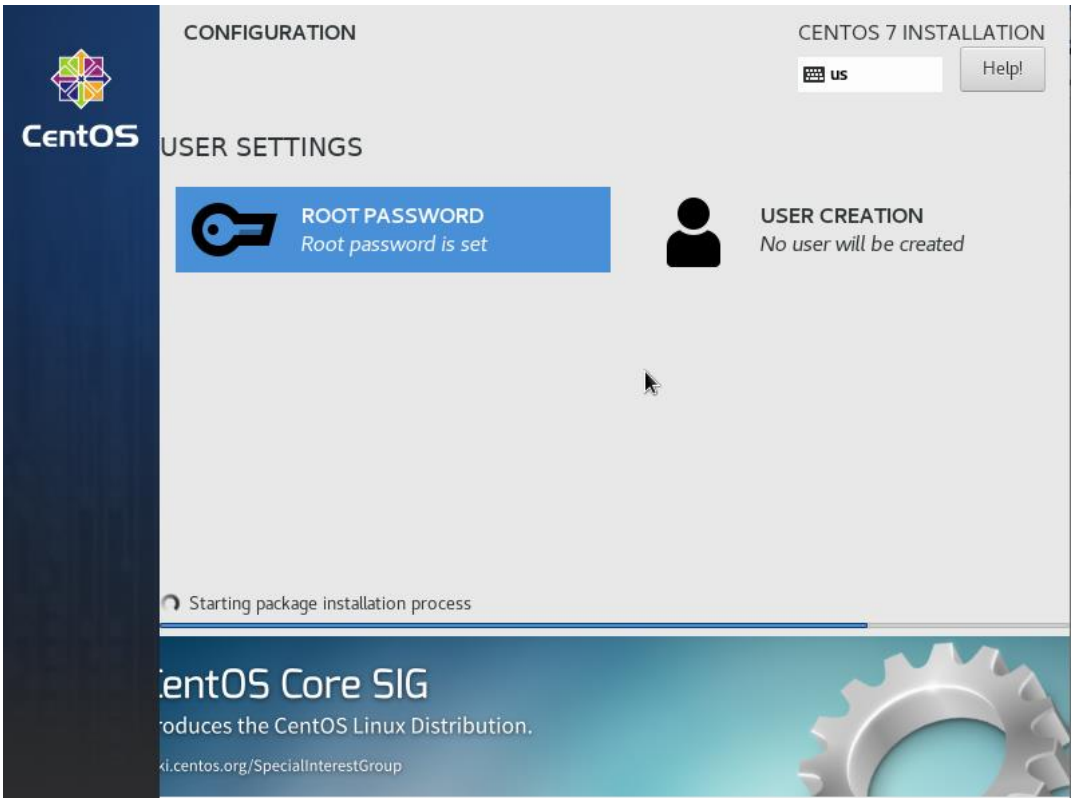

Рис. 3 Встановлення root пароля і початок інсталяції

Після завершення інсталяції переходимо в налаштування віртуальної машини на вкладку *Мережа*, розгортаємо пункт *Додатково*, та натискаємо на *Переадресування порту:*

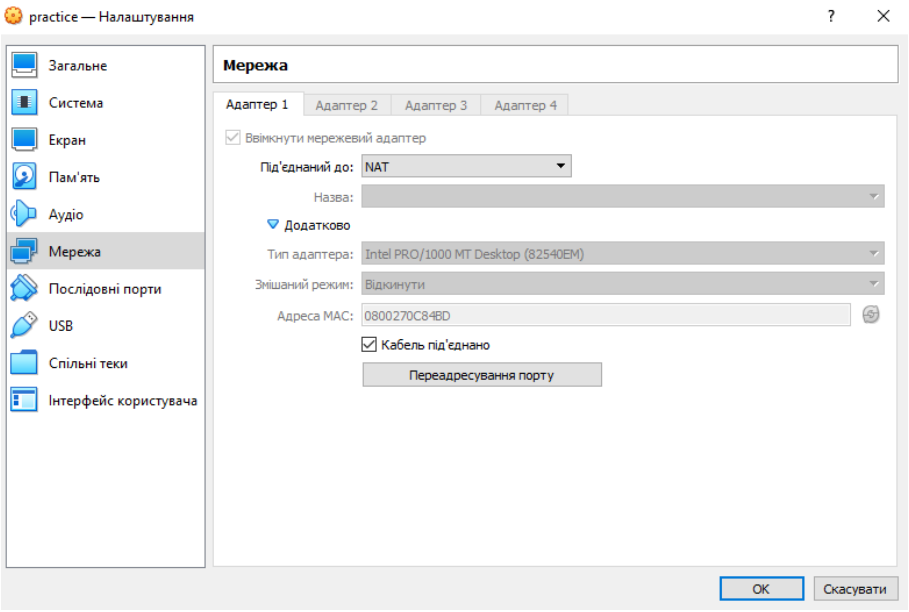

Рис. 4 Переадресація порту

Переадресовуємо 22 порт, для цього створюємо нове TCP-правило:

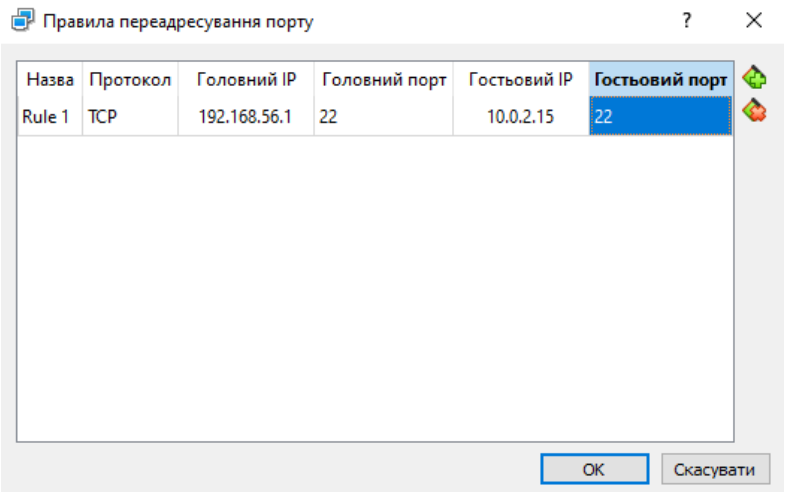

Рис. 5 Створення нового TCP-правила

Після цього відриваємо програму Putty та через неї з'єднуємось з віртуальною машиною:

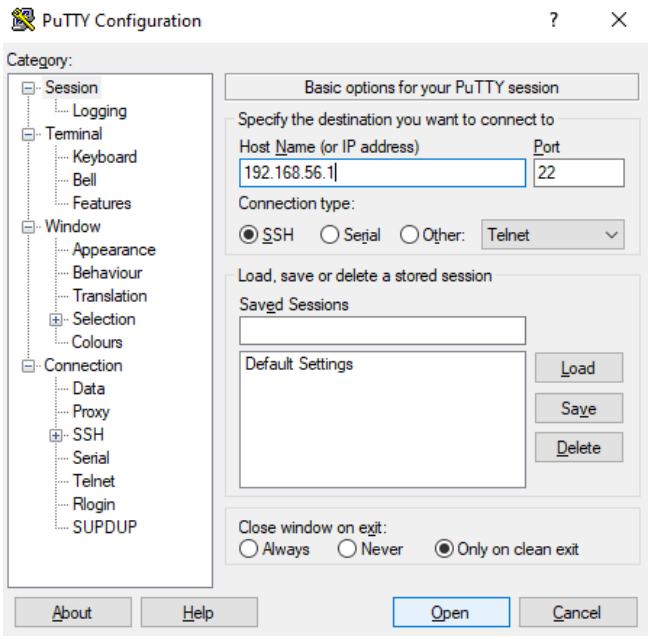

Рис. 6 Підключення до віртуальної машини за допомогою Putty

Авторизуємось:

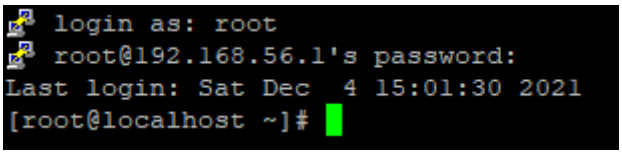

Рис. 7 Авторизація

Для перенесення даних з хостовою машиною зручно використовувати протокол sftp. Одним і зручних ПЗ для використання даного протоколу є Total Commander.

Отже, спочатку встановимо його на наш хост. Перейдемо за посиланням https://www.ghisler.com/download.htm . Завантажимо 64-бітну версію. Встановимо його. Далі нам потрібен спеціальний плагін. За посиланням https://www.ghisler.com/plugins.htm знайдемо завантаження плагіну SFTP, розпакуємо його у зручне місце. Потім нам потрібно доданий цей плагін до Total Commander. Отже, натискаємо на вкладку *Configuration*, обираємо пункт *Options.*

У меню *Configuration* перейдемо до *Plugins* ліворуч і в розділі *File system plugins* (.WFX) натиснемо *Configure.*

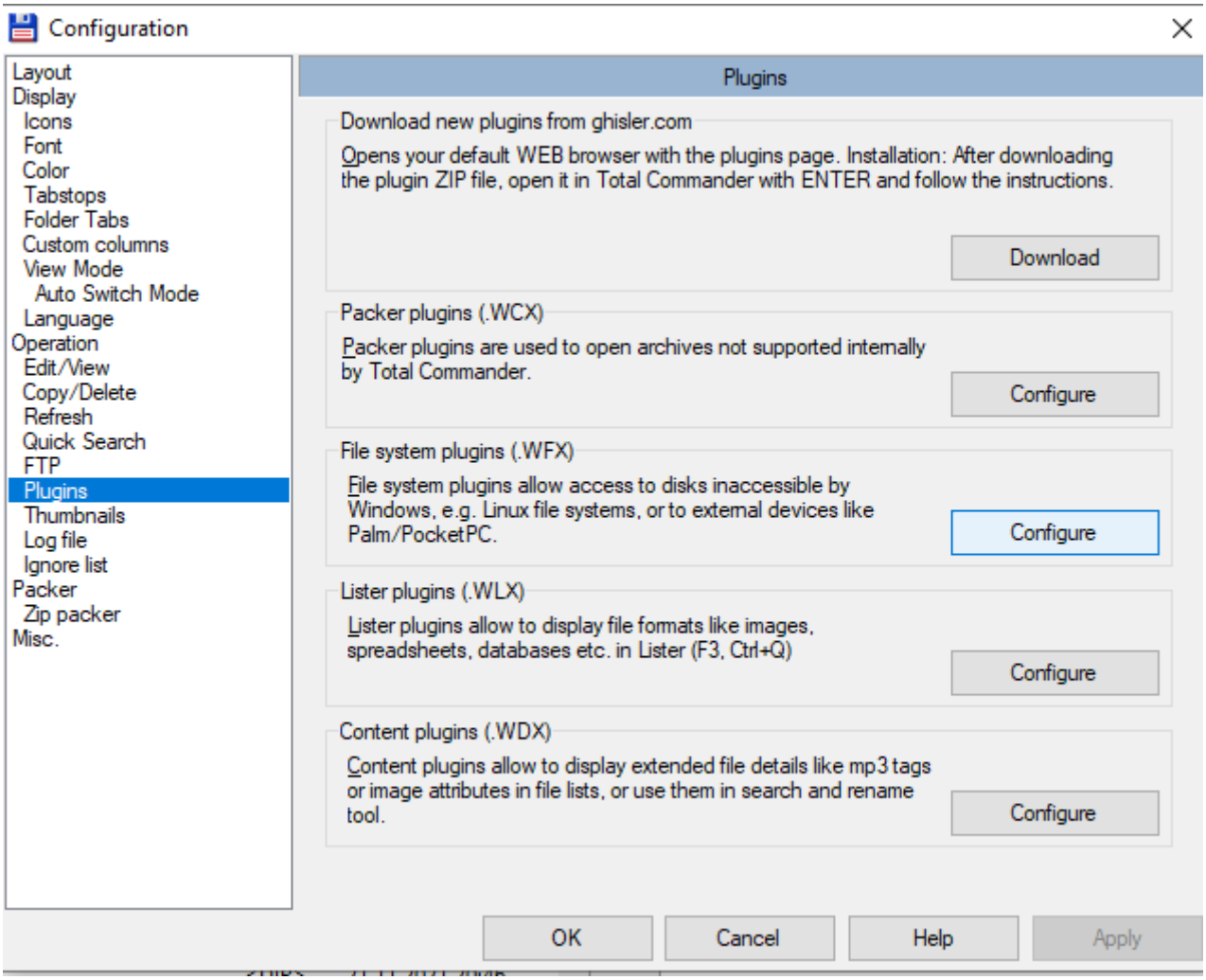

Рис. 8 Меню Configuration

З'явиться нове вікно з встановленими плагінами файлової системи. Натиснемо *Add* під списком.

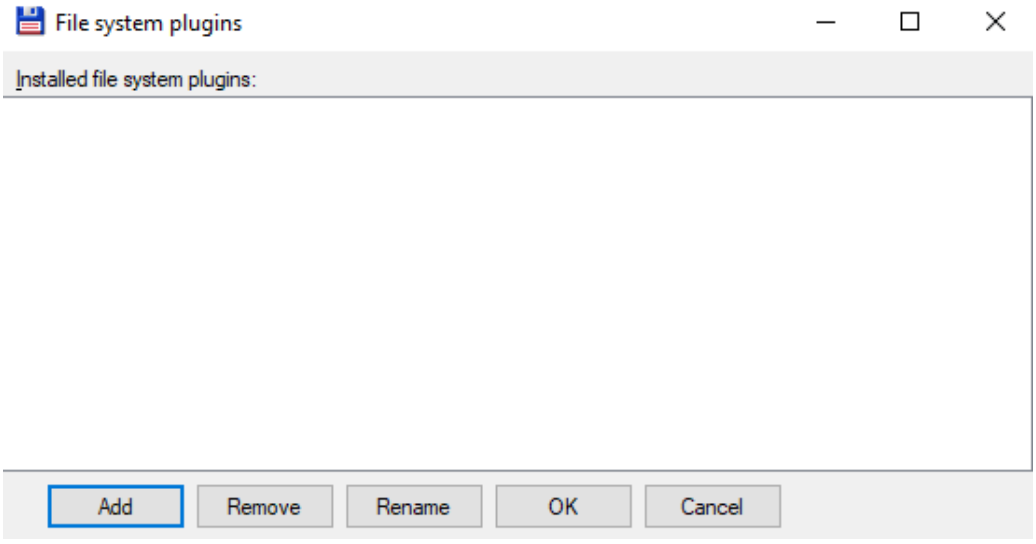

Рис.9 File system plugins

Знаходимо плагін, який розпакували раніше з розширенням .wfx та натискаємо відкрити. Даний плагін з’явився у списку. Натискаємо *Ok*.

