МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ДНІПРОВСЬКА ПОЛІТЕХНІКА»

І.М. Гаркуша

ТЕХНОЛОГІЇ DEVOPS

Методичні рекомендації до виконання лабораторних робіт
для здобувачів ступеня магістра освітньо-професійної програми
«Інформаційні системи та технології»
спеціальності F6 Інформаційні системи і технології

Дніпро
НТУ «ДП»
2025

Гаркуша І. М.

Технології DevOps [Електронний ресурс] : методичні рекомендації до виконання лабораторних робіт для здобувачів ступеня магістра освітньо-професійної програми «Інформаційні системи та технології» спеціальності F6 Інформаційні системи і технології / І. М. Гаркуша ; М-во освіти і науки України, Нац. техн. ун-т «Дніпровська політехніка». – Дніпро : НТУ «ДП», 2025. – 59 с.

Автор
І. М. Гаркуша, канд. техн. наук, доц.

Затверджено науково-методичною комісією спеціальності F6 «Інформаційні системи і технології» (протокол № 9 від 29.08.2025) за поданням кафедри інформаційних технологій та комп’ютерної інженерії (протокол № 1 від 29.08.2025).

Надані теоретичні відомості та практичні завдання, а також рекомендації щодо їх виконання. Поданий перелік рекомендованих джерел.

Орієнтовано на активізацію навчальної діяльності здобувачів ступеня магістра спеціальності F6 Інформаційні системи і технології, закріплення практичних навичок у засвоєнні дисципліни «Технології DevOps».

Відповідальний за випуск завідувач кафедри інформаційних технологій та комп’ютерної інженерії В. В. Гнатушенко, д-р техн. наук, проф.

ЗМІСТ

ВСТУП

Термін *DevOps* є акронімом від слів *Development* (розвиток, розробка) та *Operations* (операції, IT-операції, експлуатація). Виникнення цього терміну приписують Ендрю Клею Шаферу (Andrew Clay Shafer) та Патрику Дебуа (Patrick Debois) в 2007-2008 роках. Вважається що цей термін офіційно прозвучав під час обговорень на конференції Agile у канадському місці Торонто в 2008 році. У 2009 році Патрик Дебуа організував першу конференцію DevOpsDays у бельгійському місці Ґент. DevOpsDays проводяться регулярно протягом кожного року у багатьох країнах світу й у тому числі в Україні.

DevOps-інженери планують, розробляють та підтримують процеси автоматизації розробки, тестування та впровадження програмного забезпечення (ПЗ). Вони залучають різноманітні практики, що спираються на велику множину різноманітних інформаційних технологій, мов програмування та інструментів. Також DevOps-інженери залучають певну філософію щодо управління розробкою та експлуатацією ПЗ, використовують тісне спілкування та культуру організації міжкомандної взаємодії для ефективного, гнучкого та швидкого вирішення певних задач протягом життєвого циклу ПЗ.

У практиках DevOps найчастіше використовуються скриптові мови різноманітних командних процесорів (наприклад: Bash, Windows PowerShell), мови програмування (наприклад: Python, Go та ін.), інструменти контейнеризації (наприклад, Docker), платформи управління репозиторіями проєктів, які мають спеціалізовані можливості щодо безперервної інтеграції та доставлення коду (наприклад: GitLab, GitHub), спеціалізовані інструменти, такі як Jenkins, Puppet та багато інших.

Ці методичні рекомендації до виконання лабораторних робіт є частиною курсу “Технології DevOps”, що викладається в Національному технічному університеті “Дніпровська політехніка” здобувачам спеціальності F6 Інформаційні системи і технології (126 Інформаційні системи та технології) в першому семестрі першого курсу навчання за другим (магістерським) рівнем вищої освіти.

В представлених методичних рекомендаціях розглядаються найвідоміші основні інструменти та технології, що використовуються у практиках DevOps. Активно залучена технологія віртуалізації. Повна множина ПЗ, яка запропонована для розгляду у методичних рекомендаціях, представлена у додатку Б. На рис. Б.1 мовою UML (Unified Modeling Language) показана діаграма розгортання (deployment view) всіх складових мережевих вузлів після завершення виконання робіт. Таким чином, здобувач може подивитися та визначити наскільки виконані практичні роботи відповідають кінцевому результату.

Робота №1.
Використання Bash та Python при автоматизації задач

Мета роботи: автоматизувати розгортання та налаштування Web-серверу в оточені Ubuntu Server.

Теоретично-практична частина

Встановимо Ubuntu Server на віртуальну машину, використовуючи один з iso-образів серверів ресурсу https://releases.ubuntu.com/ (наприклад, LTS-версію 24.04.x). Віртуальну машину розгортати або в оточенні Oracle VirtualBox (під MS Windows, або під GNU/Linux-сумісні), або в Hyper-V (під MS Windows), або в UTM у режимі використання гіпервізору хосту (під Apple macOS).

При встановлені залишити в якості мови English, обрати встановлення варіанту Ubuntu Server, обрати встановлення на диск без підтримки LVM group (тобто повинно бути два розділи за замовчанням: BIOS grub spacer та ext4 – буде змонтований в корінь дерева файлової системи). В якості імені користувача, його пароля, а також імені серверу задати `stud`. При встановленні обрати підтримку OpenSSH без обрання додаткових компонент серверу. Для економії часу оновлення можна не встановлювати.

Типовий вигляд екрану після логіну в ОС Ubuntu Server на прикладі версії 24.04.2 представлений на рис. 1.1 (кольорову схему змінено).

Зверніть увагу, що якщо мережа налаштована вірно і за замовченням увімкнений режим отримання IP-адреси за протоколом DHCP, то у вікні входу на сервер буде відображатися ця IP-адреса.

Якщо в процесі роботи із сервером потрібно з'ясувати IP-адресу, то виконують команду:

```
ip addr
```

Однією із найвідоміших задач, яку виконує DevOps, може бути розгортання на сервері в хмарі Web-серверу. В хмарних оточеннях найчастіше використовують Debian-подібні ОС які добре себе зарекомендували. Наприклад, Ubuntu Server або власно Debian GNU/Linux.

Розглянемо процес встановлення та налаштування Web-серверу *Lighttpd*.

Перевірити наявність в системі встановленого пакету *lighttpd* можна за допомогою, наприклад, команд:

```
which lighttpd
```

або

```
dpkg -l | grep lighttpd
```

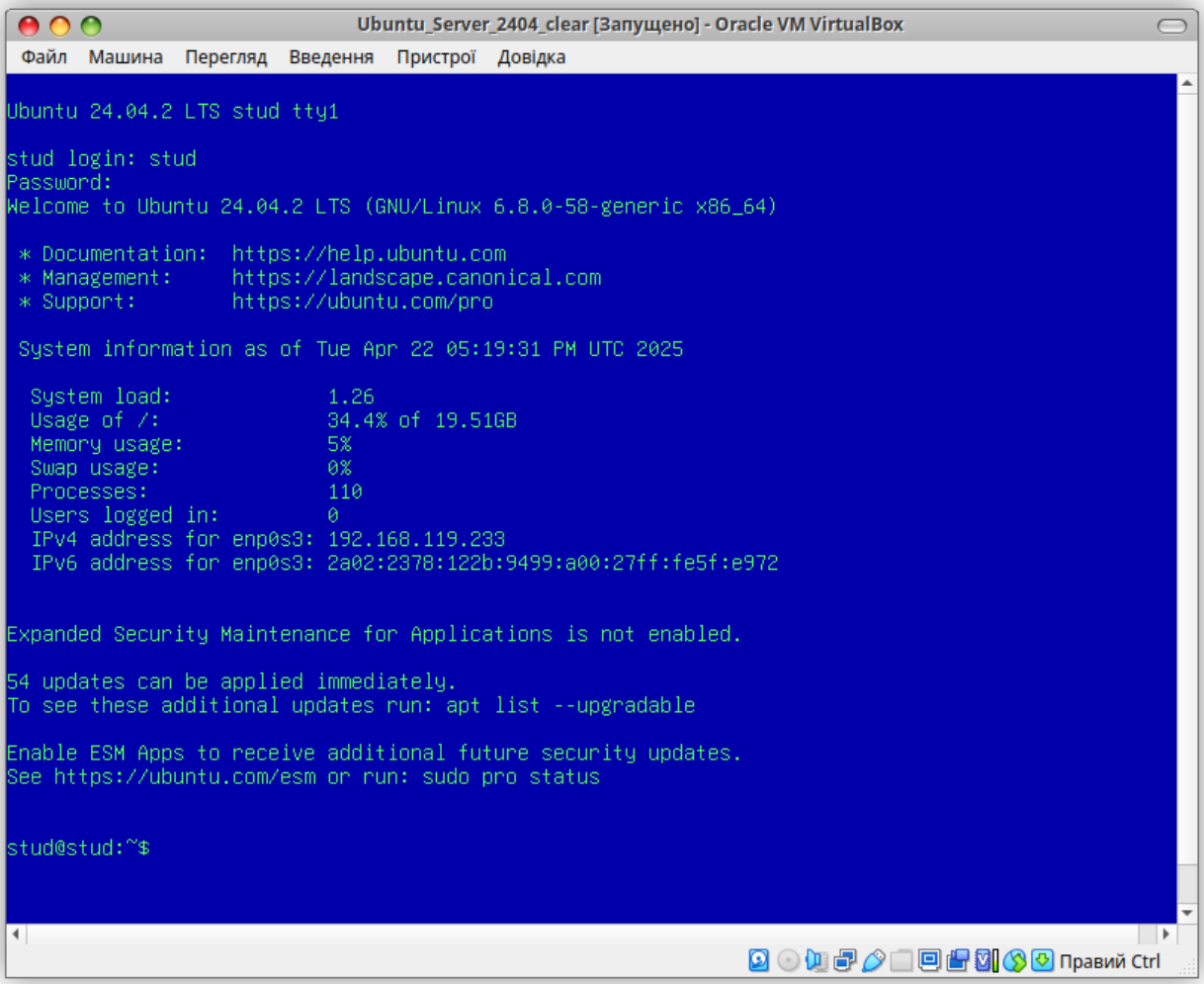


Рис. 1.1. Вигляд екрану після вдалого логіну в оточення Ubuntu Server

Якщо пакет відсутній (а при встановленні нової ОС так і є), виконайте команди:

```
sudo apt update
sudo apt -y install lighttpd
```

Відкрийте вікно браузера в основній ОС, та в рядок адресу введіть IP-адрес серверу (звісно за певних налаштувань мережі може бути інша IP-адреса ні ж та, що представлена на рис. 1.1). Вірний результат представлений на рис. 1.2.