У списку дискових одиниць вибираємо *Network Neighborhood*. Натискаємо *Secure FTP connection*. Потім натискаємо F7. Створюємо нове з’єднання. Назву можна вибрати довільну. В полі *Connect to* пишемо адресу віртуальної машини. В полях *username* і *password* вказуємо логін користувача та його пароль. Натискаємо *OK*:

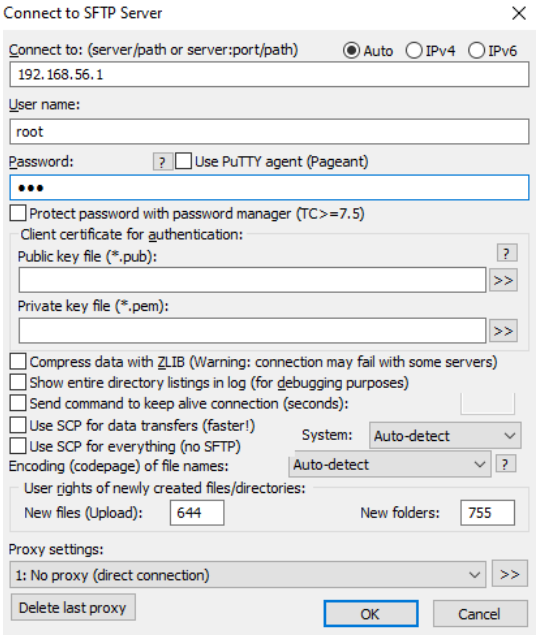

Рис.10 Підключення до SFTP сервера

При правильному налаштуванні можемо побачити дерево файлів нашої віртуальної машини:

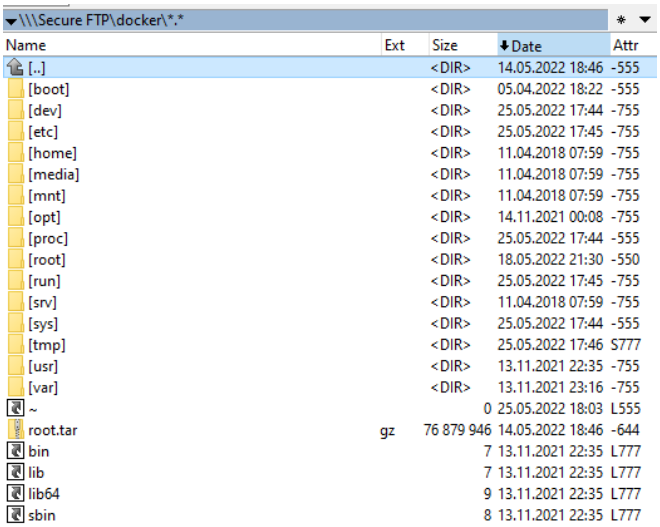

Рис.11 Дерево файлів віртуальної машини

Отже, за допомогою цих налаштувань фактично ми можемо користуватись віртуальною машиною не заходячи на неї.

Тепер оновлюємо систему за допомогою команди `yum update`.

Після оновлення перезавантажимо систему використовуючи команду `reboot`.

Встановимо пакети для роботи з Docker. Команди, які потрібно виконати:

```
yum install yum-utils device-mapper-persistent-data lvm2
yum-config-manager --add-repo
https://download.docker.com/linux/centos/docker-ce.repo
yum install docker-ce
yum install docker-ce-cli
yum install containerd.io
```

Перевіримо чи працює Docker. Спочатку запустимо службу Docker, а потім перевіримо її статус:

```
systemctl start docker
systemctl status docker
```

Рис. 12 Перевірка служби Docker

Тепер налаштовуємо середовище для .Net. Виконаємо команди:

```
rpm -Uvh https://packages.microsoft.com/config/centos/7/packages-
microsoft-prod.rpm
yum install dotnet-sdk-5.0
```

Для перевірки створимо тестовий каталог та стандартну консольну програму, яка виведе “Hello World!”

```
mkdir test
cd test/
dotnet new console -o App -n NetCore.Docker
cd App/
dotnet run
```

```
[root@localhost ~]# mkdir test
[root@localhost ~]# cd test/
[root@localhost test]# dotnet new console -o App -n NetCore.Docker
The template "Console Application" was created successfully.

Processing post-creation actions...
Running 'dotnet restore' on App/NetCore.Docker.csproj...
  Determining projects to restore...
  Restored /root/test/App/NetCore.Docker.csproj (in 571 ms).
Restore succeeded.

[root@localhost test]# cd App/
[root@localhost App]# dotnet run
Hello World!
```

Рис. 13 Робота .Net середовища

Як бачимо середовище успішно налаштоване та готове для наших подальших дій.

Створимо каталог /root/sf і в ньому файл users.txt та в ручну додамо кілька логінів

Для зручності спочатку створимо веб-службу на ОС Windows.

Спочатку запускаємо Visual Studio 2019. Створюємо пустий проект Asp.Net:

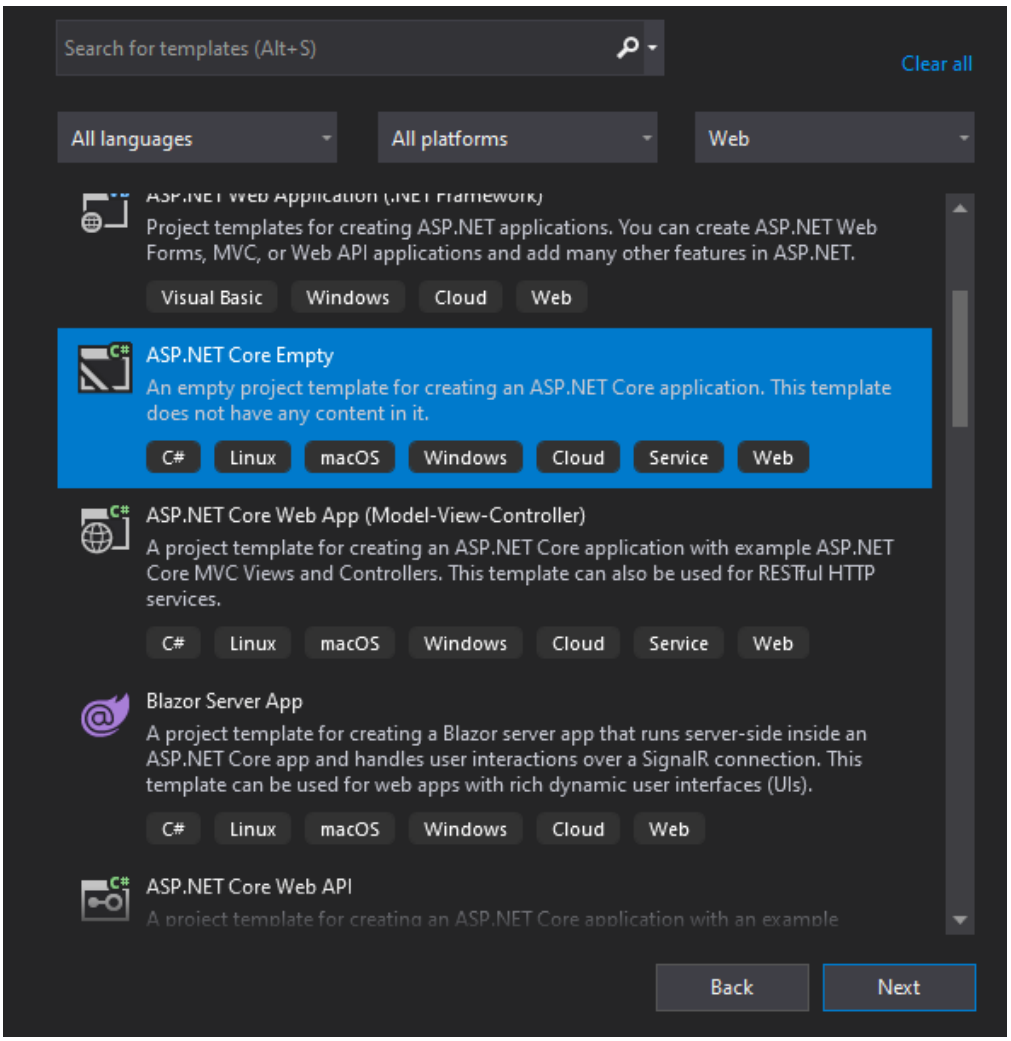

Рис. 14 Створення проекту Asp.Net

Для розробки використаємо фреймворк .Net 5.0.

Для роботи з форматом JSON нам потрібен пакет Newtonsoft.Json. Натискаємо на вкладку *Tools*, потім обираємо *NuGet Package Manager -> Manage NuGet Packages for Solution…*

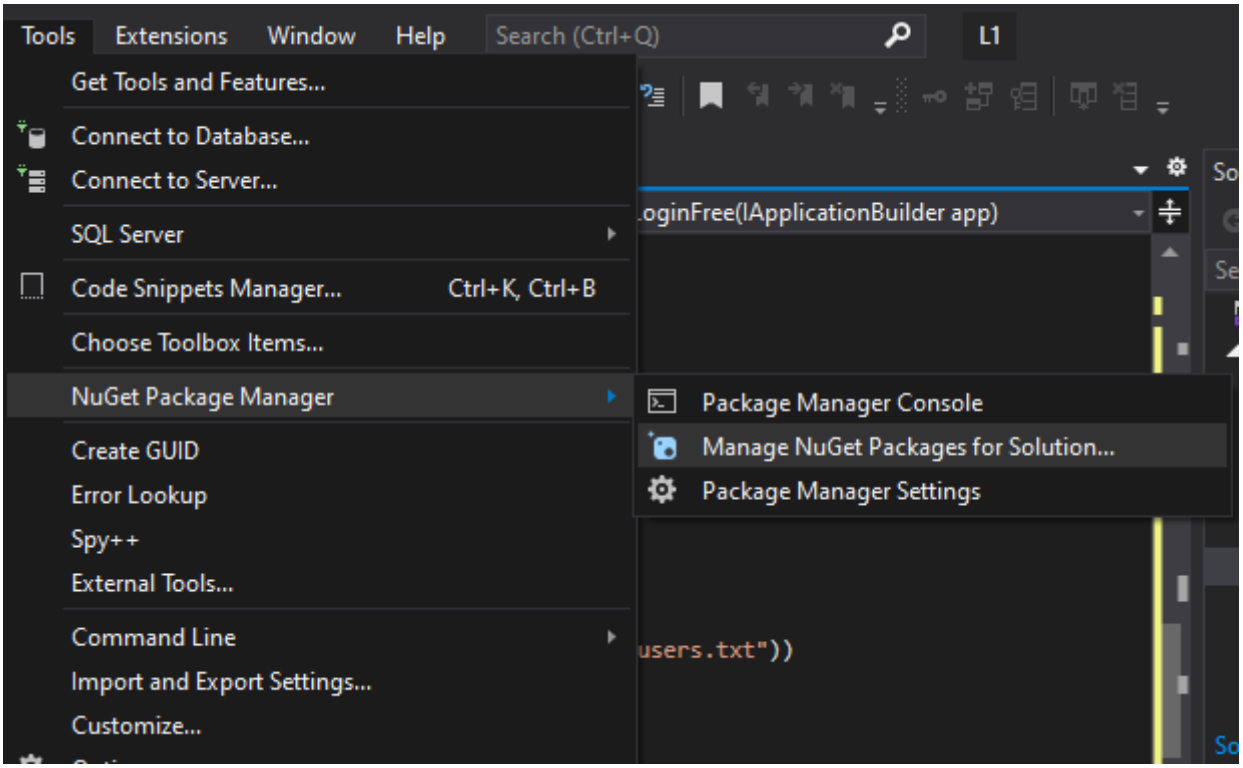

Рис.15 Вкладка Tools

У вкладці *Browser* знаходимо потрібний пакет, обираємо версію 10.0.1 та встановлюємо для нашого проекту.

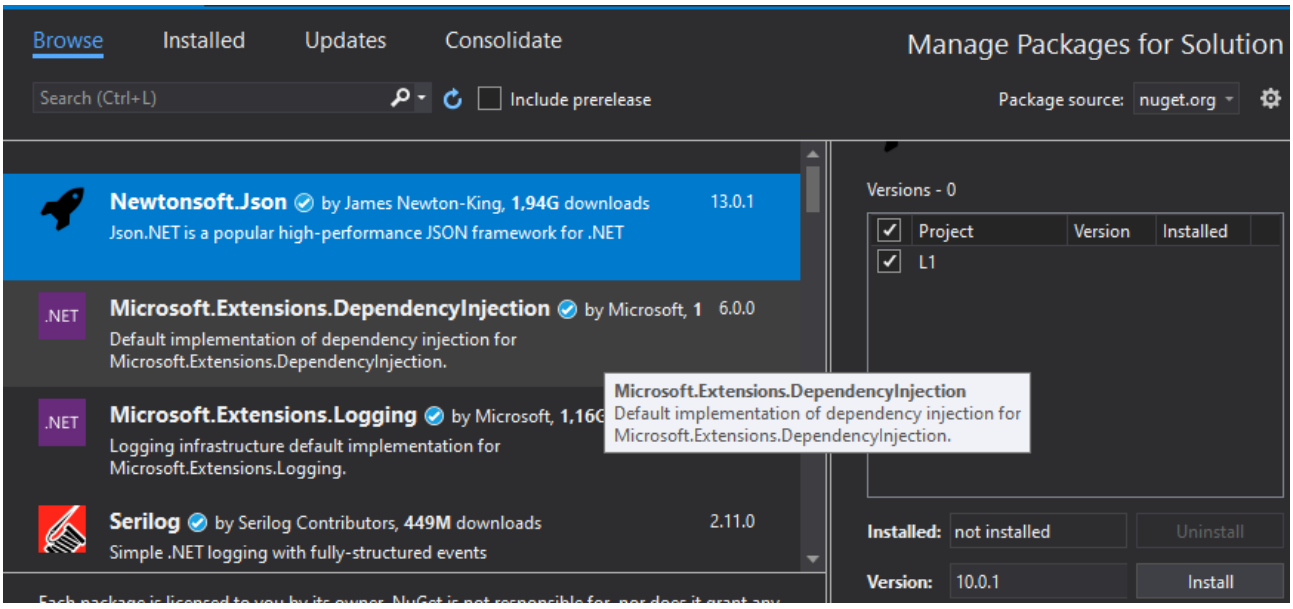

Рис.16 NuGet Package Manager

У будь-якому типі проектів ASP.NET, як і в проекті консольної програми, ми можемо знайти файл Program.cs, в якому визначено однойменний клас Program і з якого починається виконання програми. Клас Startup є вхідною точкою до програми ASP.NET. Цей клас здійснює конфігурацію програми, налаштовує сервіси, які програма буде використовувати, встановлює компоненти для обробки запиту.

Нижче наведений вміст файлу Startup.cs, який вирішує поставлену задачу, а саме веб-служба, який має три сторінки. Перша - /IsLoginFree – яка перевіряє чи вільний логін, який ми передаємо як параметр запиту GET. Друга - /AllLogins – яка виводить кортеж всіх логінів у файлі за допомогою запиту GET. І на сам кінець - /AddLogin – яка додає логін, який вказаний в тілі запиту до файлу, якщо його там немає.

Сам код:

```
app.Map ("/AllLogins", AllLogins);
app.Map("/AddLogin", AddLogin);
app.Run(async (context) =>
{
await context.Response.WriteAsync("Docker in OS Linux");
});
}
```

Метод для перевірки чи вільний логін. Спочатку створюємо булеву зміну для перевірки. Після чого відриваємо файловий потік для зчитування файлу використовуємо директиву using для того щоб одразу закривати потік. Зчитуємо перший рядок файлу, перевіряємо чи він рівний параметру методу, якщо так булева змінна дорівнює false, якщо ні, то циклічно зчитуємо наступний рядок файлу, поки не знайдемо рівність, чи не закінчаться рядки. Метод повертає значення змінної isFree.

```
public static bool IsFree(string login)
{
bool isFree = true;
using (StreamReader fileObj = new StreamReader("/user/src/app/shared_folder/users.txt"))
{
string s = fileObj.ReadLine();
while (s != null)
{
if (s == login)
```

```
{
isFree = false;
}
s = fileObj.ReadLine();
}
}
return isFree;
}
```

Метод для додавання нового логіна. Спочатку ініціалізовуємо пустий рядок. Потім перевіряємо чи немає такого логіну у файлі. Якщо він вільний, то відкриваємо файловий потік над нашим файлом та записуємо його у файл. Після чого виводимо повідомлення про це. Якщо логін зайнятий, то виводим відповідне повідомлення.

```
public static string NewLogin(string login)
{
string text = "";
if (IsFree(login))
{
using (StreamWriter fileObj = new StreamWriter("/user/src/app/shared_folder/users.txt",
true))
{
fileObj.WriteLine(login);
}
text = string.Format("Login {0} is successful added", login);
}
else
{
text = string.Format("Login {0} is already exist", login);
}
return text;
}
```

Метод для виведення кортежу всіх логінів. Спочатку створюємо пустий список з рядків. Потім відкриваємо файловий потік для читання. Потім зчитуємо перший рядок, якщо він не дорівнює null, то додаємо його у список і зчитуємо наступний рядок. Виходимо з циклу тоді, коли зчитаний рядок дорівнює null. Метод повертає список логінів.

```
public static List<string> Users()
{
List<string> users = new List<string>();
using (StreamReader fileObj = new StreamReader("/user/src/app/shared_folder/users.txt"))
{
string line = fileObj.ReadLine();
```

```
while (line != null)
{
users.Add(line);
line = fileObj.ReadLine();
}
}
return users;
}
```

Делегат конфігурує окремий конвеєр проміжного програмного забезпечення. Він викликає метод Run, який додає компоненти в конвеєр. У стрінгову змінну записуємо значення параметру. Відповіддю на запит буде серіалізований результат роботи методу IsFree() параметром якого буде параметр запиту.

```
private static void IsLoginFree(IApplicationBuilder app)
{
app.Run(async context =>
{
string login = context.Request.Query["login"];
await context.Response.WriteAsync(JsonConvert.SerializeObject(IsFree(login)));
});
}
```

Делегат AddLogin присвоює стрінговій змінній значення тіла запиту. Після чого відкриваємо файловий потік для зчитування над цим значення зчитуємо значення з потоку і записуємо його у нову змінну. Після чого парсимо його в JSON. Відповіддю на запит буде серіалізований результат роботи методу NewLogin().

```
private static void AddLogin(IApplicationBuilder app)
{
app.Run(async context =>
{
var request = context.Request.Body;
string login;
using (StreamReader reader = new StreamReader(request))
{
string body = await reader.ReadToEndAsync();
login = JObject.Parse(body)["login"].ToString();
}
await context.Response.WriteAsync(JsonConvert.SerializeObject(NewLogin(login)));
});
}
```

Відповіддю на запит, який пов'язаний з даним делегатом, буде серіалізований результат роботи методу Users()

```
private static void AllLogins(IApplicationBuilder app)
```

```
{
app.Run(async context =>
{
await context.Response.WriteAsync(JsonConvert.SerializeObject(Users()));
});
}
}
```

Після створення коду програми натискаємо правою кнопкою миші на проект, потім *Add->Docker Support*

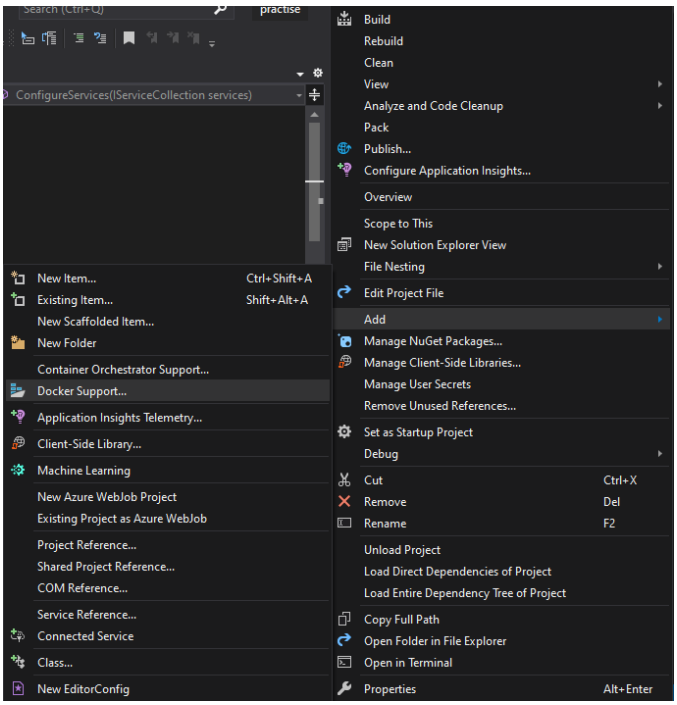

Рис. 17 Додавання Dockerfile

Обираємо ОС Linux.

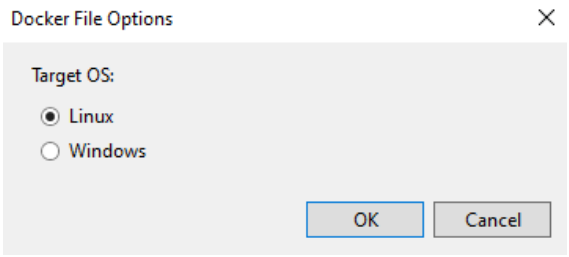

Рис. 18 Вибір цільової ОС для Dockerfile

Зверніть увагу, що якщо у Dockerfile збірка та публікація проекту написана з папки Release, то і розробка повинна буде Release.

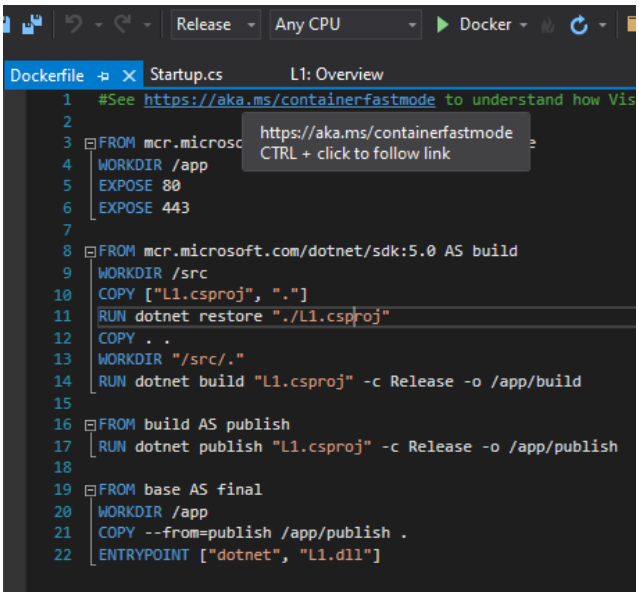

Рис. 19 Dockerfile

Тепер потрібно завантажити проект в Linux. За допомогою Total Commander перенесемо папку в Linux

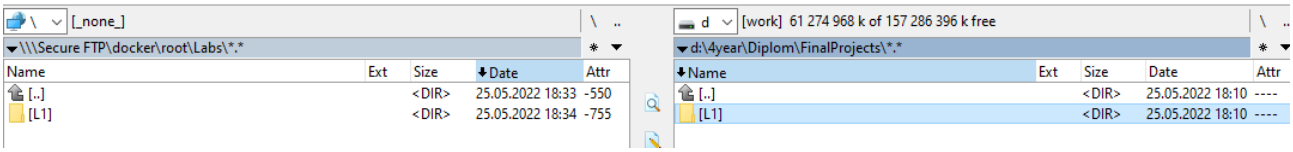

Рис. 20 Перенесення папки

Далі необхідно розгорнути наш сервіс в docker. Перейдемо в каталог, де розташований наш Dockerfile. Створюємо образ за допомогою команди :

`docker build --tag=l1 .` (в кінці команди КРАПКА!)

Перевіримо чи створився образ. Для цього виведемо список всіх образів за допомогою команди

`docker image ls`

```
[root@localhost L1]# docker image ls
REPOSITORY                          TAG       IMAGE ID       CREATED          SIZE
l1                                  latest    0a5212e1897b   4 minutes ago    206MB
<none>                              <none>    5c40b35242af   4 minutes ago    716MB
mcr.microsoft.com/dotnet/sdk        6.0       903b367ed1c5   6 months ago     715MB
mcr.microsoft.com/dotnet/sdk        5.0       326ee605795b   6 months ago     631MB
mcr.microsoft.com/dotnet/aspnet     5.0       6c1368a6aa6c   6 months ago     205MB
python                              3         4246fb19839f   7 months ago     917MB
hello-world                         latest    feb5d9fea6a5   8 months ago     13.3kB
```

Рис. 21 Список образів

Запустимо контейнер, при цьому виконаємо мапінг внутрішнього каталогу контейнера */user/src/app/shared_folder* з зовнішним каталогом */root/sf*

`docker container run -d -p 5000:80 -v /root/sf:/user/src/app/shared_folder l1`

Виведемо список всіх контейнерів

```
docker container ls -a
```

```
[root@localhost L1]# docker container ls -a
CONTAINER ID   IMAGE          COMMAND             CREATED          STATUS                      PORTS                                                NAMES
e50125f1b147   l1             "dotnet L1.dll"     51 seconds ago   Up 49 seconds               443/tcp, 0.0.0.0:5003->80/tcp, :::5003->80/tcp       charming_bardeen
4f2ec4d27fbd   hello-world    "/hello"            6 months ago     Exited (0) 6 months ago                                                          practical_poitras
```

Рис. 22 Список усіх контейнерів

Перевіримо працездатність сервісу перейшов за адресою:

http://192.168.56.1:5003/AllLogins

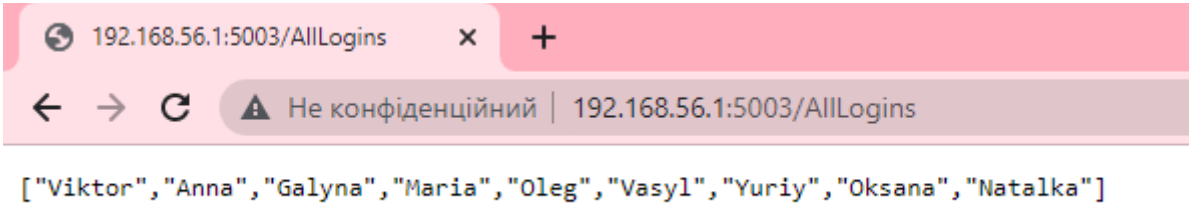

Рис. 23 Перевірка працездатності сервісу

Бачимо, що сервіс успішно розгорнений та працює.

Postman – це HTTP-клієнт для тестування API. HTTP-клієнти тестують відправку запитів клієнта на сервер і отримання відповіді від сервера.

Протестуємо запити, які відправляє наш веб-служба за допомогою Posrman-a.

Спочатку протестуємо запит для виведення всіх логінів. Обираємо метод GET та пишемо наш URL : *http://192.168.56.1:5003/AllLogins* натискаємо кнопку *Send:*

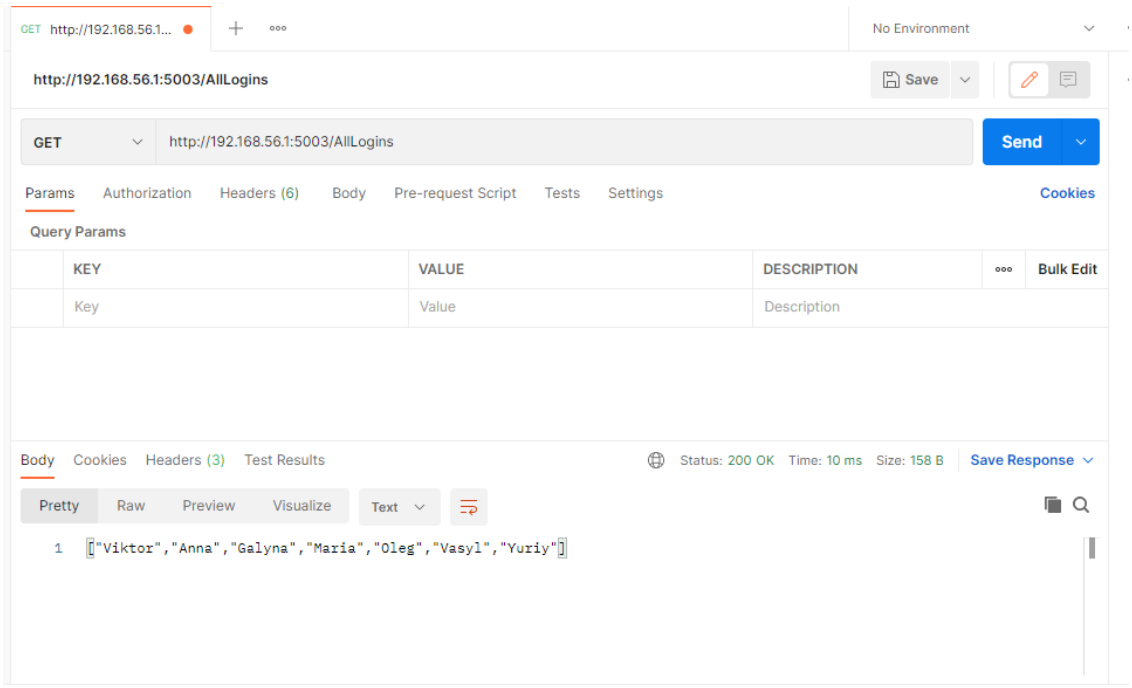

Рис. 24 Запит GET для виводу всіх логінів

Тепер перевіримо чи працює запит для перевірки зайнятості логіну. Обираємо метод GET, вводимо URL - http://192.168.56.1:5003/IsLoginFree , в параметрах пишемо логін для перевірки. Після чого натискаємо кнопку *Send:*

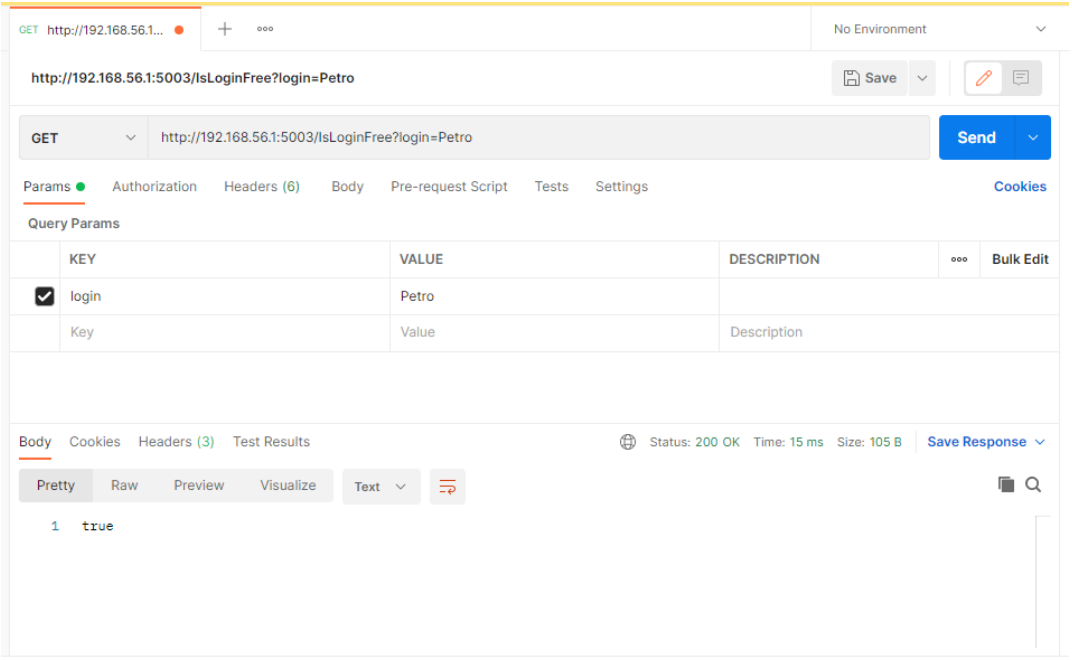

Рис. 25 Запит GET для перевірки зайнятості логіну

Отже, заданий логін вільний, перевіримо також чи запит працює правильно. Для цього вкажемо в значення параметру логін, який точно є файлі:

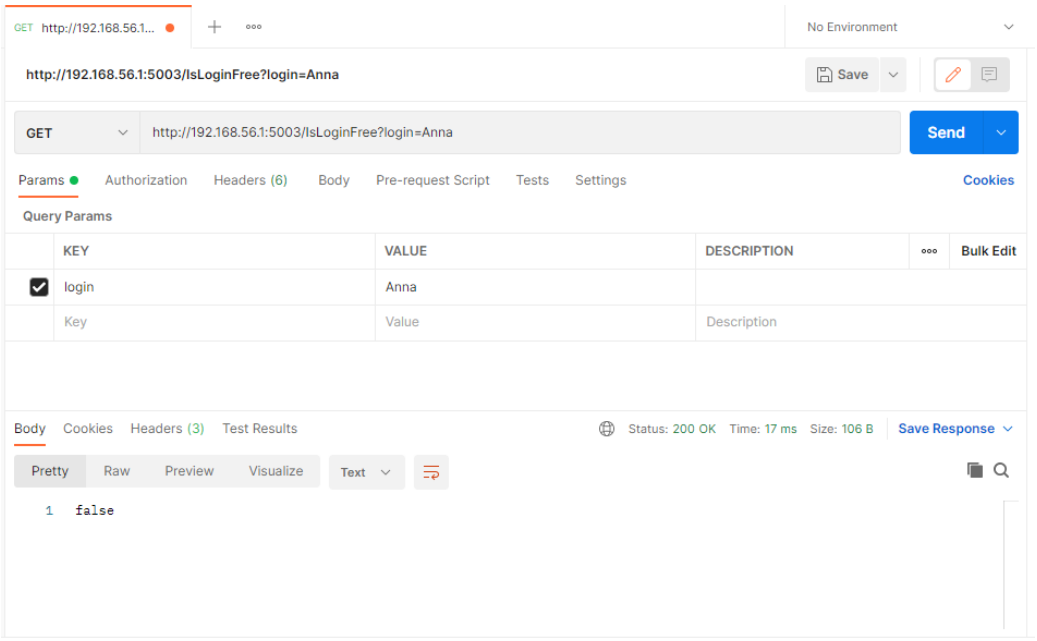

Рис. 26 Запит GET для перевірки зайнятості логіну

Такий логін і дійсно є в файлі, отже, він зайнятий. Тобто запит працює правильно.