Можлива ситуація, коли відповіді від Web-сервера немає. Тоді потрібно перевірити мережеві налаштування віртуальної машини (наприклад, в Oracle VirtualBox), а також відкриті порти за допомогою команди управління фаєрволом:

```
sudo ufw app list
```

Можна також виконати команду встановлення програми *nmap*, яка переглядає відкриті порти, та запустити її із вказівкою IP-адресу серверу. Наприклад:

```
sudo apt -y install nmap
nmap 192.168.119.233
```

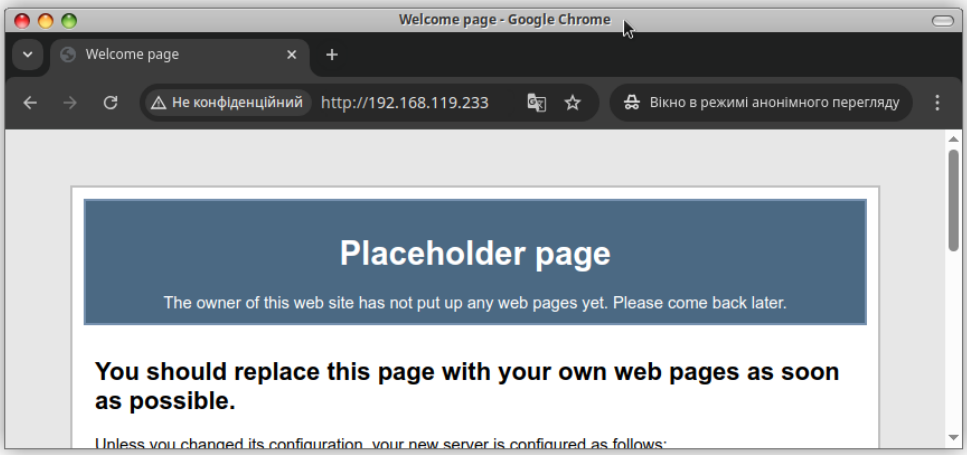


Рис. 1.2. Перевірка функціонування Web-серверу *Lighttpd*

Якщо все гаразд, то на сервері будуть відкриті за замовчанням TCP-порти: 22 (для роботи з протоколом *ssh*) та 80 для роботи з протоколом *http* (порт використовується встановленим Web-сервером).

У процесі роботи із сервером може знадобитися інформація стосовно статусу роботи *Lighttpd*, управління ним – наприклад, зупинення або перезапуск.

В сучасних Debian-подібних ОС це роблять командою *systemctl*. Наприклад:

```
systemctl status lighttpd
sudo systemctl stop lighttpd
sudo systemctl start lighttpd
sudo systemctl restart lighttpd
```

Ці команди доступні в сеансі роботи. Щоб повністю вимкнути можливість запуску *Lighttpd* на сервері, або для його повторного вмикання, виконують відповідно команди:

```
sudo systemctl disable lighttpd
sudo systemctl enable lighttpd
```

Розгорнемо на сервері підтримку сайтів двох організацій (віртуальних хостів). Наприклад, є сайт компанії *company1* з адресою: *company1.local.com* та сайт компанії *company2* з адресою: *company2.local.com*.

Спочатку зупинимо тимчасово роботу Web-серверу та виконаємо команди:

```
sudo systemctl stop lighttpd
```

```
sudo mkdir -p /var/www/company1.local.com/www
sudo mkdir -p /var/www/company2.local.com/www
chown -R www-data:www-data /var/www/company1.local.com
chown -R www-data:www-data /var/www/company2.local.com
chmod -R 750 /var/www/company1.local.com/www
chmod -R 750 /var/www/company2.local.com/www
sudo touch /var/www/company1.local.com/www/index.html
sudo touch /var/www/company2.local.com/www/index.html
```

За допомогою консольного редактору *nano* внесіть наступні рядки у файл *index.html* для першої компанії та збережіть зміни:

```
<html>
<body>
Hello, Company1!
</body>
</html>
```

Повторіть подібне для *index.html* другої компанії, змінивши привітання для Company2.

Підготуйте файли конфігурування віртуальних хостів для *Lighttpd* командами:

```
sudo touch /etc/lighttpd/conf-available/company1.local.com.conf
sudo touch /etc/lighttpd/conf-available/company2.local.com.conf
sudo ln -s /etc/lighttpd/conf-available/company1.local.com.conf \
/etc/lighttpd/conf-enabled/
sudo ln -s /etc/lighttpd/conf-available/company2.local.com.conf \
/etc/lighttpd/conf-enabled/
```

За допомогою консольного редактору *nano* додайте рядки у файл */etc/lighttpd/conf-available/company1.local.com.conf*:

```
$HTTP["host"] == "company1.local.com" {
    server.document-root = "/var/www/company1.local.com/www"
}
```

Збережіть цей вміст файлу та повторіть подібні дії для Company2.

Далі перевіримо конфігураційні файли на помилки командою:

```
sudo lighttpd -t -f /etc/lighttpd/lighttpd.conf
```

Якщо помилки відсутні, то можна запустити Web-сервер командою:

```
sudo systemctl start lighttpd
```

На хостовій системі знайдіть файл *hosts* (для Unix-сумісних: */etc/hosts*, для Windows-різновидів: *C:\Windows\System32\drivers\etc\hosts*) та зробіть

зіставлення IP-адреси серверу із символьними іменами наступним рядком (IP-адреса наведена для прикладу!):

```
192.168.119.233    company1.local.com    company2.local.com
```

Перевірте роботу віртуальних хостів для Company1 та Company2, вказавши в рядку адреси певну назву:

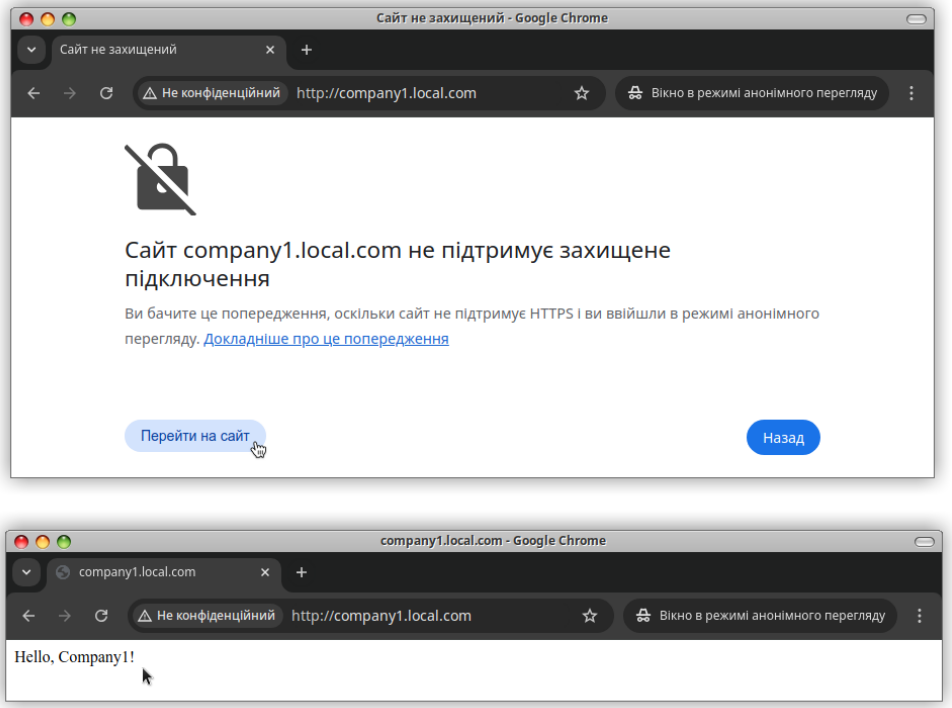


Рис. 1.3. Повернення результату звернення до Web-серверу

Зауважимо, що *Lighttpd* пише помилки за замовченням у лог-файл, розміщений за шляхом: */var/log/lighttpd/error.log*.

Завдання

Використовуючи матеріали лекцій та теоретично-практичної частини роботи, залучити Bash та Python для автоматизації процесу розгортання та налаштування Web-серверу за допомогою скрипту або скриптів для двох віртуальних хостів *company1.local.com* та *company2.local.com*. Реалізація повинна бути окремо мовою командного процесору Bash та окремо мовою програмування Python. Ім'я віртуального серверу або серверів повинні передаватися в основний скрипт як аргументи з командного рядку.

В звіті описати текст скриптів з коментарями та результатами роботи. Правила оформлення звіту подані в додатку А.

Робота №2.
Розгортання серверу GitLab

Мета роботи: опанувати процес розгортання серверу GitLab Community Edition через технологію контейнеризації docker в оточені ОС Debian GNU/Linux.

Теоретично-практична частина

GitLab – спеціалізоване ПЗ, Web-сервіс та платформа для управління репозиторіями проєктів та підтримки виконання різноманітних задач розробників, тестувальників та DevOps-інженерів. Був започаткований українськими розробниками. Перша версія вийшла у 2011 році. З того часу це ПЗ стало досить популярним та відомим для ІТ у всьому світі. Станом на квітень 2025 року поточна версія – 17.11. В якості прикладу в роботі розглянута саме ця версія.

З 2013 року *GitLab* існує у двох редакціях – Enterprise Edition (EE) та Community Edition (CE) під ліцензією MIT Expat.

Розглянемо процес встановлення інструментарію контейнеризації *docker* в середовищі GNU/Linux-сумісної ОС та розгортання серверу *GitLab*. Спочатку встановимо та мінімально налаштуємо ОС Debian GNU/Linux. Виконайте наступні кроки.

1. З ресурсу https://www.debian.org/download використати iso-образ для встановлення Debian GNU/Linux в оточені Oracle VirtualBox (під GNU/Linux-сумісну ОС, MS Windows) або UTM (під Apple macOS), або Hyper-V (під MS Windows). При встановлені вказати об’єм ОЗП мінімум 4096 Мбайт та 2 CPU-ядра, 30 Гб дискового простору, завантаження з оптичного диску, використання PAE/NX в опціях CPU, використання кешування контролером I/O хосту.
2. Обрати встановлення Debian GNU/Linux через графічний встановлювач, мова – українська. При встановлені задати пароль для суперкористувача *root*: *stud*. Обрати створення користувача *stud* з таким же паролем. Ім’я комп’ютера при встановлені: *debian*.
3. В процесі встановлення, якщо при обранні українського дзеркала *deb.debian.org* виникає помилка, повернутися до початку цього кроку й спробувати обрати інший дзеркальний сервер, наприклад, *debian.volia.net*.
4. В якості ПЗ для встановлення обрати полегшений варіант, а саме:
   – настільне оточення Debian;
   – Xfce;
   – SSH-сервер;
   – стандартні системні утиліти.

5. Після встановлення, під користувачем *stud* у вікні терміналу ввести команди:

```
$ su
# /sbin/usermod -a -G sudo stud
exit
```

6. Виконати повторний вхід у сеанс користувача *stud*.

7. Виконати підготовку для встановлення гостьових доповнень ОС та додаткових інструментів:

```
sudo apt update
sudo apt -y install build-essential htop tree ca-certificates curl
sudo apt -y install linux-headers-$(uname -r)
```

8. Для Oracle VirtualBox обрати відповідний пункт меню для встановлення гостьових доповнень ОС. Відкрити CD-диск в гостьовій ОС та у вікні терміналу виконати команди:

```
cd /media/cdrom0/
sudo bash ./VBoxLinuxAdditions.run
sudo reboot
```

9. Після перезавантаження Debian примусово вилучити диск з гостьовими доповненнями із пристрою CD-ROM.

10. Призначити в якості програми терміналу за замовченням – *xfce4-terminal* за допомогою команди:

```
sudo update-alternatives --config x-terminal-emulator
```

11. Встановити *docker*, використовуючи інструкцію за посиланням – https://docs.docker.com/engine/install/debian/#install-using-the-repository а саме виконати команди:

```
sudo install -m 0755 -d /etc/apt/keyrings
sudo curl -fsSL \
https://download.docker.com/linux/debian/gpg -o /etc/apt/keyrings/docker.asc
sudo chmod a+r /etc/apt/keyrings/docker.asc
echo "deb [arch=$(dpkg --print-architecture) \
signed-by=/etc/apt/keyrings/docker.asc] \
https://download.docker.com/linux/debian \
$(. /etc/os-release && echo "$VERSION_CODENAME") stable" | \
sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt update
sudo apt -y install docker-ce docker-ce-cli \
containerd.io docker-buildx-plugin docker-compose-plugin
```

12. Перевірити функціонування сервісу *docker* командою:

```
systemctl status docker
```

13. Для розгортання серверу *GitLab* підготуйте каталоги:

```
mkdir -p ~/gitlab/server
mkdir ~/gitlab/server/{data,logs,config}
sudo -s
echo 'export GITLAB_HOME=/home/stud/gitlab/server' >> /root/.bashrc
exit
```

14. Перевідкрити вікно терміналу.