Тепер спробуємо додати логін в файл. Для цього змінюємо метод на POST, вводимо URL - http://192.168.56.1:5003/AddLogin , в параметрах переходимо на вкладку *Body*, обираємо *raw* та вказуємо тип *JSON*. Після чого пишемо логін, який хочемо додати в такому форматі:

```
{"login":"Oksana"}
```

Натискаємо кнопку *Send*:

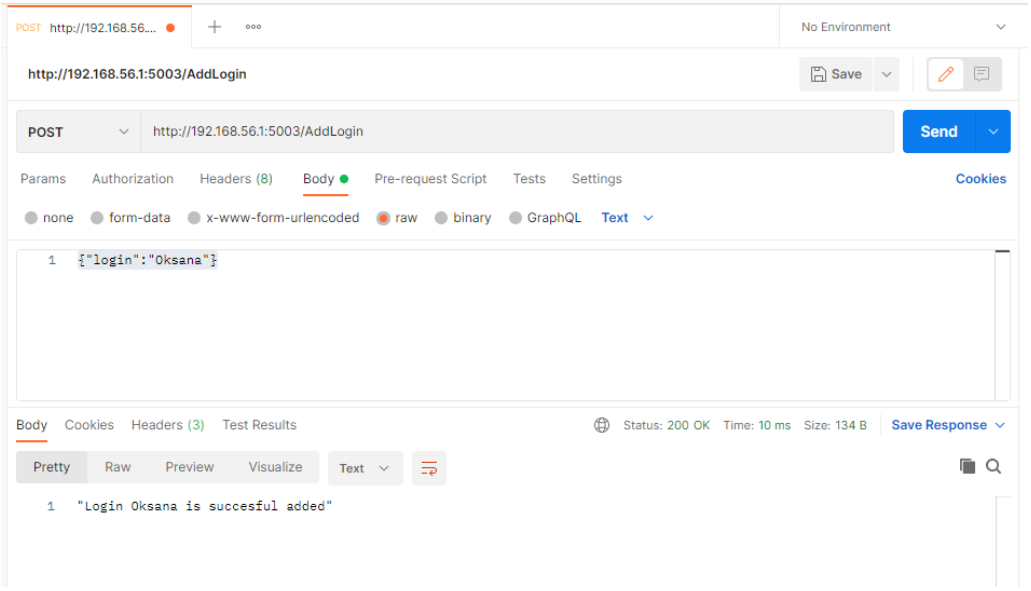

Рис. 27 Запит POST для додавання логіну

Якщо ж захочемо знову додати цей ж логін, отримаємо повідомлення, що він вже існує:

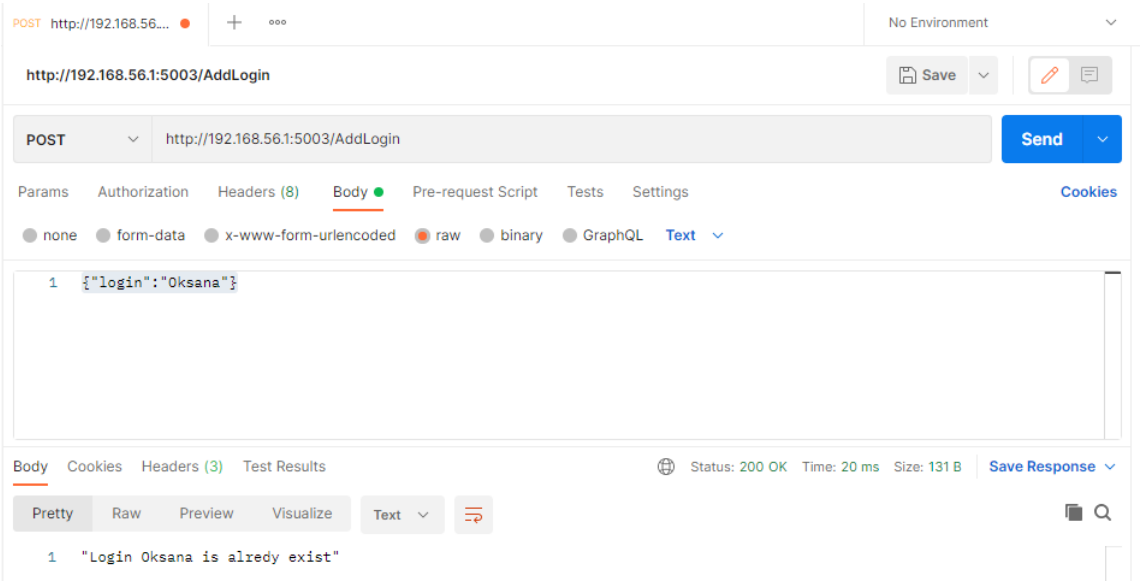

Рис. 28 Запит POST для додавання логіну

Перевіримо чи логін додався:

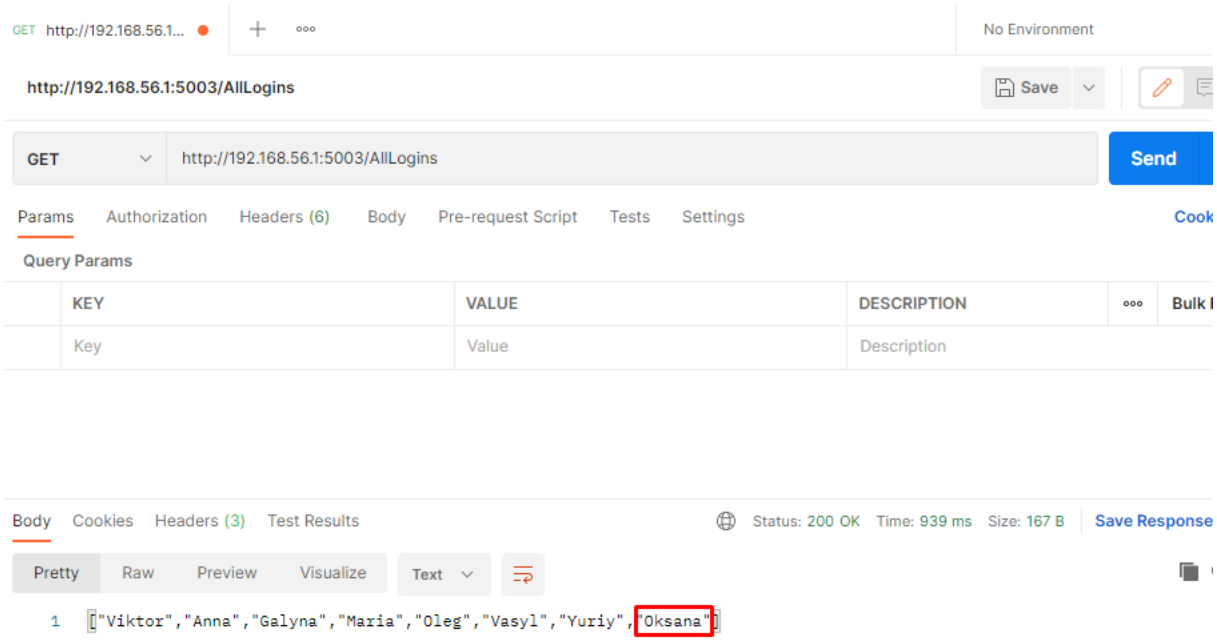

Рис. 29 Перевірка чи відбулось додавання логіну

Оскільки успішно є файлі, робимо висновок, що всі три запити працюють правильно.

Тепер створимо клієнтську частину – Windows Form, яка буде виконувати ті ж функції. В Visual Studio 2019 створюємо

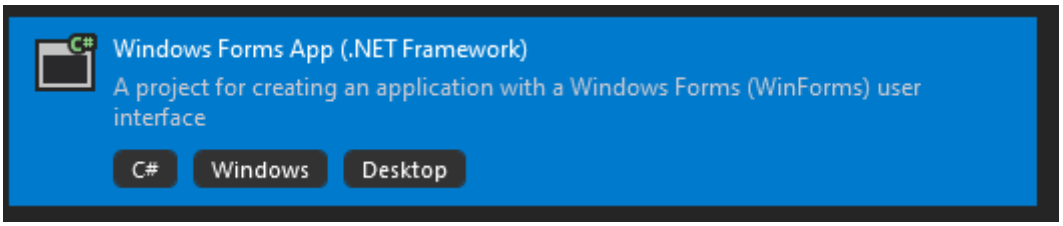

Рис. 30 Створення Windows Form

Наш клієнт буде мати вигляд:

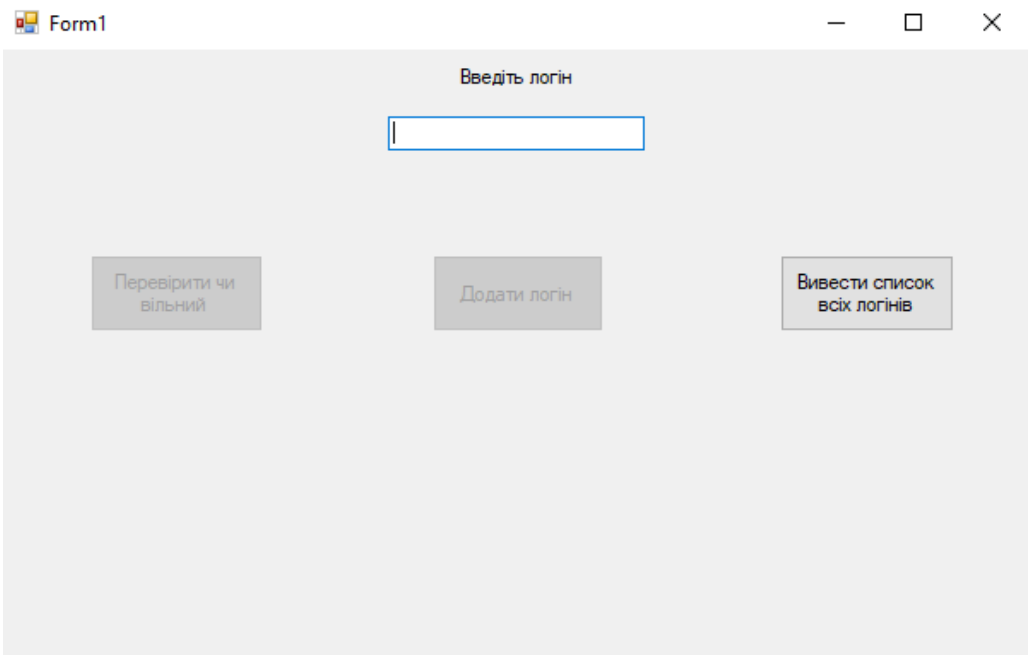

Рис.31 Вигляд клієнта

При натиску на першу кнопку проводиться перевірка на зайнятість логіну, на другу – додавання логіну, на третю – вивід всіх логінів.

Код Form1.cs:

Клас графічної форми:

```
public partial class Form1 : Form
{
```

Конструктор, у якому викликається метод, який виконує ініціалізацію компонентів форми з файла дизайнера

```
public Form1()
{
InitializeComponent();
}
```

Метод, для заблокування кнопок якщо текстове поле порожнє

```
private void textBox1_TextChanged(object sender, EventArgs e)
{
button1.Enabled= !string.IsNullOrWhiteSpace(textBox1.Text);
button2.Enabled= !string.IsNullOrWhiteSpace(textBox1.Text);
}
```

Метод, який викликається при натиску на першу кнопку. Робимо видимим label2, створюємо стрінгову змінну. Гарантуємо видалення веб-клієнта за межами using, веб-клієнт використовується для відправки запитів. Записуємо відповідь з даного url в змінну та присвоюємо label2 її значення

```
private void button1_Click(object sender, EventArgs e)
{
```

```
label2.Visible = true;
string B;
using (WebClient W = new WebClient())
{
B = W.DownloadString("http:192.168.56.1:5003/IsLoginFree?login=" + textBox1.Text);
label2.Text = B;
}
}
```

метод, який викликається при натиску на третю кнопку. Робимо видимим label2, створюємо стрінгову змінну. Гарантуємо видалення веб-клєнта за межами using, веб-клієнт використовується для відправки запитів Записуємо відповідь з даного url в змінну та присвоюємо label2 її значення

```
private void button3_Click(object sender, EventArgs e)
{
label2.Visible = true;
string B;
using (WebClient W = new WebClient())
{
B = W.DownloadString("http:192.168.56.1:5003/AllLogins");
label2.Text = B;
}
}
```

Метод, який викликається при натиску на другу кнопку. Робимо видимим label2. Гарантуємо видалення веб-клєнта за межами using, веб-клієнт використовується для відправки запитівколекція Headers, щоб встановлює заголовок HTTP Content-Type на application/json, вивантажуємо вказаний рядок на вказаний ресурс за допомогою методу POST, для виводу присвоюємо label2 нашу відповідь

```
private void button2_Click(object sender, EventArgs e)
{
label2.Visible = true;
using (WebClient W = new WebClient())
{
W.Headers[HttpRequestHeader.ContentType] = "application/json";
string response = W.UploadString("http:192.168.56.1:5003/AddLogin", "{ \"login\":
\""+textBox1.Text+"\"}");
label2.Text = response;
}
}
}
```

Перша кнопка:

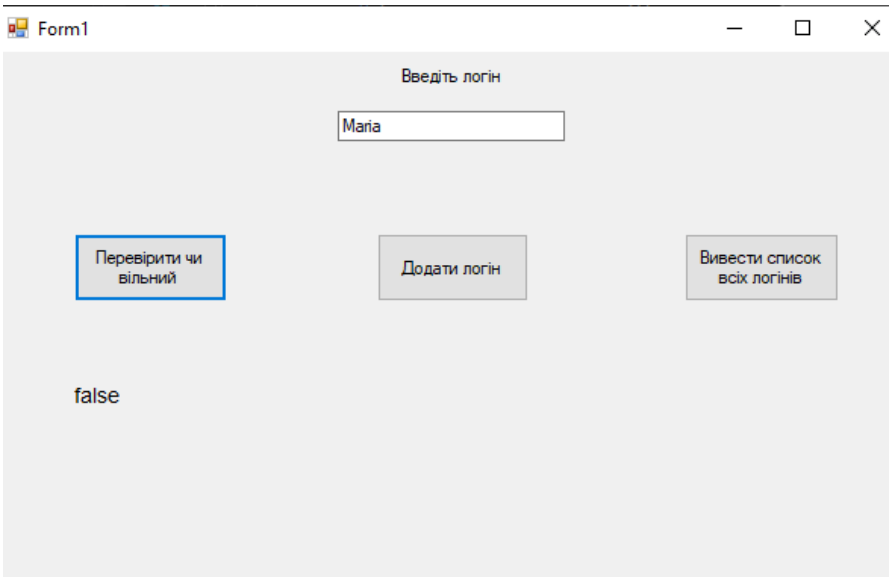

Рис. 32 Робота першої кнопки

Друга кнопка:

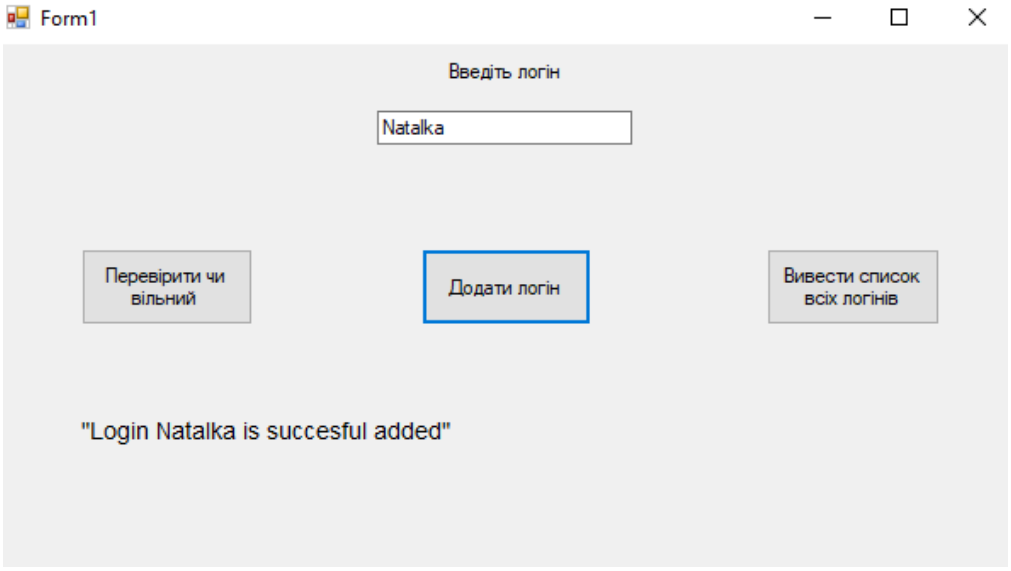

Рис. 33 Робота другої кнопки

Третя кнопка:

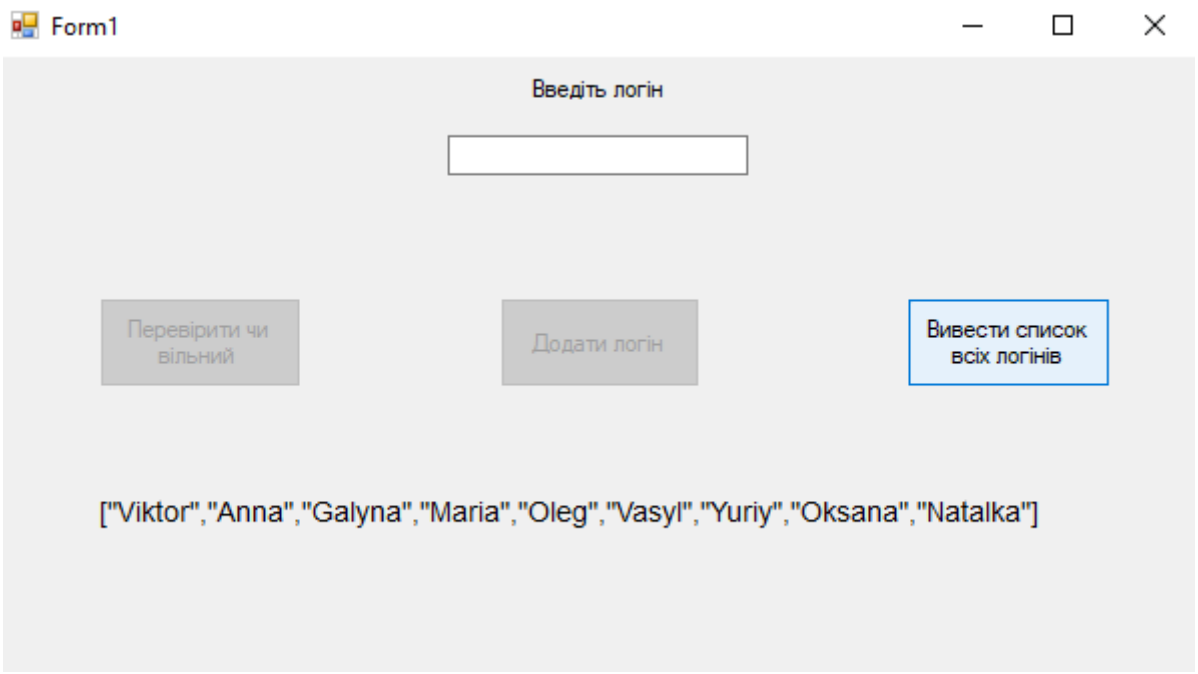

Рис. 34 Робота третьої кнопки

Видно, що всі три кнопки працездатні і форма працює

2.2 Друга лабораторна робота

2.2.1 Мета другої лабораторної роботи

Навчитись працювати з колекціями об’єктів, що представлені у форматі JSON. Створити веб-службу, яка виконує певні дії над колекціями об'єктів та над самими об'єктами.

2.2.2 Завдання до другої лабораторної роботи

Створіть веб-службу для роботи з колекціями об'єктів. Колекція об'єктів має бути представлена у вигляді списку JSON-об’єктів. Об’єкт повинен мати 6-8 полів. Варіанти об’єктів - список рубрик телеканалу/список фільмів/список книг. Веб-служба повинна виконувати такі завдання. Обов’язково додати валідацію значень. Виводити колекцію об’єктів, додавати новий об’єкт чи кілька у існуючу, збереження колекції у новий файл. Відповідно до варіанту повинні бути реалізовані також наступні функції – фільтрування по певному полю, після чого виведення відфільтрованих об’єктів / пошук об’єкту та його

виведення/видалення об’єктів за певною умовою, після чого виведення нової колекції.

2.2.3 Приклад виконання другої лабораторної роботи

Створимо нову веб-службу. Для роботи з колекцією потрібно створити ще два класи. Перший клас буде описувати поля об’єкту. У цьому прикладі об’єктом є особа. У неї є 10 полів, які характеризують її. Інший клас – це клас для обробки даних, для роботи з класом Person. У ньому повинна бути описана основна робота веб-служби. Тобто, завантаження даних, їх збереження, фільтрування за певної ознакою, у прикладі відбувається фільтрування за статтю, тобто, виводимо список тих, об’єктів, які належать до однієї з статей; пошук особи, у якої така ж електронна адреса, як і значення параметру запиту; додавання нового об’єкту колекції до існуючої; оновлення даних, видалення особи. Також клас містить метод, для перевірки валідності даних.

Отже, нижче наведені код веб-служби.

Клас Person описує набір властивостей, який визначає об'єкт класів. Також є атрибути для перевірки валідності даних та властивість, яка обраховує вік кожної особи та задає його.

```
public class Person
{
public string FirstName { get; set; }
public string LastName { get; set; }
public string DateofBirth { get; set; }
[EmailAddress(ErrorMessage = "Invalid Email Address")]
public string Email { get; set; }
[RegularExpression(@"^([0-9]{10})$", ErrorMessage = "Invalid Phone Number.")]
public string Phone { get; set; }
public string City { get; set; }
public string Adrress { get; set; }
public string Country { get; set; }
public string Sex { get; set; }
public string Age
{
get
{
var dateTime = DateTime.Parse(this.DateofBirth);
int age = (DateTime.Today.Year - dateTime.Year);
if (DateTime.Today.DayOfYear < dateTime.DayOfYear)
{
age--;
}
return age.ToString();
}
}
}
```

Клас People для роботи зі класом Person:

```
public class People
```

```
{
public static JsonSerializerOptions options = new JsonSerializerOptions { WriteIndented =
true };
public List<Person> DataPeople { get; set; }
```

Конструктор класу, у якому завантажуються дані з файлу.

```
public People(string filename)
{
LoadData(filename);
}
```

Метод завантаження даних: спочатку ініціалізовуємо новий екземпляр FileInfo, який виступає обгорткою для шляху до файлу, потім якщо файлу за шляхом filename не існує, то створюємо новий файл. Після цього відриваємо файловий потік для зчитування файлу та використовуємо директиву using для того, щоб одразу закривати потік; зчитуємо у стрінгову змінну значення файлу, десеріалізуємо вміст файлу і записуємо у властивість DataPeople, при виникненні будь-яких виключень у DataPeople записуємо пусти список класу Person.

```
public void LoadData(string filename)
{
FileInfo fi = new FileInfo(filename);
if (!fi.Exists)
{
FileStream fs=fi.Create();
fs.Close();
}
try
{
using (StreamReader fileObj = new StreamReader(filename))
{
string datapeople = fileObj.ReadToEnd();
try
{
this.DataPeople = JsonSerializer.Deserialize<List<Person>>(datapeople);
}
catch (Exception)
{
this.DataPeople = new List<Person>();
}
}
}
catch (Exception)
{
this.DataPeople = new List<Person>();
}
}
```

Метод, для збереження нових даних у файл. Спочатку серіалізуємо з відступами у змінну text. Після чого відриваємо файловий потік для зчитування файлу, обов'язково використовуємо директиву using для того, щоб одразу закривати потік. Потім записуємо в файл серіалізовані дані.

```
public void SaveData(List<Person> people, string filename)
{
string text = JsonSerializer.Serialize(people, options);
using (StreamWriter fileObj = new StreamWriter(filename))
{
fileObj.Write(text);
}
}
```

Метод, який шукає задане значення і повертає екземпляр класу, в якому це значення збігається ініціалізуємо новий об'єкт класу, якому присвоюємо елемент списку, містить елементи з вхідної послідовності, які задовольняють умову.

```
public Person SearchByEmail(string email)
```

```
{
Person persona = this.DataPeople.Where(p => p.Email == email).FirstOrDefault();
return persona;
}
```

Метод який , який повертає список осіб, які задовольняють задану умову.

```
public List<Person> FilterBySex(string sex)
{
List<Person> persona = new List<Person>(this.DataPeople.Where(p => p.Sex == sex));
return persona;
}
```

Метод, для додавання об'єкту класу в файл. Спочатку перевіряємо валідність введених даних, коли вони не валідні, то виводимо відповідне повідомлення. Потім якщо у файлі немає особи з заданим значенням електронної адреси, яка виступає як унікальний ідентифікатор, то тоді додаємо об'єкт класу у колекцію, після чого зберігаємо файл та виводимо відповідне повідомлення. Якщо ж така електронна адреса вже є у колекції, то знаходимо індекс цього об’єкту в колекції і на його місце додаємо новий об’єкт, який заданий параметром методу. Зберігаємо нове значення, виводимо відповідне повідомлення. Таким чином ми можемо оновити дані.

```
public string AddPerson(Person person, string filename)
{
string answer;
try
{
if (ValidateData(person) == true)
{
if (SearchByEmail(person.Email) == null)
{
this.DataPeople.Add(person);
SaveData(this.DataPeople, filename);
answer = $"Person {person.FirstName} is succesful added";
}
else
{
int i = this.DataPeople.IndexOf(SearchByEmail(person.Email));
this.DataPeople[i] = person;
SaveData(this.DataPeople, filename);
answer = $"Person {person.FirstName} is succesful updated";
}
}
else
{
answer = $"Check your data!";
}
}
catch (Exception)
{
answer = "Cannot be serialized";
}
return answer;
}
```

Метод для видалення об'єкту з колекції за заданим параметром електронної адреси. Спочатку знаходимо індекс цього об'єкту в колекції. Потім видаляємо та зберігаємо файл.

```
public void DeletePerson(string email, string filename)
{
int i = this.DataPeople.IndexOf(SearchByEmail(email));
this.DataPeople.RemoveAt(i);
SaveData(this.DataPeople, filename);
}
```

Метод для перевірки даних, повертає значення змінної valited. Якщо воно дорівнює false, то дані некоректні

```
public bool ValidateData(Person person)
```

```
{
var results = new List<ValidationResult>();
var context = new ValidationContext(person);
bool validated = false;
if (Validator.TryValidateObject(person, context, results, true))
{
validated = true;
}
return validated;
}
}
```

Код класу Startup:

Код класу Startup:

```
public class Startup
{
```

Глобальна зміна для вказування шляху на потрібний файл

```
public static string filename = "/user/src/app/shared_folder/person.txt";
public static People people = new People(filename);
```

Цей метод використовується для додавання різних сервісів в конвеєр.

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

Цей метод використовується для конфігурації конвеєра запитів HTTP.

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
{
if (env.IsDevelopment())
app.UseDeveloperExceptionPage();
}
```

Метод Map застосовується для зіставлення шляху запиту з певним делегатом, який оброблятиме запит по цьому шляху.

```
app.UseRouting();
app.Map("/ListOfPeople", ListOfPeople);
app.Map("/SearchByEmail", SearchByEmail);
app.Map("/FilterBySex", FilterBySex);
app.Map("/DeletePerson", DeletePerson);
app.Map("/AddPerson", AddPerson);
```

Метод Run додає компоненти middleware в конвеєр, ці компоненти не викликають ніякі інші компоненти і далі обробку запиту не передають.

```
app.Run(async (context) =>
{
await context.Response.WriteAsync("Laboratory work 2");
});
}
```

Делегат, який конфігурує окремий конвеєр проміжного програмного забезпечення. він викликає метод Run, який додає компоненти в конвеєр. Якщо у файлі не буде ніяких даних, то поверне відповідь, про відсутність даних. У іншому випадку поверне колекцію об'єктів у форматі масиву JSON-об’єктів.

```
private static void ListOfPeople(IApplicationBuilder app)
{
app.Run(async context =>
{

if (people.DataPeople.Count != 0)
{
await context.Response.WriteAsJsonAsync(people.DataPeople);
}
else
{

await context.Response.WriteAsJsonAsync(new { Error = "Missing data!" });
```

```
}
});
}
```

Даний делегат записує у змінну email значення параметру запиту, після чого викликається метод SearchByEmail(), якщо у колекції не буде знайдено об'єкта, значення поля Email буде дорівнювати значенню email, то він поверне повідомлення, що такого Email немає у колекції, інакше поверне об'єкт, об'єкт який відповідає умові у JSON-форматі.

```
private static void SearchByEmail(IApplicationBuilder app)
{
app.Run(async context =>
{
string email = context.Request.Query["email"];
Person person = people.SearchByEmail(email);
if (person != null)
{

await context.Response.WriteAsJsonAsync(people.SearchByEmail(email));
}
else
{
await context.Response.WriteAsJsonAsync(new { Error = "Email not found" });
}
});
}
```

Делегат записує значення параметру запиту у змінну sex. Тепер викликаємо метод FilterBySex(), який поверне список об'єктів, які відповідають заданій умові. Якщо у списку немає ні одного елемента, то виводимо повідомлення, що такої статті немає у колекції. Якщо ж є, то тоді виведемо список об'єктів, які задовольняють умову у форматі JSON-масиву.

```
private static void FilterBySex(IApplicationBuilder app)
{
app.Run(async context =>
{

string sex = context.Request.Query["sex"];
List<Person> listperson = people.FilterBySex(sex);
if (listperson.Count != 0)
{
await context.Response.WriteAsJsonAsync(people.FilterBySex(sex));
}
else
{
await context.Response.WriteAsJsonAsync(new { Error = "Sex not found" });
}
});
}
```

Цей делегат, також зчитує та записує у змінну email ключ параметру запиту. Видаляє об'єкт за заданим email, за допомогою метода DeletePerson(). Також у змінну записуємо значення поля FirstName, щоб при виведені повідомлення про видалення, вказати, кого саме видалено.

```
private static void DeletePerson(IApplicationBuilder app)
{
app.Run(async context =>
{
string email = context.Request.Query["email"];
string name = people.SearchByEmail(email).FirstName;
people.DeletePerson(email, filename);
await context.Response.WriteAsJsonAsync(new { Status = $"Person {name} is succesful
deleted" });
});
}
```

Отже, останній делегат, який ми використовуємо для цієї веб-служби спочатку записує у змінну значення тіла запиту. Потім відкриваємо файловий створений на основі цієї змінної. У іншу змінну зчитуємо та записуємо значення тіла запиту. Після цього десеріалізуємо його, додаєм це значення у колекцію та виводимо відповідне повідомлення.