15. В каталозі */home/stud/gitlab* створити файл *compose.yaml* з таким вмістом (для розгортання серверу *GitLab* версії 17.11.1):

```
services:
  gitlab:
    image: gitlab/gitlab-ce:17.11.1-ce.0
    container_name: gitlab
    restart: always
    hostname: 'gitlab.example.com'
    environment:
      GITLAB_OMNIBUS_CONFIG: external_url 'https://gitlab.example.com'
    ports:
      - '80:80'
      - '443:443'
      - '2025:22'
    volumes:
      - ${GITLAB_HOME}/config:/etc/gitlab
      - ${GITLAB_HOME}/logs:/var/log/gitlab
      - ${GITLAB_HOME}/data:/var/opt/gitlab
    shm_size: '256m'
```

16. Для першого запуску серверу *GitLab* у вікні терміналу консолі виконати команди (IP-адреса наведена для прикладу):

```
sudo -s
echo '192.168.119.190 gitlab.example.com' >> /etc/hosts
cd /home/stud/gitlab
docker compose up
```

17. Дочекатися повного старту серверу. Перший старт буде досить довгим.

18. Відкрити приватне вікно браузеру в гостьовій ОС та ввести URL:

```
https://127.0.0.1/
```

На запит про з’єднання потрібно погодитися. Якщо сервер стартував, то з’явиться сторінка для входу (рис. 2.1).

19. Відкрити друге вікно терміналу та ввести команду:

```
sudo cat /home/stud/gitlab/server/config/initial_root_password
```

20. Скопіювати тимчасовий пароль користувача *root* для входження в оточення серверу *GitLab* та виконати вхід в оточення серверу.

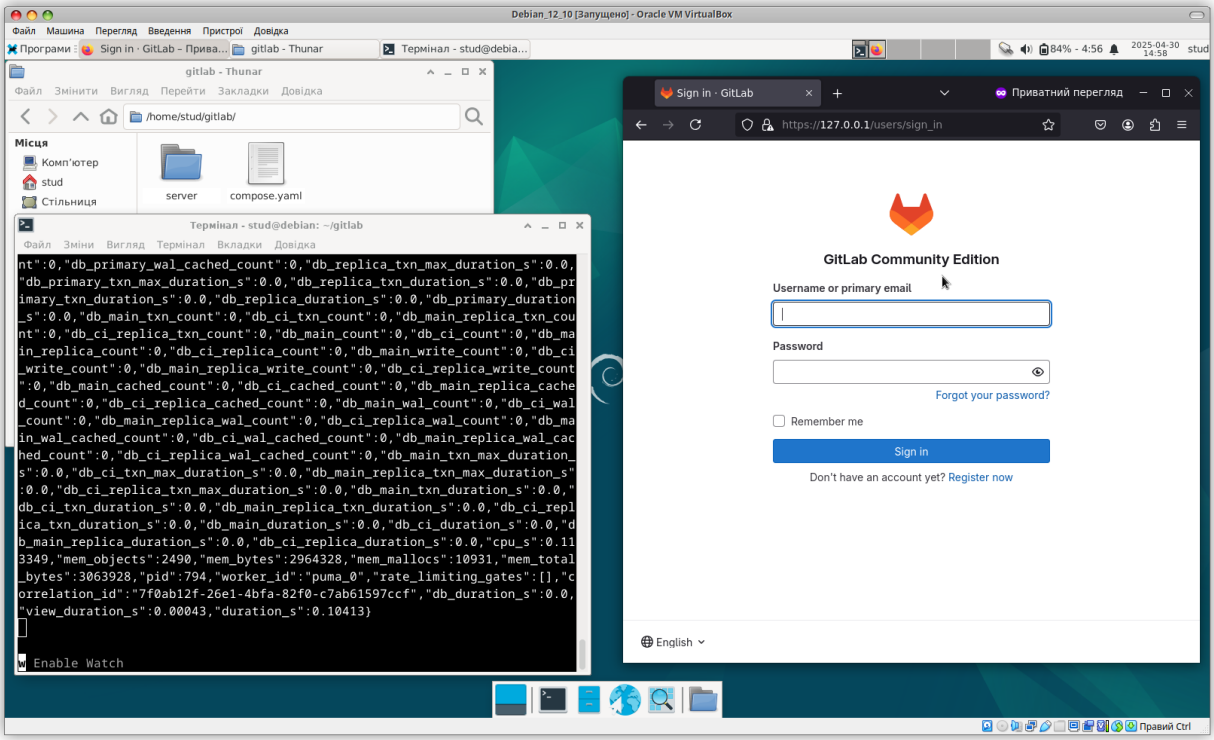


Рис. 2.1. Сторінка входу на сервер *GitLab*

21. В оточенні серверу *GitLab*, скористатися пунктом меню *Edit Profile* користувача *root*, а потім пунктом меню *Password*. Змінити поточний пароль користувача *root*, наприклад, на такий:

```
#123&stud
```

22. В оточенні серверу *GitLab* створити нового користувача (посилання *Add people*) з ім'ям *stud* та поштовою скринькою, наприклад, *stud@localhost*. Після створення увійти у режим редагування профілю *stud* та ввести його пароль, наприклад:

```
#123&9st0
```

23. Вийти з оточення *root* та здійснити вхід на сервер *GitLab* під користувачем *stud*. Можна повторити введення нового пароля (повторити введення останнього пароля) та виконати повторну спробу входу на сервер.

Таким чином, мінімальне налаштування серверу *GitLab* завершено.

Вийдіть з оточення *stud* на сервері *GitLab* та закрийте вікно браузеру.

У вікні терміналу консолі введіть комбінацію клавіш Ctrl+C та зупинить роботу контейнера із сервером *GitLab*.

В *root*-консолі введіть команду (поточний каталог – `/home/stud/gitlab`):

```
docker compose down
```

Повторить запуск серверу *GitLab* у режимі демона, використавши команду в оточені *root*-консолі:

```
docker compose up -d
```

Час на повторний запуск серверу *GitLab* може скласти близько трьох хвилин в залежності від потужності процесору та кількості ядер. Поки триває запуск серверу *GitLab*, відкрийте нове вікно терміналу консолі та скористайтеся утилітою *htop* для перегляду ресурсів ОЗП. Якщо оперативної пам'яті недостатньо (за замовченням для використання серверу *GitLab* мінімальний об'єм повинен бути 4096 Мбайт), то при наявності у хостовій ОС доступної пам'яті слід збільшити об'єм ОЗП для віртуальної машини (рис. 2.2).

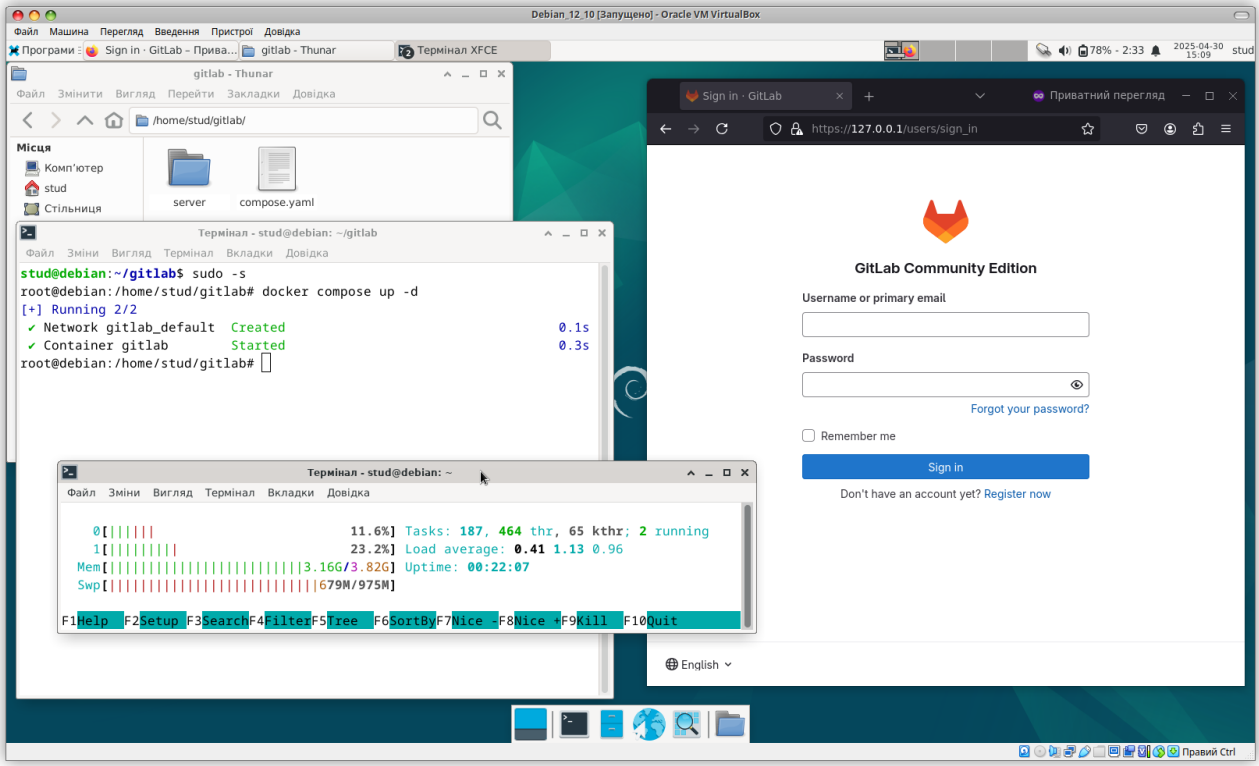


Рис. 2.2. Повторний запуск серверу *GitLab* та моніторинг використання оперативної пам'яті гостьової ОС через утиліту *htop*

Перевірте доступність профілю користувача *stud* на сервері *GitLab* та протестуйте завершення роботи серверу *GitLab* командою у відкритій раніше *root*-консолі: `docker compose down`.

Після успішного встановлення та мінімального налаштування віртуальної машини із сервером *GitLab*, перевірте доступність мережевого з'єднання між нею та віртуальною машиною із Ubuntu Server (рис. 2.3).

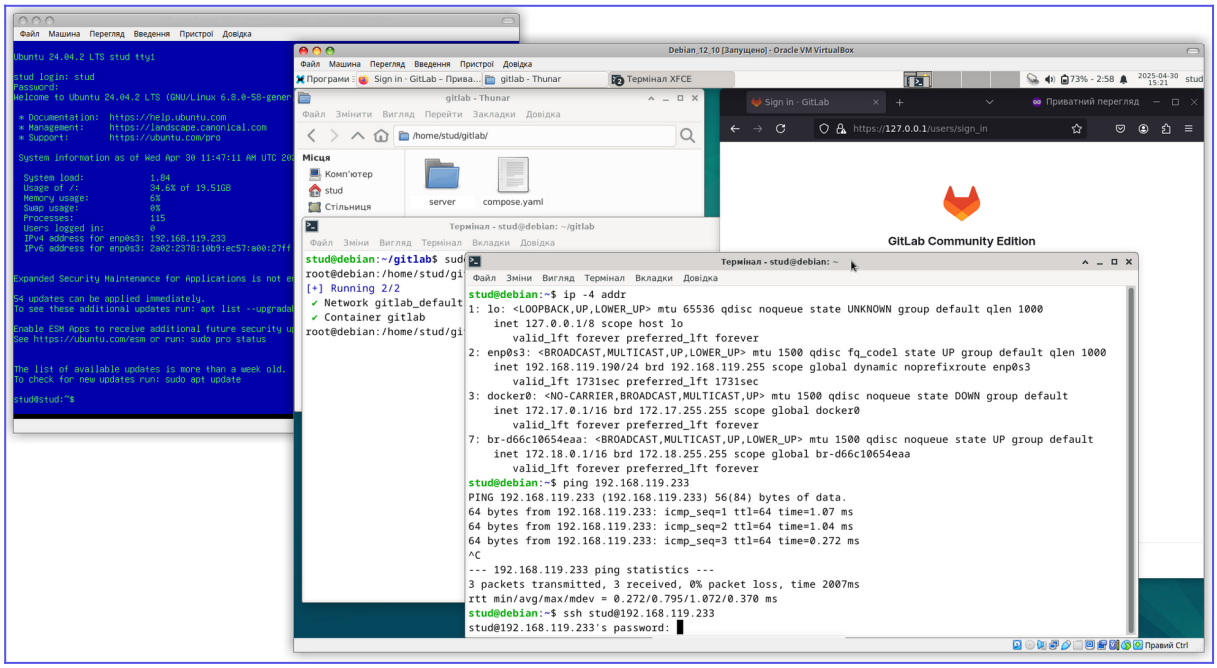


Рис. 2.3. Перевірка наявності мережевого зв'язку між віртуальними машинами із Ubuntu Server та Debian GNU/Linux

Для перевірки можна використовувати утиліту *ping* та програму зв'язку по протоколу SSH (рис. 2.3). IP-адреси, представлені на рисунку, надані для прикладу. Вони повинні мати відношення до мережі із одним номером (у прикладі це мережа 192.168.119.0). Додатково дивитися на рис. Б.1 у додатку Б.

Завдання

Використовуючи теоретично-практичну частину роботи, виконати розгортання серверу *GitLab* на базі *docker*-контейнеру у віртуальній машині із ОС Debian GNU/Linux.

У звіті навести підтвердження створення в оточені серверу *GitLab* користувача *stud*, а також надати результати тестування мережевого з'єднання між віртуальними машинами із Ubuntu Server та Debian. Включити опис послідовності всіх дій. Правила оформлення звіту подані в додатку А.

При захисті роботи володіти знаннями для відповіді на питання викладача за матеріалами лекцій та теоретично-практичної частини роботи.

Робота №3.
Використання мови Go

Мета роботи: автоматизувати збір статистики при обранні певних URL на базі власного Web-сервісу, що розроблений мовою програмування Go. Забезпечити збірку, тестування та розгортання цього сервісу в оточенні Ubuntu Server.

Теоретично-практична частина

Вхідний стан для роботи:

– ОС Ubuntu Server із встановленим та налаштованим Web-сервером з роботи №1;

– створений в Ubuntu Server користувач *stud* із дозволом доступу по протоколу SSH;

– встановлене середовище розробки MS Visual Studio Code на стороні клієнта (наприклад, в ОС Debian GNU/Linux; встановлювач доступний за посиланням: https://code.visualstudio.com/); клієнтом не може виступати віртуальна машина із встановленим сервером *GitLab*.

Нехай на віртуальному хості *company1.local.com* є сторінка з деякими посиланнями на товари – комп'ютерні комплектуючи. Потрібно побудувати деякий сервіс (під назвою, наприклад, *counter*) з певною реалізацією REST API, який буде дозволяти накопичувати запити зі сторінки Web-серверу при обранні певних URL користувачами сторінки. Сервіс *counter* повинен прослуховувати TCP-порт 8030.

Примітка: REST (*REpresentational State Transfer*) – поширений зручний архітектурний стиль, на базі якого будується переважна більшість сучасних Web API сервісів. Кажуть, що якщо Web API слідує принципам REST, то таке API називають *RESTful API*. Одним із принципів REST є Uniform Interface (уніфікований інтерфейс) – REST API спирається на чотири основні інтерфейси: ресурси, HTTP-методи (GET, POST, PUT/PATCH, DELETE), представлення ресурсів (як само дані подано клієнту), та посилання між ресурсами. Це створює однаковість у способі взаємодії клієнта та сервера.

Для спрощення задачі будемо вважати, що дані статистики зберігатимуться в БД SQLite з іменем *products*.

Нехай маємо кінцеві точки з'єднань REST API, які описані в таблиці 3.1.

Нехай БД *products* містить таблиці *category* та *monitor* з описом структур, що представлені в таблиці 3.2.

Таблиця 3.1

Приклади кінцевих точок REST API

№ з/п	*Кінцева точка з'єднання*	*HTTP-метод*	*Опис*
1.	/categories	GET	Повертає перелік категорій товарів
2.	/categories/monitors	GET	Повертає перелік моніторів
3.	/categories/monitors	POST	Створює новий опис монітору (додає його до списку моніторів)
4.	/categories/monitors/id	GET	Повертає опис обраного монітору
5.	/categories/monitors/id	PATCH	Збільшує лічильник використання посилання на обраний монітор в БД

Таблиця 3.2

Структури таблиць БД *products*

Ім'я поля	*Тип*	*Опис*
Таблиця *category*		
pk_id	Ціле	Ідентифікатор категорії товару
name	Рядок	Назва категорії товару
Таблиця *monitor*		
pk_id	Ціле	Ідентифікатор товару
fk_category_id	Ціле	Ідентифікатор категорії товару
name	Рядок	Назва монітору
count	Ціле	Лічильник кліків на посилання товару

Розробку додатку мовою Go краще проводити в будь-якій зручній ОС. Після завершення тестування, її розміщують на стороні серверу.

Наприклад, нехай розробка ведеться в оточені ОС Debian GNU/Linux. Тоді для встановлення Go для поточного розробника виконуємо команди:

```
mkdir ~/install && cd ~/install
wget https://go.dev/dl/go1.24.2.linux-amd64.tar.gz
sudo tar -zxvf ./go1.24.2.linux-amd64.tar.gz -C /opt
mkdir -p $HOME/go_projects/{bin,src}
echo 'export GOROOT=/opt/go' >> $HOME/.bashrc
echo 'export GOPATH=$HOME/go_projects' >> $HOME/.bashrc
echo 'export PATH=$PATH:$GOROOT/bin:$GOPATH/bin' >> $HOME/.bashrc
sudo reboot
```

Після повторного логіну в ОС під власним профілем, у вікні терміналу потрібно перевірити доступність компілятора Go командою:

```
go version
```

Для подальшого виконання завдання потрібно встановити в ОС додаткові пакунки командами:

```
$ sudo apt -y install sqlite3 libsqlite3-dev sqlitebrowser
```

Для роботи із локальними БД SQLite3, можна скористатися програмою *SQLiteBrowser*. Якщо вона недоступна із репозиторію ОС, то її можна завантажити за посиланням: https://sqlitebrowser.org/dl/.

Наступним кроком йде створення каталогу проєкту та його ініціалізація. Для цього у вікні терміналу консолі виконайте команди:

```
$ mkdir -p $GOPATH/src/counter && cd $GOPATH/src/counter
$ touch ./main.go
$ go mod init counter
```

Додайте в модуль *main.go*, наприклад, наступний код:

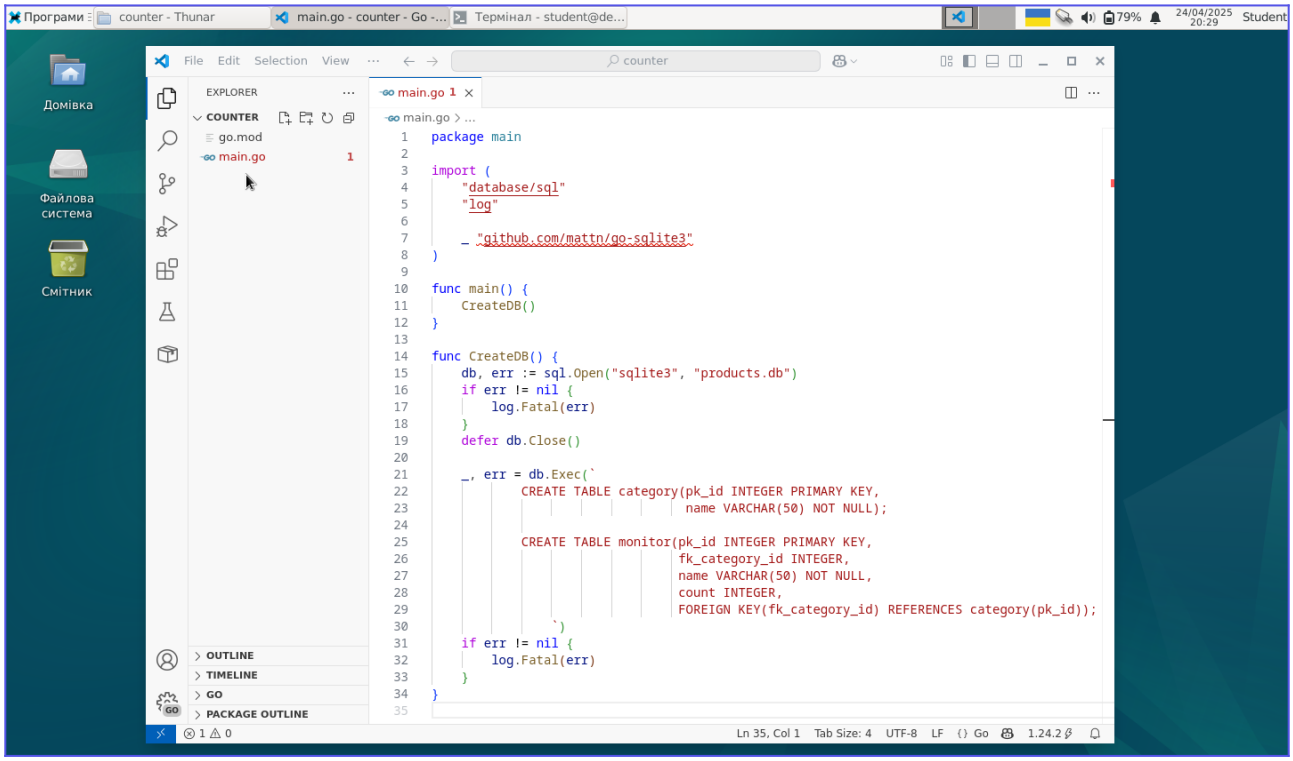


Рис. 3.1. Початковий код в модулі *main.go*

Після введення коду, знаходячись в терміналі у каталозі *$GOPATH/src/counter*, введіть команди:

```
$ go get github.com/mattn/go-sqlite3
$ go mod tidy
```

Введіть команду збірки та виконайте програму:

```
$ go build
$ ./counter
```

Текст коду, представлений на рис. 3.1, дозволить створити БД і його можна взяти за основу. Створену БД (файл *products.db*) можна відкрити окремо в програмі *SQLiteBrowser* (рис. 3.2) та окремо побудувати ER-діаграму БД у, наприклад, програмі *DBeaver Community* (за необхідності community-версія доступна за посиланням – https://dbeaver.io/download/):

```
$ sqlitebrowser ./products.db
```