```
private static void AddPerson(IApplicationBuilder app)
{
app.Run(async context =>
{
var request = context.Request.Body;
string body;
using (StreamReader reader = new StreamReader(request))
{
body = await reader.ReadToEndAsync();
}
Person form = JsonSerializer.Deserialize<Person>(body);
string answer = people.AddPerson(form, filename);
await context.Response.WriteAsJsonAsync(new { Status = answer });
});
}
}
```

Після створенні цієї веб-служби перенесемо її на Linux за допомогою протоколу SFTP так само, як показано вище.

Тепер потрібно розгорнути і цю веб-службу у Docker. Для використаємо ті ж команди, що використовувались для минулої лабораторної роботи. Проте, потрібно змінити назви зображень та порт, щоб кілька контейнерів працювало одночасно і ми мали до них доступ.

Команди (перед цим запустіть службу Docker):

```
docker build --tag=l2 .
```

```
docker container run -d -p 5001:80 -v /root/sf:/user/src/app/shared_folder l2
```

Також, що веб-служба була доступна, потрібно переадресувати порт в середині віртуальної машини створивши нове TCP-правило:

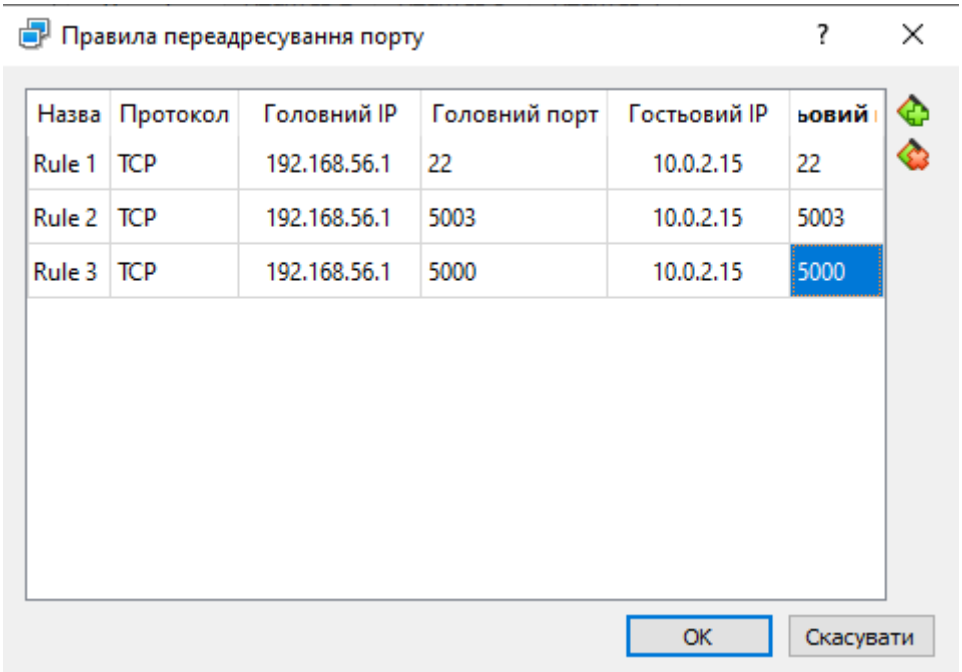

Рис. 35 Переадресація порту

Перевіримо роботу веб-служби.

Перша її функція – це вивід всієї колекції. Для зручності використовуємо ПЗ Postman. Створюємо новий запит, за адресою:

http://192.168.56.1:5000/listofpeople

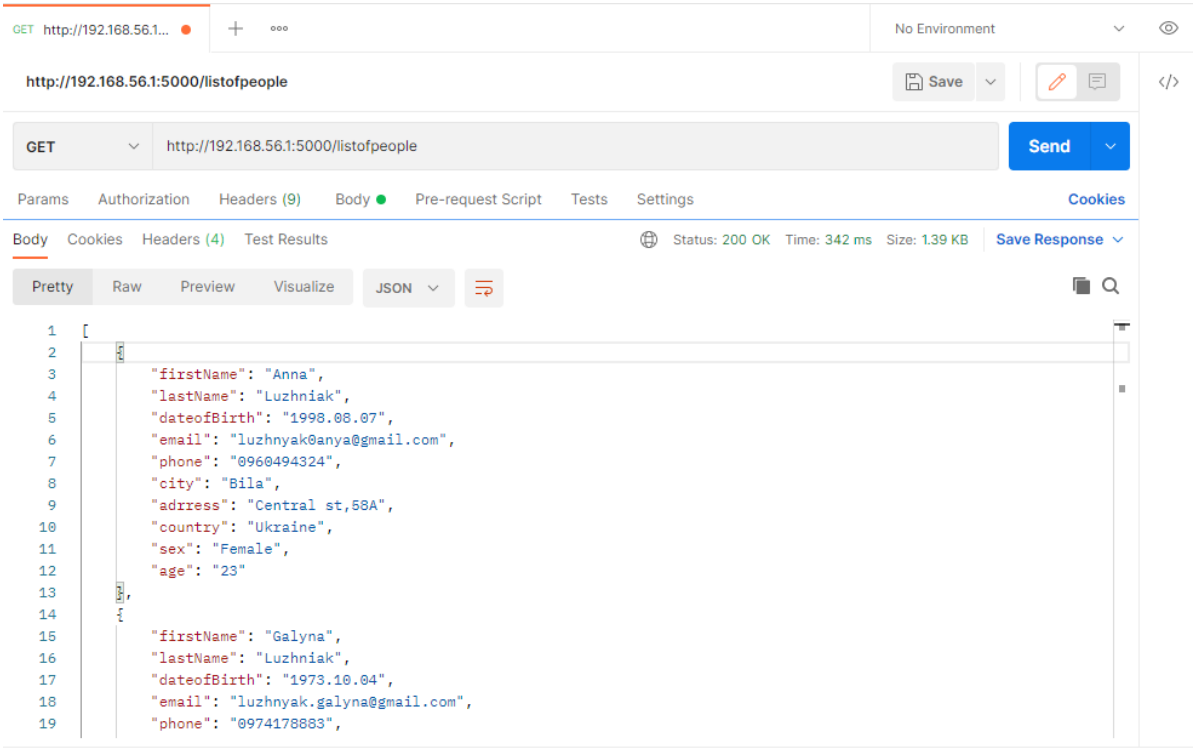

Рис. 36 вивід всієї колекції

Як бачимо відповідь- це список усіх об’єктів у форматі JSON.

Тепер знайдемо об’єкт у колекції, у якого така ж електронна адреса як і значення параметру запиту. Посилання

http://192.168.56.1:5000/searchbyemail?email=pp@gmail.com

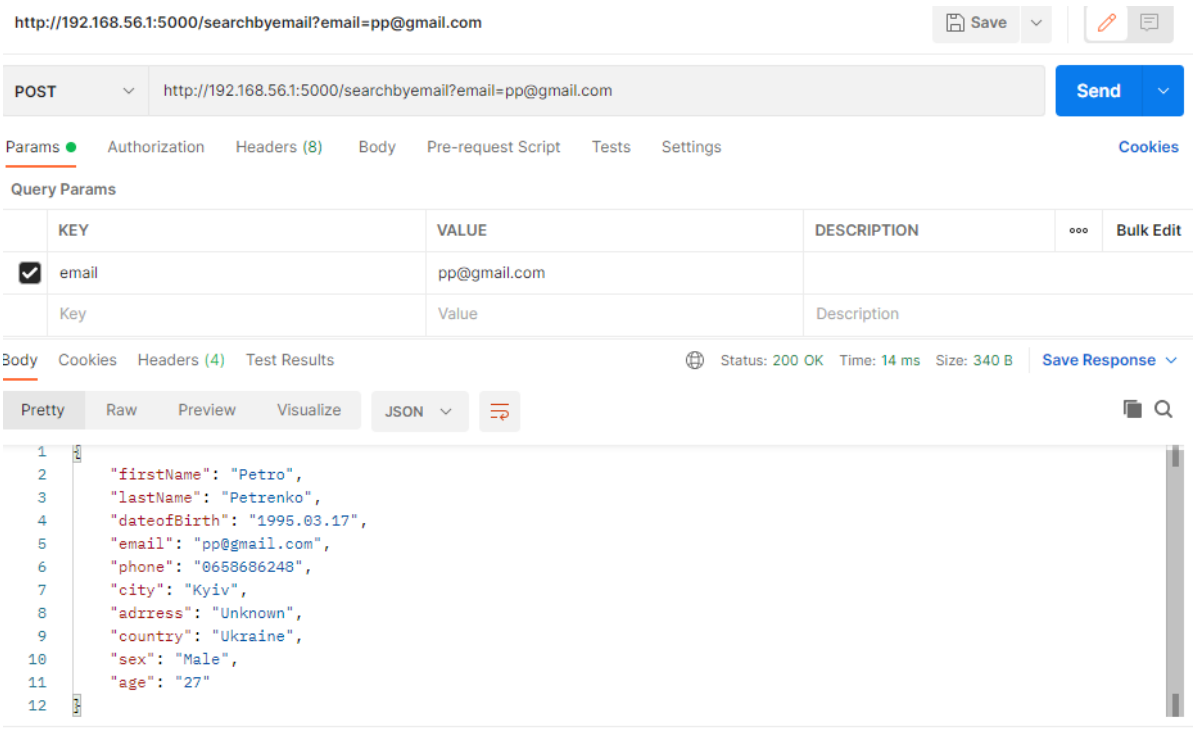

Рис. 37 Пошук за електронною адресою

Відповідь – один об’єкт, який задовільняє умову
Фільтрування за статтю –

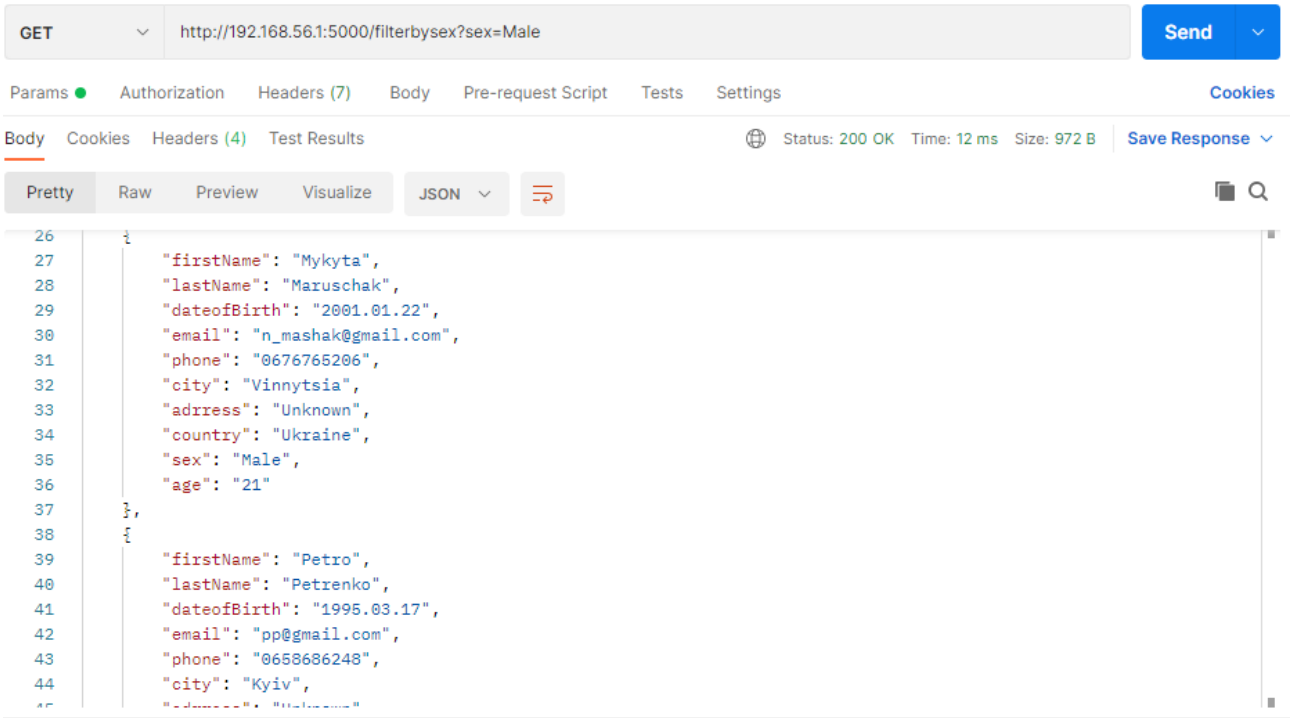

Рис. 38 Фільтрування за статтю

Результат – колекція, у якій значення поля Sex – Male.
Додавання нової особи. Для цього у вкладці Body у форматі JSON пишемо об’єкт, який хочемо додати. Посилання http://192.168.56.1:5000/AddPerson

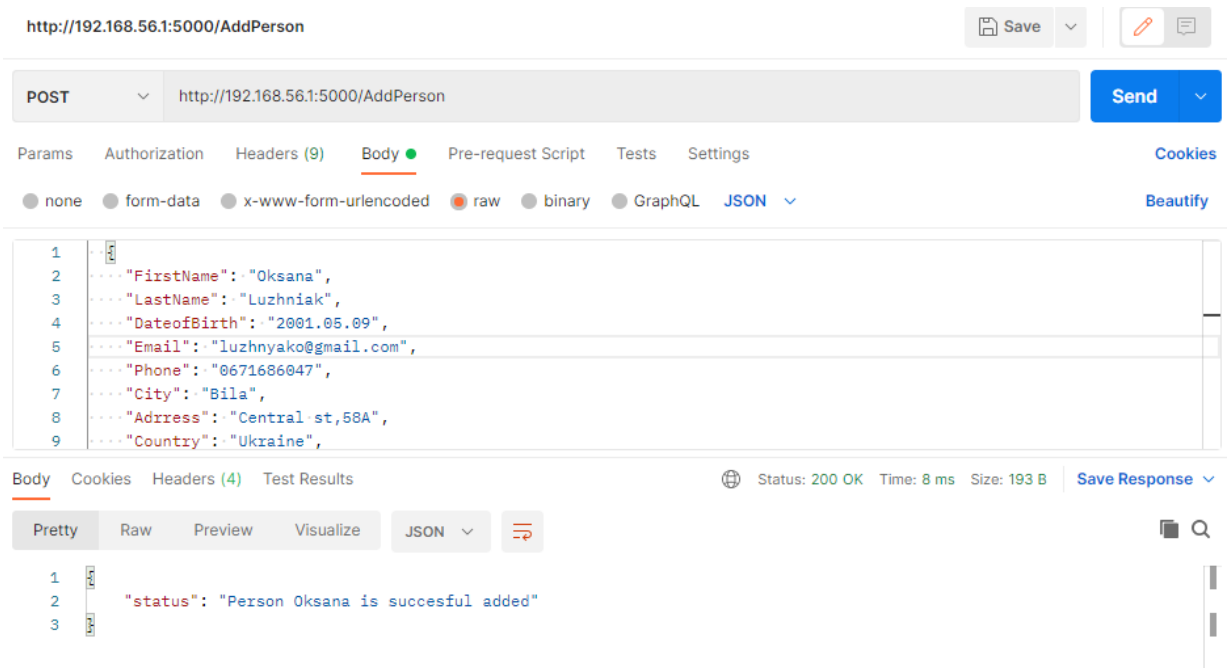

Рис. 39 Додавання нового об'єкту

Видалення особи. Для видалення особи з колекції, у параметрі запиту потрібно вказати електронну адресу цієї особи. Адреса http://192.168.56.1:5000/DeletePerson?email=n_mashak@gmail.com

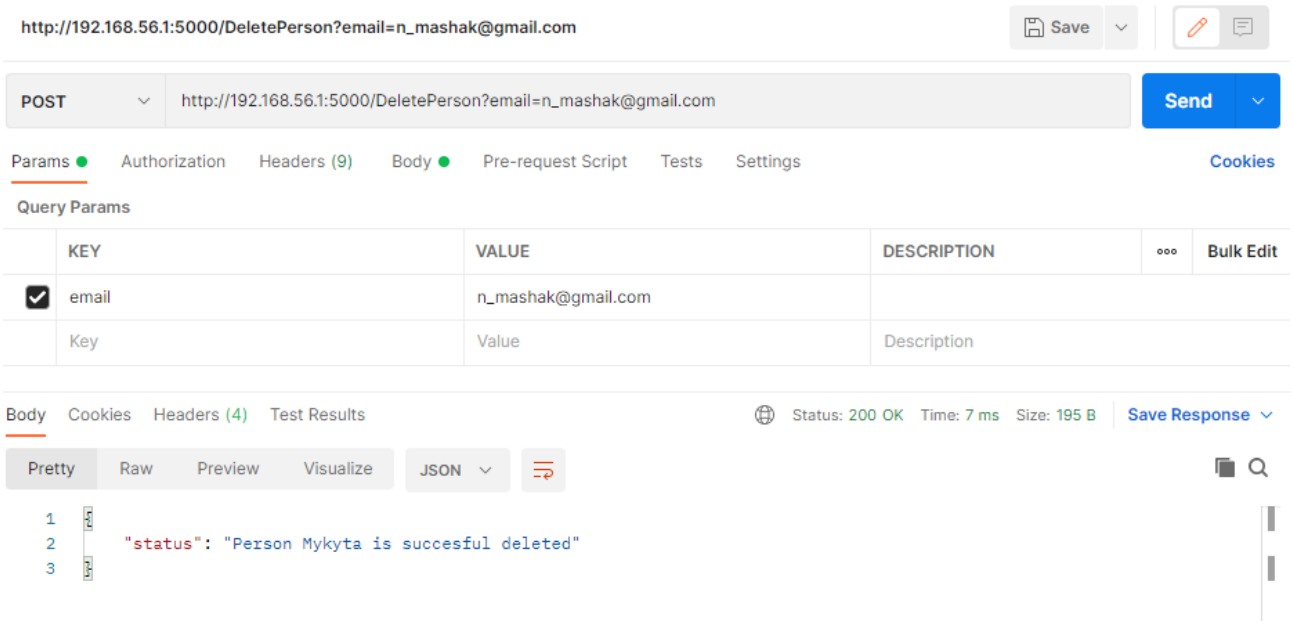

Рис. 40 Видалення об'єкту

Як бачимо всі функції працюють.

2.3 Третя лабораторна робота

2.3.1 Мета роботи третьої лабораторної роботи

Дослідити авторизацію в ASP.NET Core. Створити веб-службу, у якій працює авторизація. Доступність до функцій веб-служби повинна бути надана тільки

авторизованим користувачам. Навчитись розгортати веб-службу в Docker з підкримкою протоколу HTTPS.

2.3.2 Завдання лабораторної роботи

Створити веб-службу, в якій є авторизація. Доступність до функції повинна бути надана лише авторизованим користувачам. Авторизація може бути зроблення за допомогою JWT-токенів/або будь-якого іншого механізму. Початкова сторінка може бути доступна всім користувачам. Авторизований користувач також повинен мати роль. Відповідно до ролі юзеру доступні відповідно по варіантам: Administrator – видалення колекції об’єктів, або об’єкту; User – усі варінти – вивід усієї колекції, фільтрація по певній ознаці/ пошук об’єкту по якомусь полю. Варіанти колекцій з попередньої лабораторної роботи.

2.3.3 Приклад виконання

З основу візьмемо веб-службу з минулої лабораторної роботи. Вона буде виконувати ті ж функції, проте доступні вони будуть лише авторизованим користувачам. З минулої лабораторної не змінився лише код класу Person. Також додамо два нових класи User та Users дл роботи з користувачами. Отже, для реалізації авторизації потрібно використати Asp.Net Core MVC. Моделями будуть два класи Person та User. Також створимо два контролера: один для авторизації, інший виконує всі інші функції. Необхідним є файл credentials.txt, де будуть записані дані, для авторизації користувачів.
Код усіх файлів наведений нижче:
Контролер, який відповідає за реагування веб-службою на запити після авторизації.

```
public class APIController : Controller
{

private readonly People _people;

public APIController(People people)
{
_people = people;
}
```

Головна сторінка, яка при відсутності будь-яких помилок, повертає надпис.

```
[Route("/")]
public IActionResult Index()
{
return Ok("Laboratory work 3");
}
```

Якщо користувач, авторизований, то даний запит, якщо у файлі є данні, повертає колекцію об'єктів. У іншому випадку повертає повідомлення про відсутність таких даних.

```
[Authorize]
[Route("/listofpeople")]
public IActionResult ListOfPeople()
{
```

```
if (_people != null)
{
return Json(_people.DataPeople);
}
else
{
return Ok("Missing data");
}
}
```

Авторизований користувач відправляє на веб-службу GET-запит з параметром та його значення. Зчитуємо у змінну це значення, після чого викликаємо метод для пошуку email в колекції. Якщо нічого не відповідає умові, то виводимо про це повідомлення, інакше - виводимо той об'єкт, який задовольняє умову у форматі JSON.

```
[Authorize]
[HttpGet("/searchbyemail")]
public IActionResult SearchByEmail()
{
string email = Request.Query["email"];
Person person = _people.SearchByEmail(email);
if (person != null)
{

return Json(person);
}
else
{
return Json(new { Error = "Email not found" });
}
}
```

Надсилаємо GET-запит та параметр, за яким ми будемо фільтрувати. Записуємо його у змінну. Потім викликаємо метод для фільтрації. Після чого перевіряємо кількість у відфільтрованій колекції. Якщо вона не рівна нулю, то виводимо колекцію. Якщо ж кількість дорівнює нулю, тобто в списку нічого немає, виводимо про це повідомлення

```
[Authorize]
[HttpGet("/filterbysex")]
public IActionResult FilterBySex()
{
string sex = Request.Query["sex"];
List<Person> listperson = _people.FilterBySex(sex);
if (listperson.Count != 0)
{
return Json(listperson);
}
else
{
return Json(new { Error = "Sex not found" });
}
}
```

Надсилаємо DELETE-запит, з параметром - електронною адресою, яка є ідентифікатором. За цим параметром ми будемо видаляти з колекції об'єкти. Отже, зчитуємо у змінну значення параметру. а потім у змінну name записуємо значення поля FirstName, об'єкту який задовольняє умову, тобто викликаємо метод SearchByEmail та за його допомогою знаходим об'єкт з такою електронною адресою. Після чого видаляємо об'єкт з колекції за допомогою метода DeletePerson. Та виводимо повідомлення, що такий об'єкт видалено.

```
[Authorize]
[HttpDelete("/deleteperson")]
public IActionResult DeletePerson()
{
string email = Request.Query["email"];
```

```
string name = _people.SearchByEmail(email).FirstName;
_people.DeletePerson(email);
return Json(new { Status = $"Person {name} is succesful deleted" });
}
```

Надсилаємо POST-запит, у тіло якого записуємо колекцію даних, які хочемо додати. Потім записуємо значення з тіла у змінну request. Далі відкриваємо потік для записування, після чого змінній body присвоюємо значення, яке зчитаємо з потоку. Після цього десеріалізуємо у список осіб form, після цього визначаємо кількість екземплярів у списку. Тепер ініціалізуємо змінну countnames для визначення кількості доданих екземплярів. Потім запускаємо цикл, від 0 до кількості доданих елементів і спочатку додаємо нульовий елемент списку і інкрементуємо countnames, після чого рухаємось далі, поки не додамо всі елементи списку. Після додавання виводимо повідомлення про те, скільки елементів додано.

```
[Authorize]
[HttpPost("/addperson")]
public async Task<IActionResult> AddPerson()
{

var request = Request.Body;
string body;
using (StreamReader reader = new StreamReader(request))
{
body = await reader.ReadToEndAsync();
}
List<Person> form = JsonSerializer.Deserialize<List<Person>>(body);
int count = form.Count;
int countnames=0;
for (int i = 0; i < count; i++)
{

bool ans=_people.AddPerson(form[i]);
if (ans==true)
{
countnames++;
}
}
return Json(new { Status = $"Added {countnames} people" });

}
}
```

Контролер, який відповідає на запит авторизації.

```
public class AuthController : Controller
{
private readonly Users _users;
public AuthController(Users users)
{
_users = users;
}
```

Метод, для формування токена. У змінні now записано час, в який створюється токен. Створюємо набір об'єктів claims, який містить інформацію про користувача, в даному випадку його логін. Сам токен - це об'єкт JwtSecurityToken, для ініціалізації якого застосовуються ті самі константи та ключ безпеки, які визначені в класі AuthOptions і які використовувалися в класі Startup для налаштування JwtBearerAuthenticationMiddleware. В кінці повертаємо значення токена, конвертуючи його в тип string.

```
public static string GetToken(string username)
{
var now = DateTime.UtcNow;
var claims = new List<Claim> { new Claim(ClaimTypes.Name, username) };
var jwt = new JwtSecurityToken(
```

```
issuer: AuthOptions.ISSUER,
audience: AuthOptions.AUDIENCE,
notBefore: now,
claims: claims,
expires: now.Add(TimeSpan.FromMinutes(AuthOptions.LIFETIME)),
signingCredentials: new SigningCredentials(AuthOptions.GetSymmetricSecurityKey(),
SecurityAlgorithms.HmacSha256));
var encodedJwt = new JwtSecurityTokenHandler().WriteToken(jwt);

var response = new
{
access_token = encodedJwt,
};
return response.ToString();
}
```

Користувач відправляє на веб-службу GET-запит з параметрами логіном та паролем. Потім зчитуємо значення цих параметрів у відповідні змінні. Після чого перевіряємо за допомогою двох методів чи такі дані є допустимими. Тобто, чи вони існують у файлі з креденшеналами. А також обов'язково перевіряємо чи даному логіну відповідає заданий пароль. Якщо всі умови виконані, то повертаємо результат роботи метода GetToken(), тобто токен для такого користувача, у форматі JSON.

```
[HttpGet("/token")]
public IActionResult Token(string user, string password)
{
user = Request.Query["login"];
password = Request.Query["password"];

if (_users.CheckLogin(user) != null && _users.CheckPassword(password) != null)
{
if (user == _users.CheckLogin(user).Login & password == _users.CheckLogin(user).Password)
{
return Json(JsonSerializer.Serialize(GetToken(user)));
}
else
{
return Json(new { Error = "Credentials is invalid" });
}
}
else
{
return Json(new { Error = "Credentials is empty" });
}
}
}
```

Цей клас - це набір певних властивостей, які визначають якусь особу. В блоці get властивості Age, виконуються дії розрахунку віку кожної особи для отримання значення цієї властивості. Також є атрибут EmailAddress, для перевірки коректності введеної електронної адреси, а також RegularExpression для перевірки коректності номеру телефону.

```
public class Person
{
public string FirstName { get; set; }
public string LastName { get; set; }
public string DateofBirth { get; set; }
[EmailAddress(ErrorMessage = "Invalid Email Address")]
public string Email { get; set; }
[RegularExpression(@"^([0-9]{10})$", ErrorMessage = "Invalid Phone Number.")]
public string Phone { get; set; }
public string City { get; set; }
public string Adrress { get; set; }
public string Country { get; set; }
public string Sex { get; set; }
public string Age
```

```
{
get

{
var dateTime = DateTime.Parse(this.DateofBirth);
int age = (DateTime.Today.Year - dateTime.Year);
if (DateTime.Today.DayOfYear < dateTime.DayOfYear)
{
age--;
}
return age.ToString();
}
}
}
```

Клас User використовується для задання властивостей користувача для авторизації.

```
public class User
{
public string Login { get; set; }
public string Password { get; set; }
}
```

Клас AuthOptions описує властивості, які потрібні для генерації токену. Константа ISSUER - це видавець токена. Сюди можна написати будь-яке значення. AUDIENCE представляє того, хто буде використовувати токен. Значення також може бути будь-яким. Константа KEY зберігає ключ, який буде застосовуватися для створення токена. Константа LIFETIME - це час життя токену.

```
public class AuthOptions
{
public const string ISSUER = "MyServer";
public const string AUDIENCE = "MyClient";
const string KEY = "mysupersecret_secretkey!123";
public const int LIFETIME = 60;
public static SymmetricSecurityKey GetSymmetricSecurityKey()
{
return new SymmetricSecurityKey(Encoding.ASCII.GetBytes(KEY));
}
```

Даний клас взаємодіє з класом Person.

```
public class People
{
static JsonSerializerOptions options = new JsonSerializerOptions { WriteIndented = true
};
static string filename = "/root/l3/person.txt";

public List<Person> DataPeople { get; set; }
```

Конструктор класу.

```
public People()
{
LoadData(filename);
}
```

Метод, який викликається в конструкторі. Він використовується для завантаження даних. Спочатку створюємо екземпляр класу FileInfo над шляхом, який передаємо в параметрах. Потім якщо за таким шляхом, такого файлу не існує, то створюємо його. Після чого відкриваємо файловий потік. У змінну записуємо значення, яке зчитуємо з файлу. Після чого десеріалізуємо його та записуємо у властивість DataPeople. При виникненні виключень записуємо у властивість пустий список.

```
public void LoadData(string filename)
{
FileInfo fi = new FileInfo(filename);
```

```
if (!fi.Exists)
{
FileStream fs = fi.Create();
fs.Close();
}
using (StreamReader fileObj = new StreamReader(filename))
{
string datapeople = fileObj.ReadToEnd();
try
{
this.DataPeople = JsonSerializer.Deserialize<List<Person>>(datapeople);
}
catch (Exception)
{

this.DataPeople = new List<Person>();
}
}
}
```

Метод для збереження будь-яких змін. Серіалізуємо список, який передаємо в параметрах. Після цього відкриваємо файловий потік і записуємо у файл наше серіалізоване значення.

```
public void SaveData(List<Person> people)
{
string text = JsonSerializer.Serialize(people, options);
using (StreamWriter fileObj = new StreamWriter(filename))
{
fileObj.Write(text);
}
}
```

Метод для пошуку серед колекції певного екземпляру, який відповідає певному критерію. А саме в даному випадку повертаємо об'єкт класу, у кому властивість Email дорівнює значенню, яке задане у параметрах методу.

```
public Person SearchByEmail(string email)
{
Person persona = this.DataPeople.Where(p => p.Email == email).FirstOrDefault();
return persona;
}
```

Даний метод використовується для фільтрації списку за певною умовою. Він повертає колекцію об'єктів, у якій поле Sex дорівнює значенню у параметрах.

```
public List<Person> FilterBySex(string sex)
{
List<Person> persona = new List<Person>(this.DataPeople.Where(p => p.Sex == sex));
return persona;
}
```

Метод AddPerson додає об'єкт у колекцію. Спочатку перевіряємо валідність даних. Потім перевіряємо чи вже такий об'єкт у колекції. Для цього викликаємо метод SearchByEmail(),якщо він повертає null, тобто об'єкту з такою електронною адресою немає у колекції, то додаємо його. Інакше визначаємо індекс цього об'єкту у колекції та додаємо наш новий об'єкт на нове місце, таким чином ми оновлюємо дані. В кінці викликається метод SaveData() та зберігає зміни.

```
public bool AddPerson(Person person)
{
bool answer=true;
if (ValidatePhone(person) == true)
{
if (SearchByEmail(person.Email) == null)
{
this.DataPeople.Add(person);
SaveData(this.DataPeople);
}
```

```
else
{
int j = this.DataPeople.IndexOf(SearchByEmail(person.Email));
this.DataPeople[j] = person;
SaveData(this.DataPeople);
}
return answer;
}
else { return answer = false; }
}
```

Для видалення об'єкту використовується метод DeletePerson. В параметрах передаємо електронну адресу об'єкта, якого хочемо видалити. Потім знаходимо індекс цього об'єкту. Та видаляємо за цим індексом. Після чого зберігаємо.

```
public void DeletePerson(string email)
{
int i = this.DataPeople.IndexOf(SearchByEmail(email));
this.DataPeople.RemoveAt(i);
SaveData(this.DataPeople);
}
```

Цей метод використовується для перевірки коректності властивостей Email та Phone. Коли він повертає false, то це означає, що перевірка не пройдена, дані некоректні.

```
public bool ValidatePhone(Person person)
{
var results = new List<ValidationResult>();
var context = new ValidationContext(person);
bool validated = false;
if (Validator.TryValidateObject(person, context, results, true))
{
validated = true;
}
return validated;
}
}
```

Даний клас використовується для взаємодії з класом User.

```
public class Users
{
static JsonSerializerOptions options = new JsonSerializerOptions { WriteIndented = true
};
static string filename = "/root/l3/credentials.txt";
public List<User> DataUsers { get; set; }
```

Конструктор класу

```
public Users()
{
LoadUsers(filename);
}
```

Працює так само як метод People.LoadData().