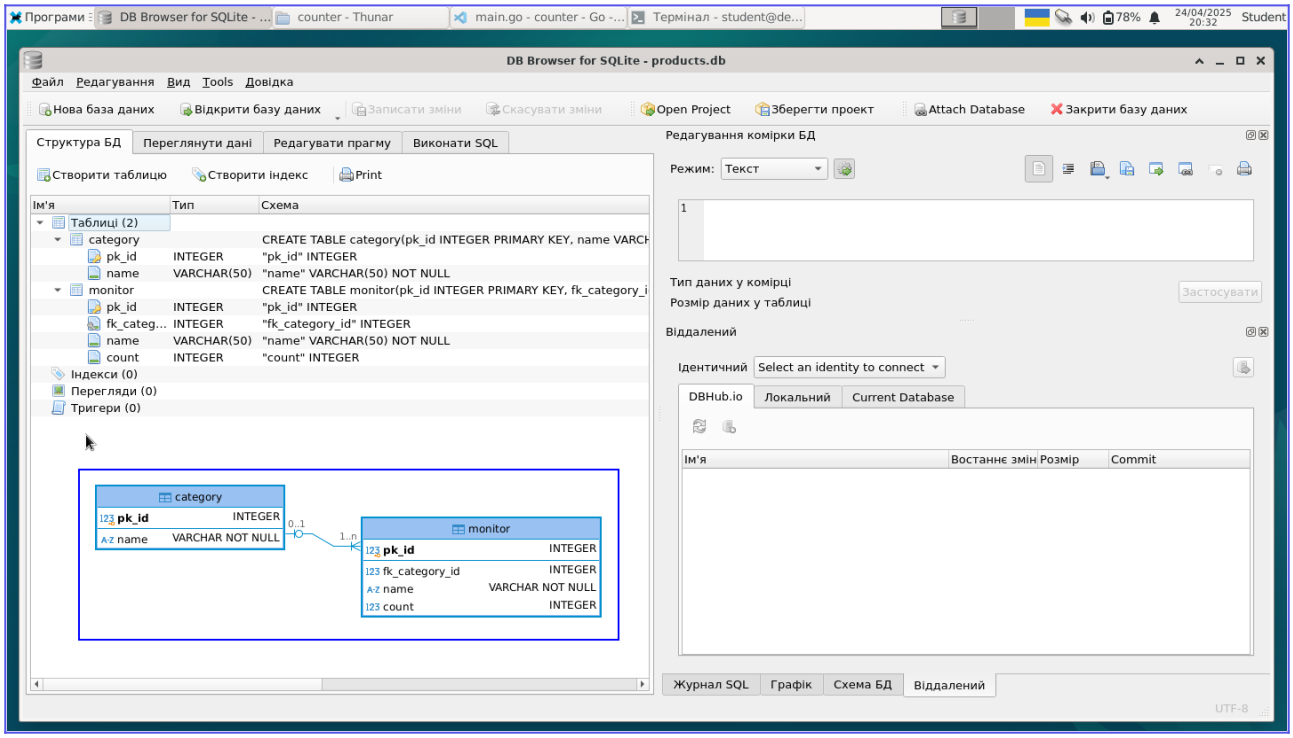


Рис. 3.2. БД *products.db*, що відкрита в програмі *SQLiteBrowser* та створена окремо в програмі *DBeaver Community* її ER-діаграма

Для подальшого тестування можна заповнити таблиці наступними рядками за допомогою SQL-команд:

```
INSERT INTO category (pk_id, name) VALUES (1, ' Monitors');
INSERT INTO monitor (pk_id, fk_category_id, name, count)
VALUES (1, 1, 'Samsung Odyssey G5 C27G55TQBI', 0);
INSERT INTO monitor (pk_id, fk_category_id, name, count)
VALUES (2, 1, 'LG 27GN800-B', 0);
```

```
INSERT INTO monitor (pk_id, fk_category_id, name, count)
VALUES (3, 1, 'DELL U2720Q', 0);
```

Для більшої гнучкості розробки подібних програм, пропонується вдатися до певних практик, які рекомендують розбиття тексту коду програми на модулі та підкаталоги. Так, наприклад, якщо розробка не використовує якісь особливі пакети чи фреймворки, а залучає переважно пакети стандартної бібліотеки, то можна обрати наступну файлову структуру проєкту:

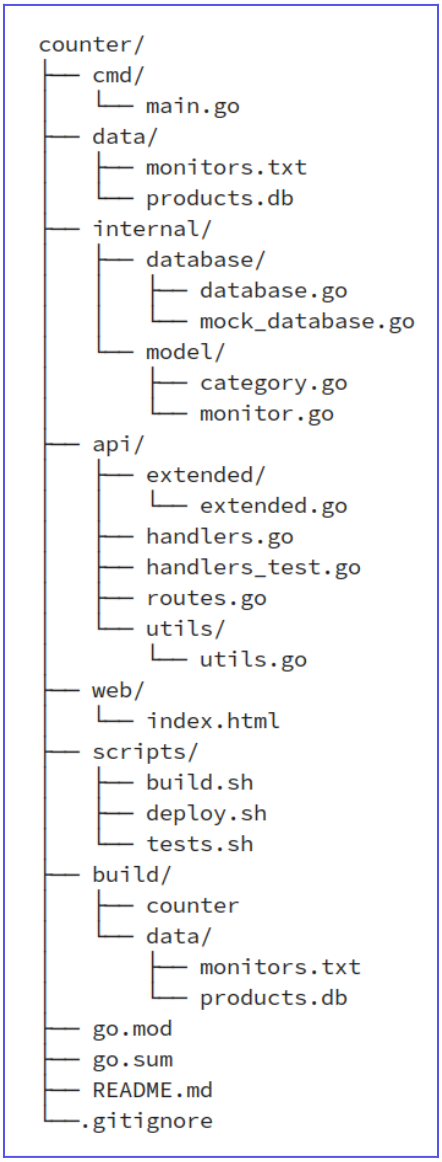


Рис. 3.3. Приклад файлової структури проєкту *counter*

Можлива й інша файлова структура проєкту.

Призначення модулів:

– *main.go* (пакет *main*) – містить головну точку входу в програму – функцію `main()`, в якій контролюються параметри програми – певні аргументи

командного рядку, а також додаткові функції, що забезпечують виконання певних задач – створення БД, запуск сервера, виведення довідки;

– *database.go* (пакет *counter/internal/database*) – містить опис інтерфейсу роботи з БД та відповідну структуру, що його реалізовує для роботи через драйвер, а також глобальну змінну, що експортується та надає доступ до БД. Інтерфейс описує методи для роботи із таблицями БД *products*;

– *mock_database.go* (пакет *counter/internal/database*) – містить структуру, що реалізовує інтерфейс з *database.go* для цілей тестування (імітує БД *products*);

– *category.go* та *monitor.go* (пакет *counter/internal/model*) – містять опис структур даних для опрацювання із відповідними сутностями та їх множинами, а також структури DTO (Data Transfer Object) для використання певних операцій API (структури створюються із врахуванням експортування полів);

– *routes.go* (пакет *counter/api*) – містить функцію, яка пов'язує кінцеву точку API із функцією-обробником HTTP-запиту;

– *handlers.go* (пакет *counter/api*) – реалізація функцій-обробників HTTP-запитів;

– *handlers_test.go* (пакет *counter/api*) – містить функції тестування кінцевих точок API;

– *extended.go* (пакет *counter/api/extended*) – реалізація внутрішніх додаткових функцій, які реалізують певні частини функцій-обробників HTTP-запитів;

– *utils.go* (пакет *counter/api/utils*) – містить допоміжну функцію для узагальненої реалізації відповіді сервера із зазначенням коду статусу та повідомлення; використовується у функціях модулів пакетів *counter/api* та *counter/api/extended*.

В каталозі *counter/scripts* розміщені скрипти збірки, тестування та публікації на сервер.

БД *products* повинна бути створена ще до запуску серверу – в неї можна занести вже передвизначені дані із форматованого текстового файлу *data/monitors.txt* із наступним прикладом вмісту:

```
1,"Monitors"
1,"Samsung Odyssey G5 C27G55TQBI"
2,"LG 27GN800-B"
3,"DELL U2720Q"
```

Першим рядком вказується категорія товару із своїм ідентифікатором, а в наступних рядках описуються конкретні моделі моніторів.

Нехай створення БД відбувається командою:

```
counter --createdb
```

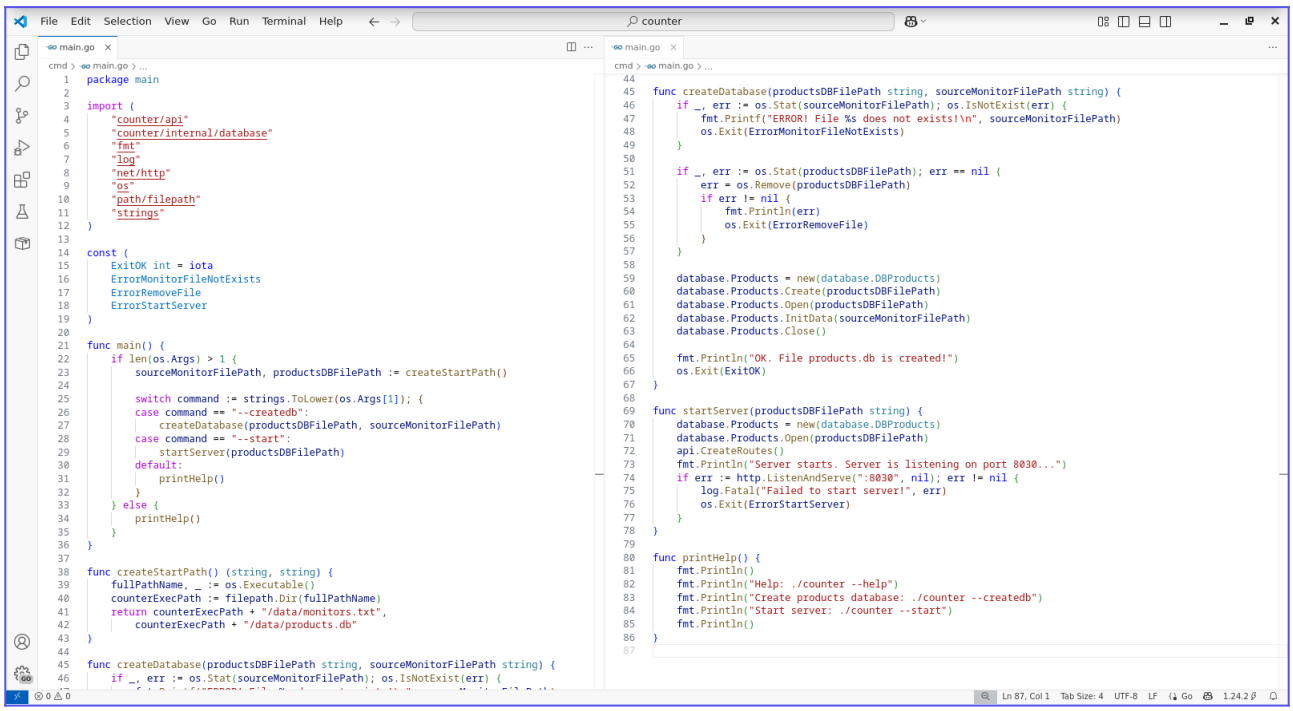
Запуск Web-сервісу проводиться командою:

```
counter --start
```

Довідка по програмі виводиться командою:

```
counter --help
```

Тоді маємо наступний приклад тексту програмного коду модуля *main.go*:



```go
package main

import (
    "counter/api"
    "counter/internal/database"
    "fmt"
    "log"
    "net/http"
    "os"
    "path/filepath"
    "strings"
)

const (
    ExitOK int = iota
    ErrorMonitorFileNotExists
    ErrorRemoveFile
    ErrorStartServer
)

func main() {
    if len(os.Args) > 1 {
        sourceMonitorFilePath, productsDBFilePath := createStartPath()

        switch command := strings.ToLower(os.Args[1]); {
        case command == "--createdb":
            createDatabase(productsDBFilePath, sourceMonitorFilePath)
        case command == "--start":
            startServer(productsDBFilePath)
        default:
            printHelp()
        }
    } else {
        printHelp()
    }
}

func createStartPath() (string, string) {
    fullPathName, _ := os.Executable()
    counterExecPath := filepath.Dir(fullPathName)
    return counterExecPath + "/data/monitors.txt",
        counterExecPath + "/data/products.db"
}

func createDatabase(productsDBFilePath string, sourceMonitorFilePath string) {
    if _, err := os.Stat(sourceMonitorFilePath); os.IsNotExist(err) {
        fmt.Printf("ERROR! File %s does not exists!\n", sourceMonitorFilePath)
        os.Exit(ErrorMonitorFileNotExists)
    }

    if _, err := os.Stat(productsDBFilePath); err == nil {
        err = os.Remove(productsDBFilePath)
        if err != nil {
            fmt.Println(err)
            os.Exit(ErrorRemoveFile)
        }
    }

    database.Products = new(database.DBProducts)
    database.Products.Create(productsDBFilePath)
    database.Products.Open(productsDBFilePath)
    database.Products.InitData(sourceMonitorFilePath)
    database.Products.Close()

    fmt.Println("OK. File products.db is created!")
    os.Exit(ExitOK)
}

func startServer(productsDBFilePath string) {
    database.Products = new(database.DBProducts)
    database.Products.Open(productsDBFilePath)
    api.CreateRoutes()
    fmt.Println("Server starts. Server is listening on port 8030...")
    if err := http.ListenAndServe(":8030", nil); err != nil {
        log.Fatal("Failed to start server!", err)
        os.Exit(ErrorStartServer)
    }
}

func printHelp() {
    fmt.Println()
    fmt.Println("Help: ./counter --help")
    fmt.Println("Create products database: ./counter --createdb")
    fmt.Println("Start server: ./counter --start")
    fmt.Println()
}
```

Рис. 3.4. Приклад вмісту модуля *main.go*

З рис. 3.4 бачимо, що окремі дії програми, які повинні виконуватися при передачі програмі через командний рядок аргументів-опцій, винесені в окремі функції.

На рис. 3.5 представлений приклад реалізацій певних функцій в модулі *database.go*.

Робота із файлом БД SQLite3 виконується через глобальну змінну `Products` модуля *database.go* – рядок 34 (рис. 3.5). Оскільки ім'я починається з прописного символу, то за замовченням воно буде експортуватися з модуля. Ім'я файлу БД передається параметром у методи `Create()` та `Open()` реалізованого інтерфейсу `IDBProducts` (рядки 60, 61 та 71, рис. 3.4) в модулі *main.go.*

Функції для відпрацювання із запитами до БД представлені на рис. 3.6.

Як бачимо із рис. 3.6, для використання SQL-запитів залучаються функції пакету *database/sql* стандартної бібліотеки мови Go, що відпрацьовують із типом `sql.DB`, а саме функції: `Query()`, `Prepare()`, та `Exec()`. Окрім результатів виконання, вони повертають й помилки. Тому при необхідності їх

можна контролювати та впливати на повернення результату виконання власних функцій. Результати запитів, а також певні дані для обміну, зберігаються у структурах, що описані у модулях *category.go* та *monitor.go* (рис. 3.7).

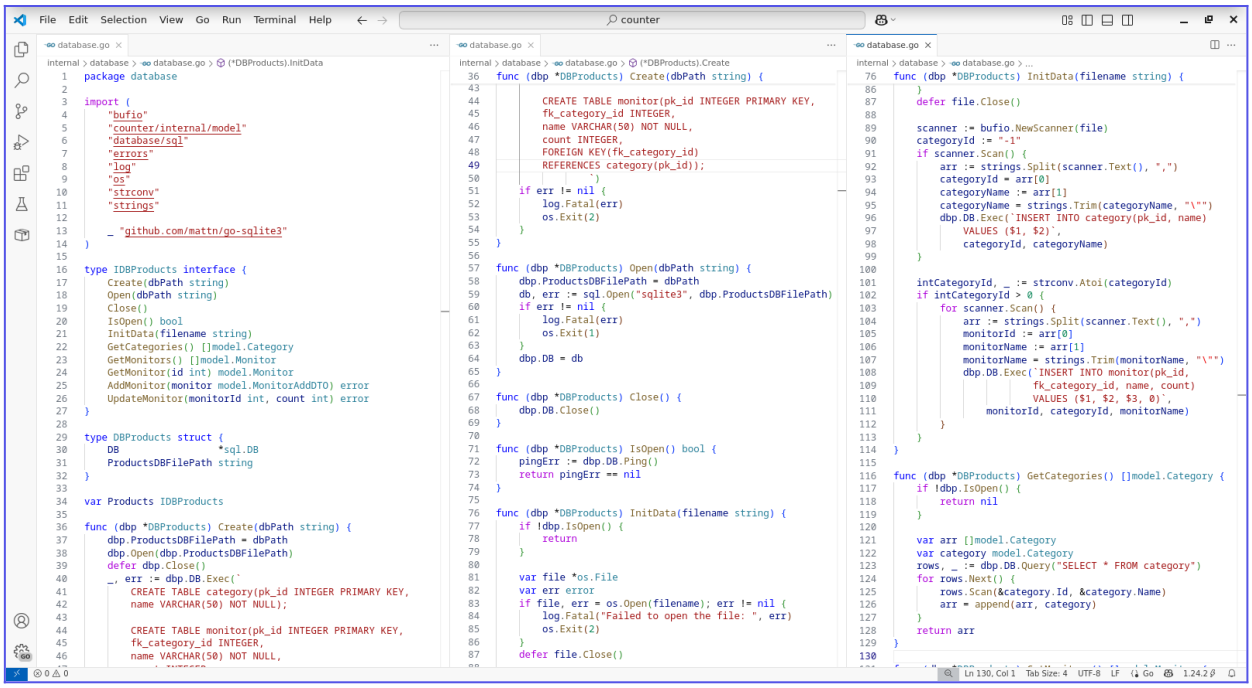


Рис. 3.5. Приклад реалізації функцій створення БД, відкриття/закриття/перевірки доступності БД, імпортування даних із текстового файлу в БД та повернення масиву категорій товарів

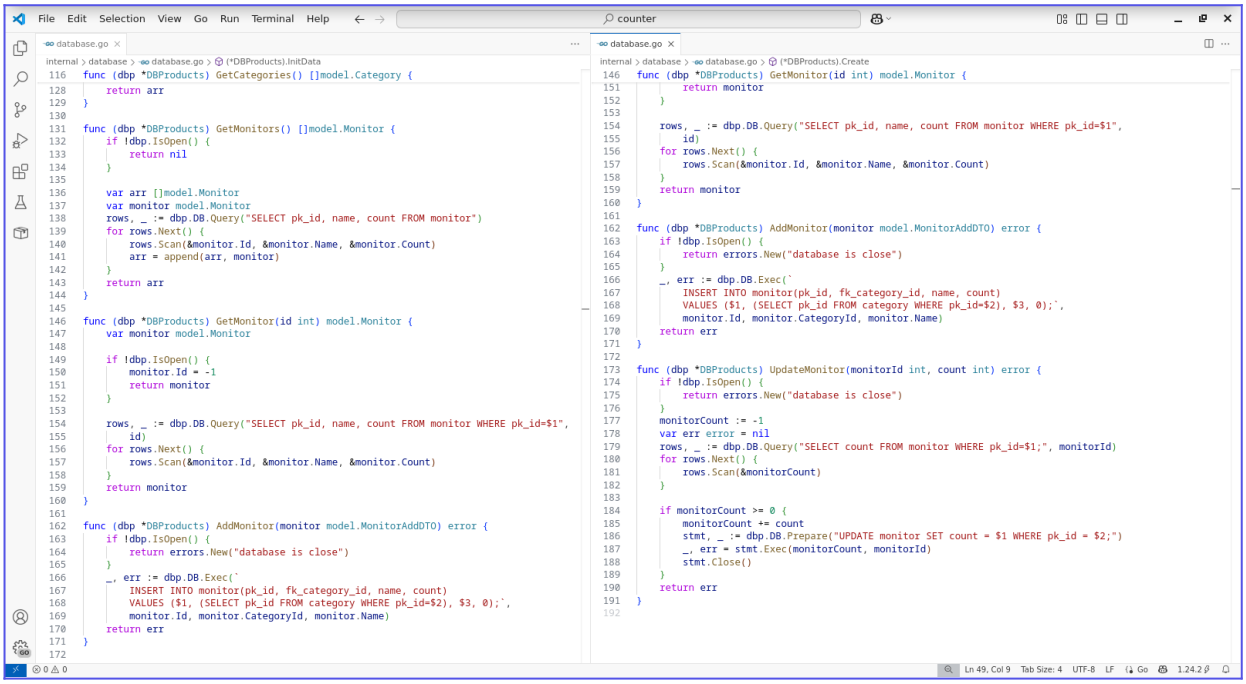


Рис. 3.6. Приклад реалізації функцій повернення масиву моніторів, повернення інформації про певний монітор, додавання нового монітору в БД (з використанням певного DTO) та оновлення поля *count* вказаного монітору