```
public void LoadUsers(string filename)
{
FileInfo fi = new FileInfo(filename);
if (!fi.Exists)
{
FileStream fs = fi.Create();
fs.Close();
}
using (StreamReader fileObj = new StreamReader(filename))
{
string datausers = fileObj.ReadToEnd();
try
{
this.DataUsers= JsonSerializer.Deserialize<List<User>>(datausers);
}
catch (Exception)
{
```

```
this.DataUsers = new List<User>();
}
}
}
```

Принцип роботи описано вище для People.SaveData().

```
public void SaveUsers(List<User> user)
{
string text = JsonSerializer.Serialize(user, options);
using (StreamWriter fileObj = new StreamWriter(filename))
{
fileObj.Write(text);
}
}
```

Дані методи використовується для перевірки наявності логіна та пароля у файлі з креденшеналами. Повертають об'єкти, які задовольняють умови.

```
public User CheckLogin(string login)
{
User user= this.DataUsers.Where(u => u.Login == login).FirstOrDefault();

return user;
}
public User CheckPassword(string password)
{
User user = this.DataUsers.Where(u => u.Password == password).FirstOrDefault();
return user;
}
}
```

Метод ConfigureServices() використовується для додавання різних сервісів та компонентів у конвеєр. Для установки автентифікації за допомогою маркерів у методі ConfigureServices при виклику services.AddAuthentication передається значення JwtBearerDefaults.AuthenticationScheme. Далі за допомогою методу AddJwtBearer() додається конфігураційна токена. Також додамо класи People і Users за допомогою метода AddSingleton() при цьому об'єкт сервісу створюється при першому зверненні до нього, усі кінцеві запити використовують один і той же раніше створений об'єкт сервісу.

```
public void ConfigureServices(IServiceCollection services)
{
services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
 .AddJwtBearer(options =>
 {
 options.RequireHttpsMetadata = true;
 options.TokenValidationParameters = new TokenValidationParameters
 {
 ValidateIssuer = true,
 ValidIssuer = AuthOptions.ISSUER,
 ValidateAudience = true,
 ValidAudience = AuthOptions.AUDIENCE,
 ValidateLifetime = true,
 IssuerSigningKey = AuthOptions.GetSymmetricSecurityKey(),
 ValidateIssuerSigningKey = true,
 };
 });
services.AddSingleton<People>();
services.AddSingleton<Users>();
services.AddControllersWithViews();
}
```

Даний метод, використовується щоб налаштувати конвеєр запитів HTTP.

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
if (env.IsDevelopment())
{
app.UseDeveloperExceptionPage();
```

```
}
app.UseRouting();
app.UseAuthentication();
app.UseAuthorization();
app.UseEndpoints(endpoints =>
{
endpoints.MapDefaultControllerRoute();
});
}
}
```

Тепер потрібно створити зображення. Команда для його створення. Як пам’ятаємо створювати його потрібно, там де знаходиться Dockerfile

```
docker build --tag=l3 .
```

Після написання коду ам потрібно розгорнути веб-службу з підтримкою HTTPS-пртоколу.

Для цього спочатку потрібно створити HTTPS-сертифікат. Для цього потрібно виконати команду:

```
dotnet dev-certs https -ep root/.aspnet/https/aspnetapp.pfx –p Тут вказати свій пароль
```

Тепер у папці з проектом створимо новий файл `docker-compose.debug.yml`.

Зміст файлу:

```
version: '3.4'
services:
  webapp:
    image: l3
    ports:
      - 80
      - 443
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - ASPNETCORE_URLS=https://+:443;http://+:80
      - ASPNETCORE_Kestrel__Certificates__Default__Password=123
      - ASPNETCORE_Kestrel__Certificates__Default__Path=/https/aspnetapp.pfx
    volumes:
      - ~/.aspnet/https:/https:ro
      - ~/l3/:/root/l3
```

Також потрібно проінсталювати Docker-Compose. Команди, які потрібно виконати:

```
sudo curl -L "https://github.com/docker/compose/releases/download/1.23.2/docker-compose-
$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
sudo chmod +x /usr/local/bin/docker-compose
```

Перевіримо версію встановленого Docker-Compose

```
docker-compose –version
```

Тепер створимо контейнер за допомогою Docker-Compose

```
docker-compose -f "docker-compose.debug.yml" up -d
```

Виведемо список усіх контейнерів:

```
[root@localhost L3]# docker container ls -a
CONTAINER ID   IMAGE         COMMAND           CREATED          STATUS                       PORTS
          NAMES
7f5f10b5f796   l3            "dotnet L3.dll"   36 seconds ago   Up 34 seconds                0.0.0.0:49156->80/tcp, :::49156->80/tcp, 0.0.0.0:49155->443/tcp, :::491
55->443/tcp   l3_webapp_1
f0a6131bdb5a   l2            "dotnet L2.dll"   23 hours ago     Exited (255) 35 minutes ago  443/tcp, 0.0.0.0:5000->80/tcp, :::5000->80/tcp
          practical_pascal
458ec034b14e   l1            "dotnet L1.dll"   26 hours ago     Exited (255) About an hour ago  443/tcp, 0.0.0.0:5003->80/tcp, :::5003->80/tcp
          nifty_liskov
4f2ec4d27fbd   hello-world   "/hello"          6 months ago     Exited (0) 6 months ago
          practical_poitras
```

Рис. 41 Список всіх контейнерів

Як бачимо, контейнер успішно створився. Після цього потрібно зробити переадресування двох портів, що вказані у колонці PORTS.

Тепер всередині контейнера потрібно створити папку та перемістити туди робочі файли. Перелік необхідних команд

`docker exec -i -t id-контейнера mkdir -p /root/l3` - створення нової папки

`docker cp /root/sf/person.txt id-контейнера:/root/l3`

`docker cp /root/sf/credentials.txt id-контейнера:/root/l3` – перенесення файлів

Після цих всіх налаштувань наша служба працює.

Покажемо її роботу. Спробуємо виконати запит

https://192.168.56.1:49155/listofpeople

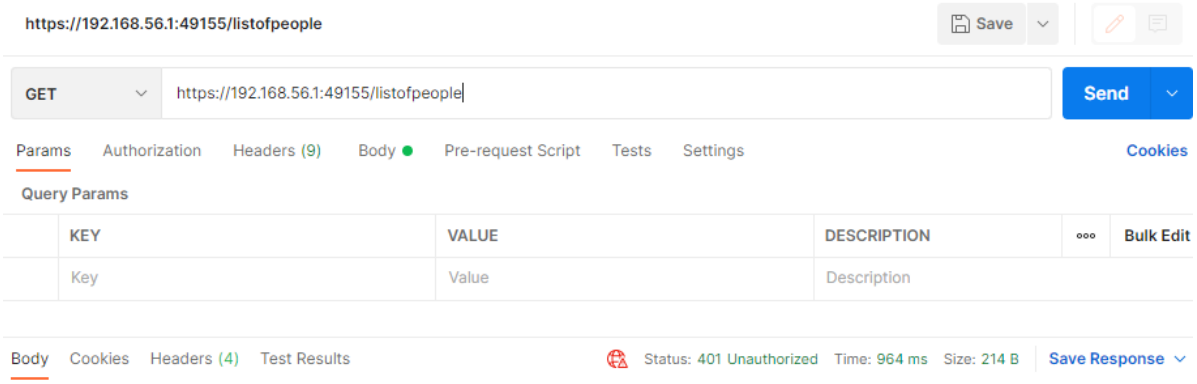

Рис. 42 Перевірка роботи

Як бачимо, ми отримуємо 401 код HTTP, що означає, що ми не авторизовані.

Для авторизації вводимо адресу

https://192.168.56.1:49155/token?login=Oksana&password=123

Де в параметрах вказуємо дані, за якими хочемо авторизуватись. Слід зазначити, що якщо такого логіну не буде у файлі credentials.txt, токен не отримаємо, також він не згенерується, якщо пароль не відповідає логіну.

В результаті ми отримуємо токен доступу:

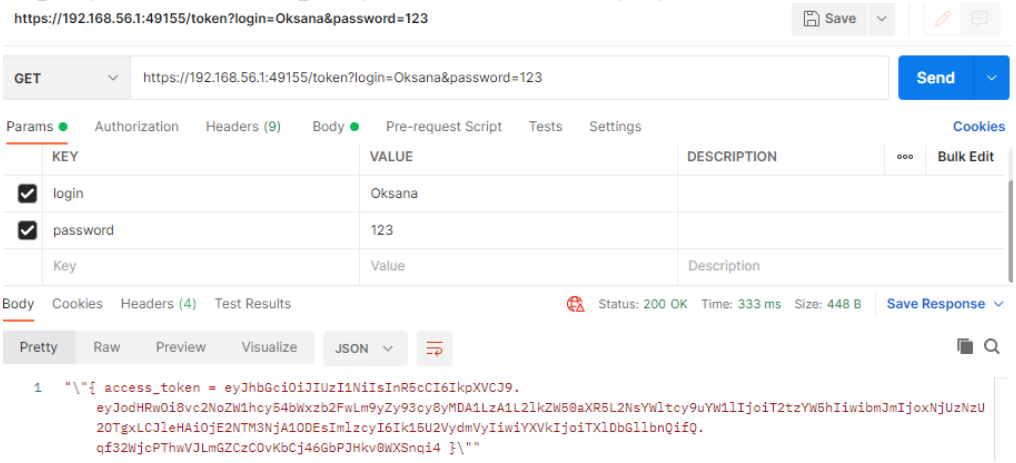

Рис. 43 Отримання токену

Для того щоб працювала авторизація, потрібно додати цей токен у заголовок. Перейдемо на вкладку Headers, в полі Key пишемо Autorization, в полі Value спочатку пишемо ключове слово Bearer і потім через пробіл токен доступу:

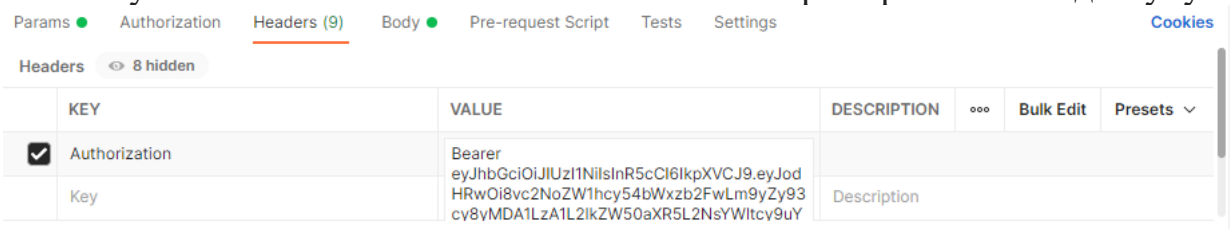

Рис. 44 Заголовок запиту

Надішлемо знову https://192.168.56.1:49155/listofpeople разом з токеном у заголовку

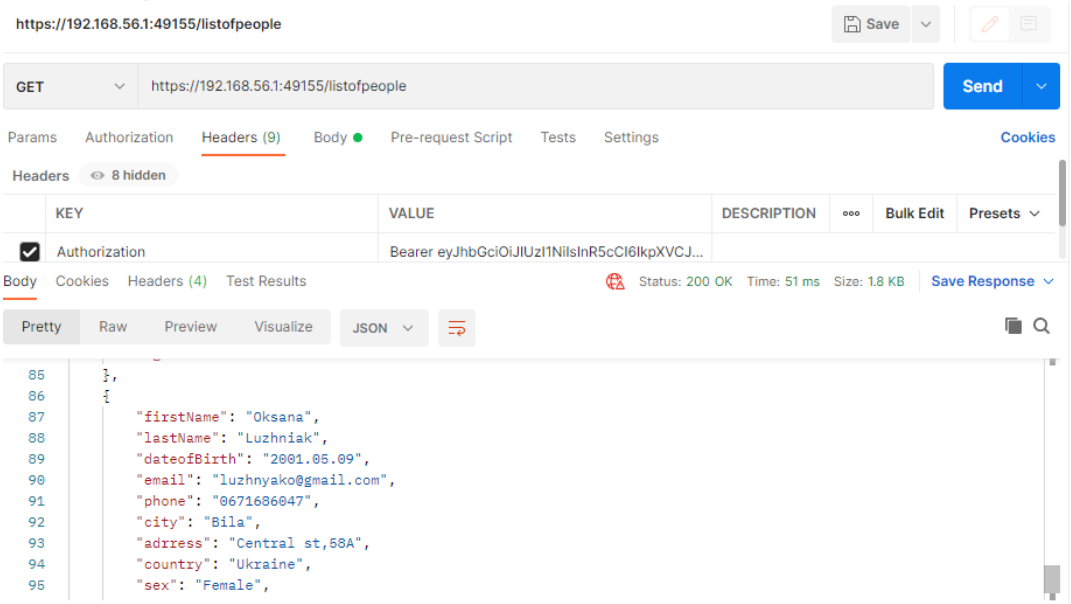

Рис. 45 Перевірка запиту з авторизацією у заголовку

На рисунку 45 видно, що веб-служба повернула код відповіді 200 OK і JSON-файл з переліком об’єктів, а це означає, що служба працює правильно.

ВИСНОВКИ

Створений факультативний курс лабораторних робіт дозволяє студентами ознайомитись або розширити знання з технології контейнеризації прикладних програм. Зокрема, показано, що контейнер може використовувати в 5-10 разів менше оперативної пам’яті, ніж віртуальна машина з працюючою програмою аналогічної функціональності, а отже дозволяє більш раціонально використовувати апаратні ресурси обчислювальної техніки.

Перша лабораторна робота демонструє, що для повної перевірки функціональності веб-служби не обов’язково розробляти для неї програмний клієнт, а достатньо сформувати і направити на веб-службу специфічний веб-запит за протоколом HTTP. При цьому показано, що веб-браузери не містять вбудованих функцій, що дозволяють виконати такий запит. Але такий запит дозволяють виконати низькорівневі HTTP-клієнти. Прикладом такого HTTP-клієнта, використаного в лабораторній роботі, є популярна програма Postman. Наведено приклад, що виконати REST-запити та SOAP-запити можна довільною мовою програмування. Зокрема в мові C# для цього передбачені стандартні .NET-об’єкти WebClient та HttpWebRequest.

Друга лабораторна робота демонструє що структура веб-запиту до веб-служби стилю REST є набагато компактніша ніж до веб-служби стилю SOAP (Simple Object Access Protocol). В першому випадку веб-запит може взагалі не містити поля Body або містити в цьому полі нескладний контент формату JSON. В другому випадку поле Body обов’язково повинно мати складний контент формату XML.

Третя лабораторна робота узагальнює, що веб-служба може бути налаштована як анонімний ресурс або як авторизований ресурс. Для авторизованих ресурсів в роботі висвітлено спеціальний механізм авторизації JWT (JSON Web Token), принцип якого є схожим до використання файлів Cookie, що формують веб-браузери при успішному вході клієнта на авторизований веб-сайт.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Підручник. Контейнеризація програми [Електронний ресурс]. URL:https://docs.microsoft.com/ru-ru/dotnet/core/docker/build-container?tabs=linux
2. What Are RESTful Web Services? [Електронний ресурс]. URL:https://docs.oracle.com/javaee/6/tutorial/doc/gijqy.html
3. What is Docker? [Електронний ресурс]. URL : https://www.docker.com/resources/what-container
4. Call a Web API From a .Net Client (C#) [Електронний ресурс]. URL: https://docs.microsoft.com/uk-ua/aspnet/web-api/overview/advanced/calling-a-web-api-from-a-net-client
5. HTTP Message Handlers in ASP.NET Web API [Електронний ресурс]. URL: https://docs.microsoft.com/uk-ua/aspnet/web-api/overview/advanced/http-message-handlers
6. ASP.NET MVC vs ASP.NET Core MVC – What to Choose for Web Development? https://www.abtosoftware.com/blog/why-use-asp-net-mvc-framework-for-web-application-development
7. JSON Web Tokens [Електронний ресурс]. URL: https://auth0.com/docs/secure/tokens/json-web-tokens
8. JWT in ASP.NET Core [Електронний ресурс]. URL: https://www.c-sharpcorner.com/article/jwt-json-web-token-authentication-in-asp-net-core/