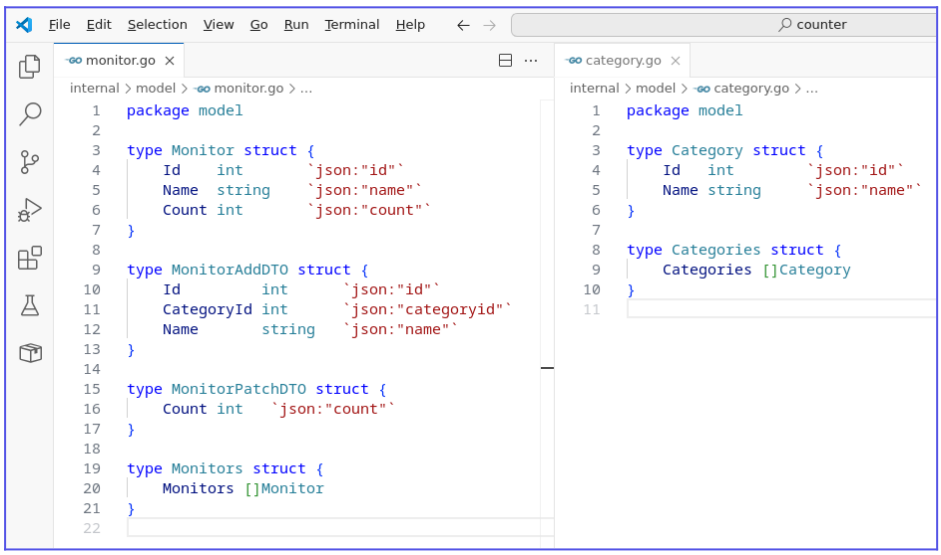


Рис. 3.7. Моделі даних з БД, що описані структурами

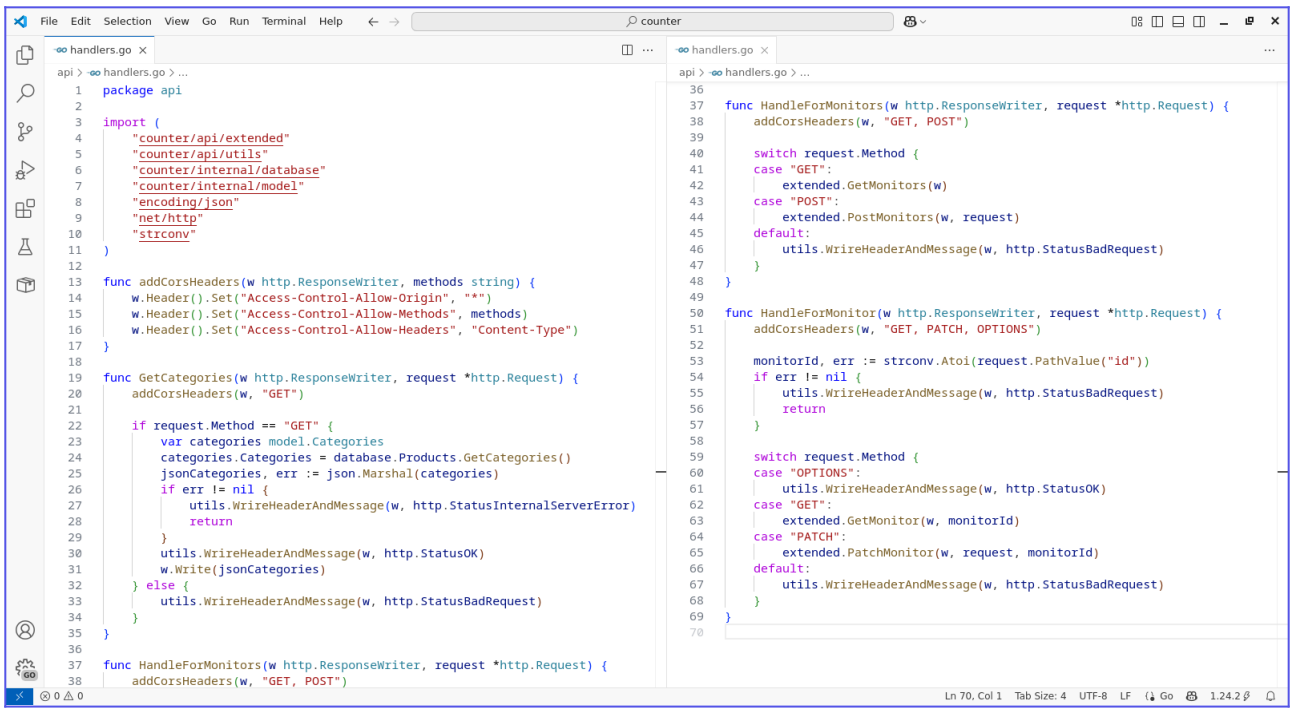
Поля структур іменуються з верхнього регістру для їх експортування.

Реалізація зв'язування кінцевих точок API з таблиці 3.1 відбувається в окремій функції модуля *routes.go* і може виглядати наступним чином:

```
package api

import (
    "net/http"
)

func CreateRoutes() {
    http.HandleFunc("/categories", GetCategories)
    http.HandleFunc("/categories/monitors", HandleForMonitors)
    http.HandleFunc("/categories/monitors/{id}", HandleForMonitor)
}
```

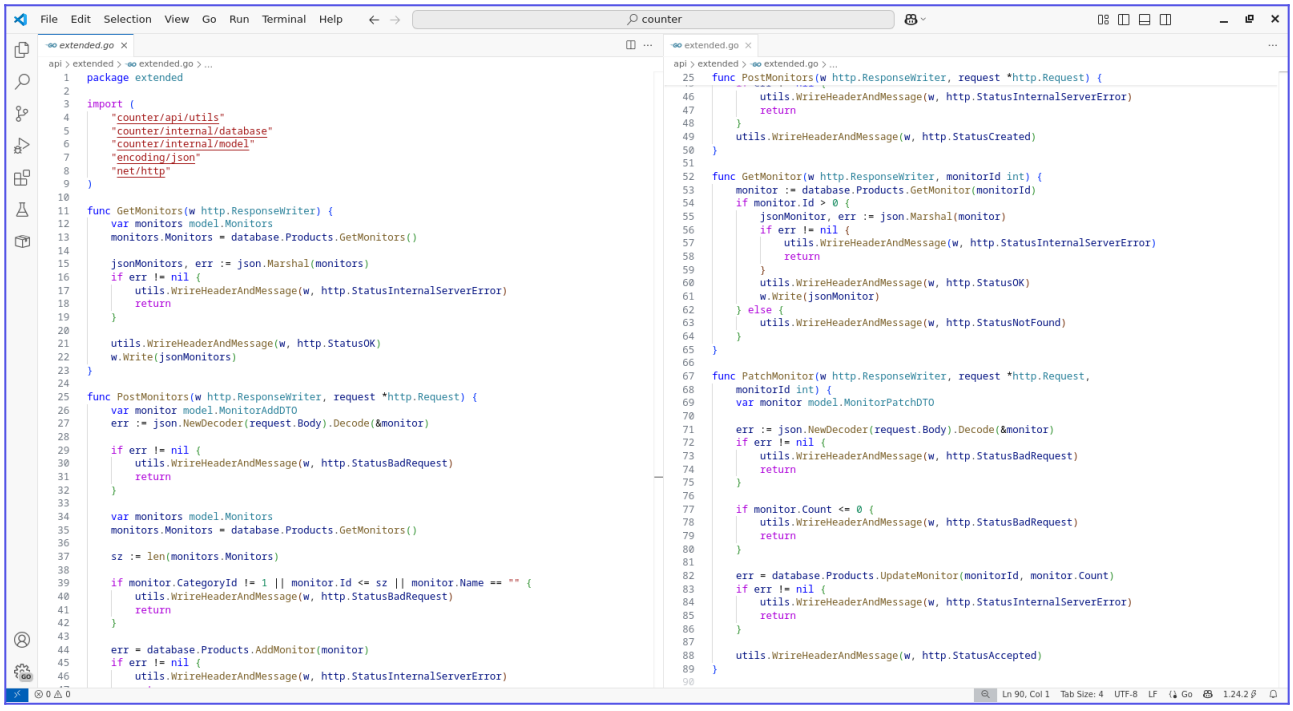
Оскільки функції-обробники (рис. 3.8) описані в тому ж пакеті *counter/api*, то вони викликаються без вказівки назви пакету. Сама функція `CreateRoutes()` викликається в момент старту серверу – рядок 72, рис. 3.4.

У функціях на рис. 3.8 йде перевірка методу HTTP-запиту згідно таблиці 3.1. Існують різноманітні бібліотеки та фреймворкі, які значно полегшують подібну обробку, але у прикладі використана тільки стандартна бібліотека Go. Як бачимо, за певними кінцевими точками API повинні оброблятися різні методи HTTP-запитів, тому конкретна реалізація подібних функцій винесена в додатковий модуль *extended.go* (рис. 3.9).



```go
package api

import (
    "counter/api/extended"
    "counter/api/utils"
    "counter/internal/database"
    "counter/internal/model"
    "encoding/json"
    "net/http"
    "strconv"
)

func addCorsHeaders(w http.ResponseWriter, methods string) {
    w.Header().Set("Access-Control-Allow-Origin", "*")
    w.Header().Set("Access-Control-Allow-Methods", methods)
    w.Header().Set("Access-Control-Allow-Headers", "Content-Type")
}

func GetCategories(w http.ResponseWriter, request *http.Request) {
    addCorsHeaders(w, "GET")

    if request.Method == "GET" {
        var categories model.Categories
        categories.Categories = database.Products.GetCategories()
        jsonCategories, err := json.Marshal(categories)
        if err != nil {
            utils.WriteHeaderAndMessage(w, http.StatusInternalServerError)
            return
        }
        utils.WriteHeaderAndMessage(w, http.StatusOK)
        w.Write(jsonCategories)
    } else {
        utils.WriteHeaderAndMessage(w, http.StatusBadRequest)
    }
}

func HandleForMonitors(w http.ResponseWriter, request *http.Request) {
    addCorsHeaders(w, "GET, POST")

    switch request.Method {
    case "GET":
        extended.GetMonitors(w)
    case "POST":
        extended.PostMonitors(w, request)
    default:
        utils.WriteHeaderAndMessage(w, http.StatusBadRequest)
    }
}

func HandleForMonitor(w http.ResponseWriter, request *http.Request) {
    addCorsHeaders(w, "GET, PATCH, OPTIONS")

    monitorId, err := strconv.Atoi(request.PathValue("id"))
    if err != nil {
        utils.WriteHeaderAndMessage(w, http.StatusBadRequest)
        return
    }

    switch request.Method {
    case "OPTIONS":
        utils.WriteHeaderAndMessage(w, http.StatusOK)
    case "GET":
        extended.GetMonitor(w, monitorId)
    case "PATCH":
        extended.PatchMonitor(w, request, monitorId)
    default:
        utils.WriteHeaderAndMessage(w, http.StatusBadRequest)
    }
}
```

Рис. 3.8. Приклад реалізації функцій-обробників HTTP-запитів



```go
package extended

import (
    "counter/api/utils"
    "counter/internal/database"
    "counter/internal/model"
    "encoding/json"
    "net/http"
)

func GetMonitors(w http.ResponseWriter) {
    var monitors model.Monitors
    monitors.Monitors = database.Products.GetMonitors()

    jsonMonitors, err := json.Marshal(monitors)
    if err != nil {
        utils.WriteHeaderAndMessage(w, http.StatusInternalServerError)
        return
    }

    utils.WriteHeaderAndMessage(w, http.StatusOK)
    w.Write(jsonMonitors)
}

func PostMonitors(w http.ResponseWriter, request *http.Request) {
    var monitor model.MonitorAddDTO
    err := json.NewDecoder(request.Body).Decode(&monitor)

    if err != nil {
        utils.WriteHeaderAndMessage(w, http.StatusBadRequest)
        return
    }

    var monitors model.Monitors
    monitors.Monitors = database.Products.GetMonitors()

    sz := len(monitors.Monitors)

    if monitor.CategoryId != 1 || monitor.Id <= sz || monitor.Name == "" {
        utils.WriteHeaderAndMessage(w, http.StatusBadRequest)
        return
    }

    err = database.Products.AddMonitor(monitor)
    if err != nil {
        utils.WriteHeaderAndMessage(w, http.StatusInternalServerError)
        return
    }
    utils.WriteHeaderAndMessage(w, http.StatusCreated)
}

func GetMonitor(w http.ResponseWriter, monitorId int) {
    monitor := database.Products.GetMonitor(monitorId)
    if monitor.Id > 0 {
        jsonMonitor, err := json.Marshal(monitor)
        if err != nil {
            utils.WriteHeaderAndMessage(w, http.StatusInternalServerError)
            return
        }
        utils.WriteHeaderAndMessage(w, http.StatusOK)
        w.Write(jsonMonitor)
    } else {
        utils.WriteHeaderAndMessage(w, http.StatusNotFound)
    }
}

func PatchMonitor(w http.ResponseWriter, request *http.Request,
    monitorId int) {
    var monitor model.MonitorPatchDTO

    err := json.NewDecoder(request.Body).Decode(&monitor)
    if err != nil {
        utils.WriteHeaderAndMessage(w, http.StatusBadRequest)
        return
    }

    if monitor.Count <= 0 {
        utils.WriteHeaderAndMessage(w, http.StatusBadRequest)
        return
    }

    err = database.Products.UpdateMonitor(monitorId, monitor.Count)
    if err != nil {
        utils.WriteHeaderAndMessage(w, http.StatusInternalServerError)
        return
    }

    utils.WriteHeaderAndMessage(w, http.StatusAccepted)
}
```

Рис. 3.9. Приклад реалізації функцій-обробників HTTP-запитів
в модулі *extended.go*

При реалізації функцій-обробників активно використовується пакет для підтримки роботи із даними формату JSON (пакет *encoding/json* стандартної бібліотеки мови Go). Він реалізує кодування та декодування JSON, що описано

в RFC 7159. Для цих операцій використовується механізм маршалінгу або демаршалінгу із залученням опису структур даних.

Наприклад, у рядку 24 (рис. 3.8) отримуємо перелік категорій товарів, а на наступному рядку проводимо маршалінг, отримуючи байт-кодовану послідовність символів, які представляють дані в json-форматі. Цю послідовність й повертає сервер – рядок 31 (рис. 3.8). Зверніть увагу, що в модулі активно використовується власна утилітарна функція `WriteHeaderAndMessage()`, яка пише у потік HTTP-відповіді заголовок, код статусу, та (за необхідності) текстове повідомлення про помилку (рис. 3.10).

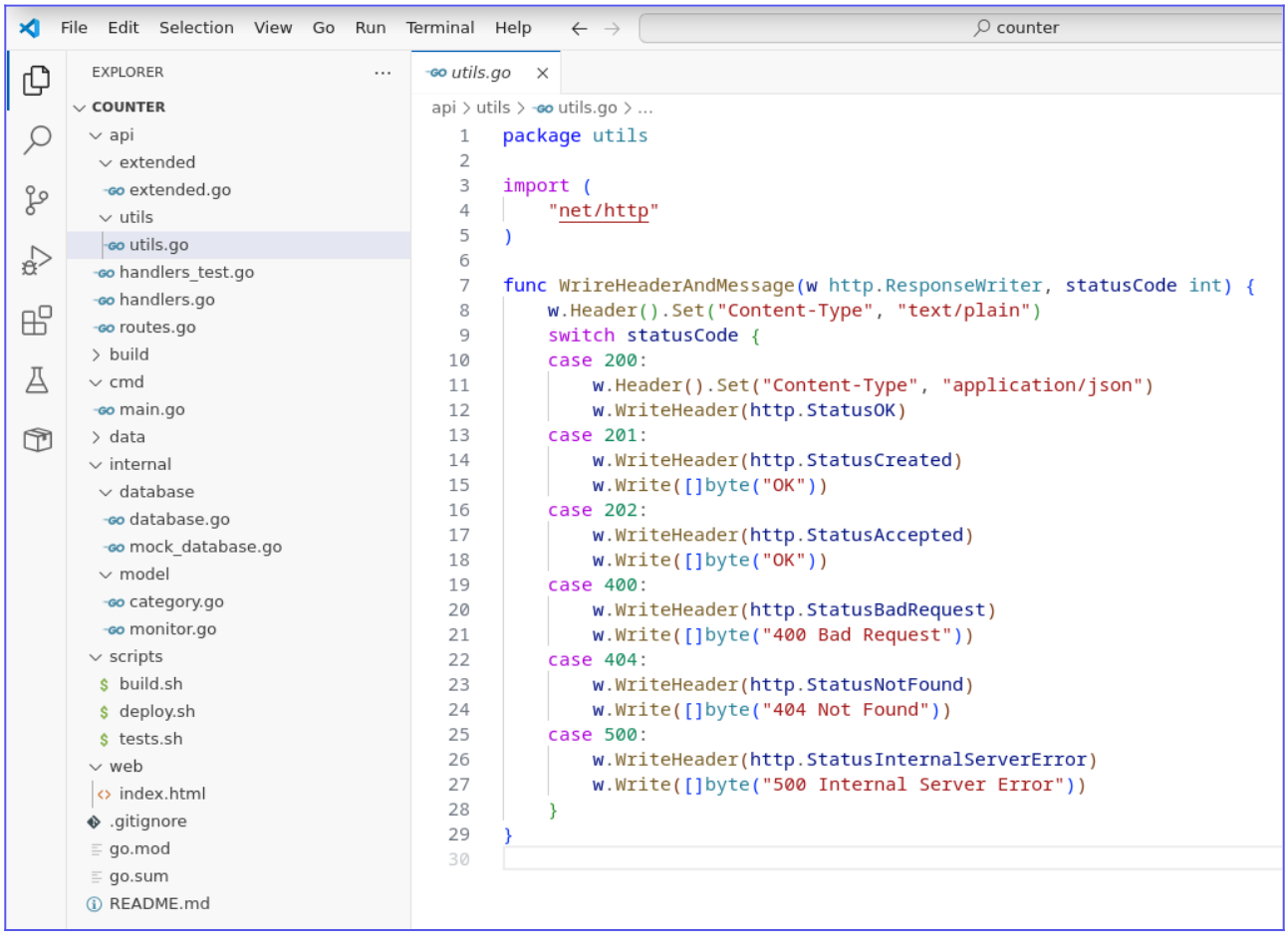


Рис. 3.10. Приклад реалізації утилітарної функції узагальнення створення заголовків та текстів кодів статусу HTTP-відповіді

В представленому спрощеному варіанті проєкту відсутнє логування. Його також можна додати окремо. В сучасній практиці, як правило, реалізують логування (у вікно терміналу консолі та у зовнішні файли).

При розробці подібних проєктів активно залучаються різні техніки тестування. Охоплення тестами може бути на різних рівнях. В якості прикладу в роботі розглянуті тільки два тести кінцевих точок з'єднання за GET-методом HTTP для: `/categories` та `/categories/monitors`. Тести розміщені в модулі

handlers_test.go (рис. 3.11). Оскільки при виконанні модульних тестів не залучаються фізичні підключення до БД, то сама БД імітується деякою моделлю. Використовується описаний функціями інтерфейс `IDBProducts`. Такі так звані *mock*-функції (імітаційні функції) описані окремо в модулі *mock_database.go* (рис. 3.12 та рис. 3.13).

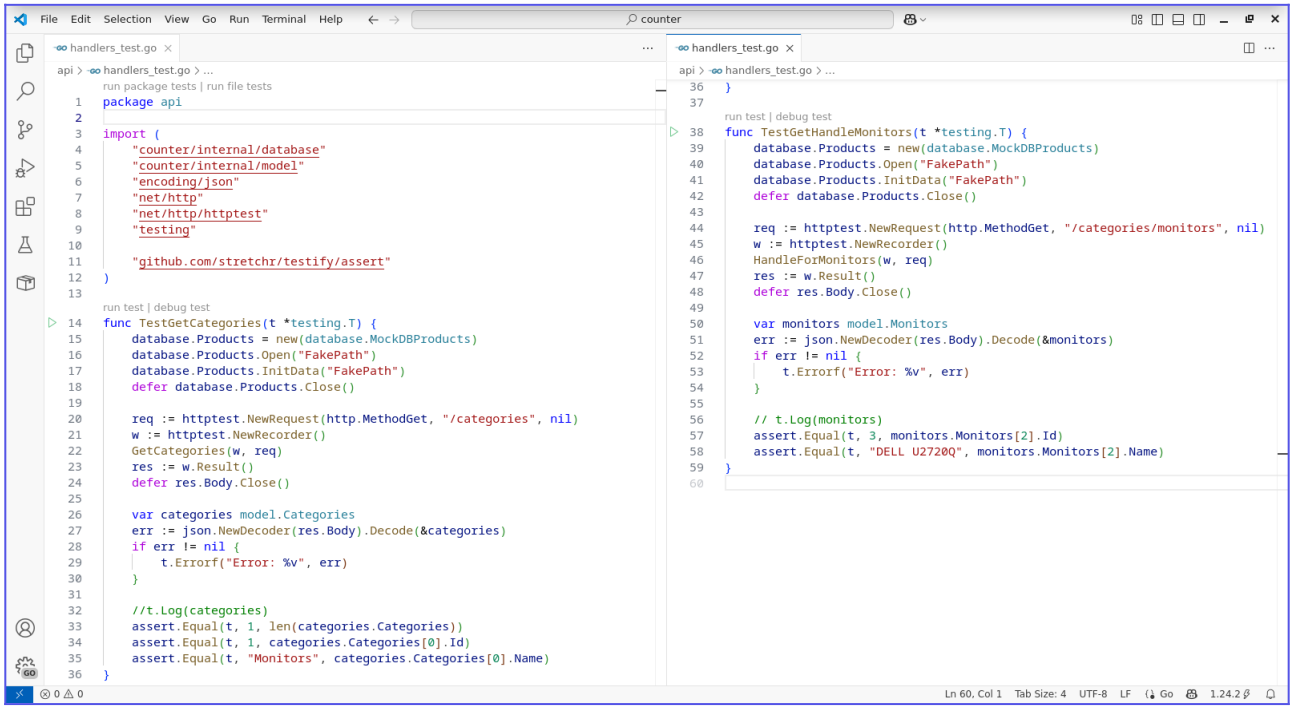


Рис. 3.11. Приклад реалізації функцій тестування деяких кінцевих точок API

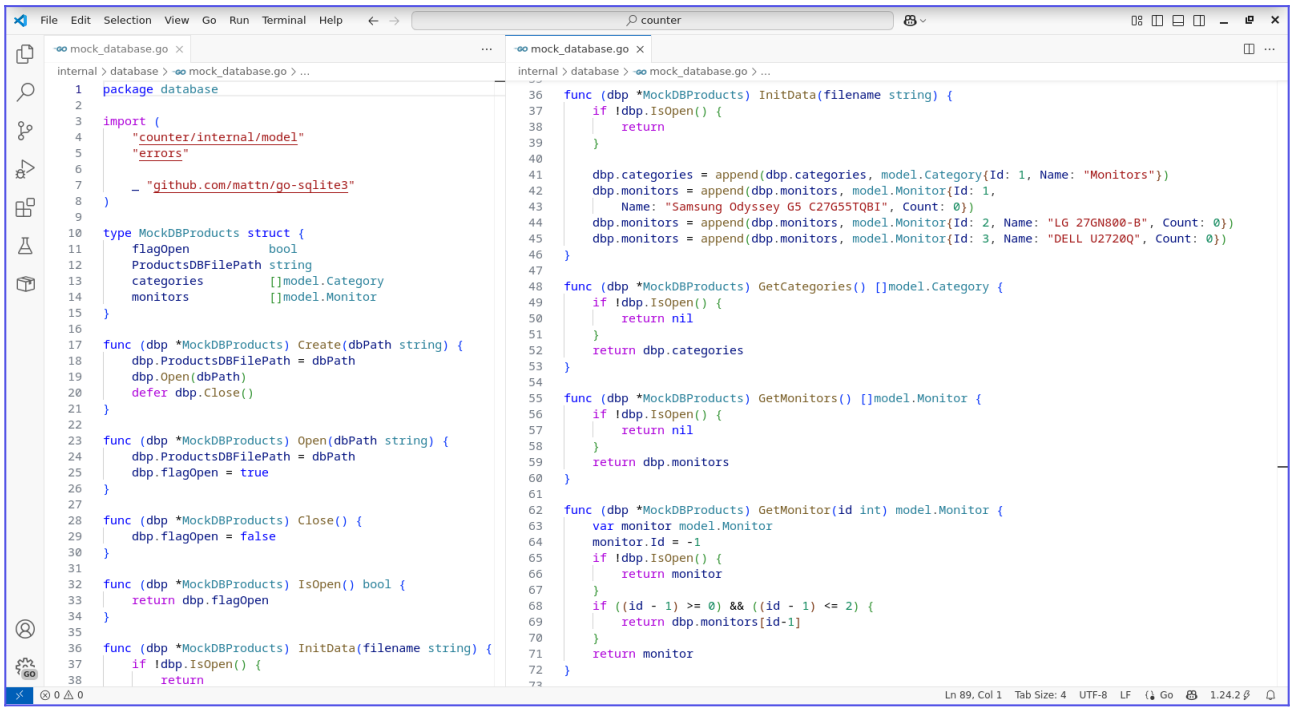


Рис. 3.12. Приклад реалізації імітаційних функцій (*mock*-функцій) інтерфейсу `IDBProducts`

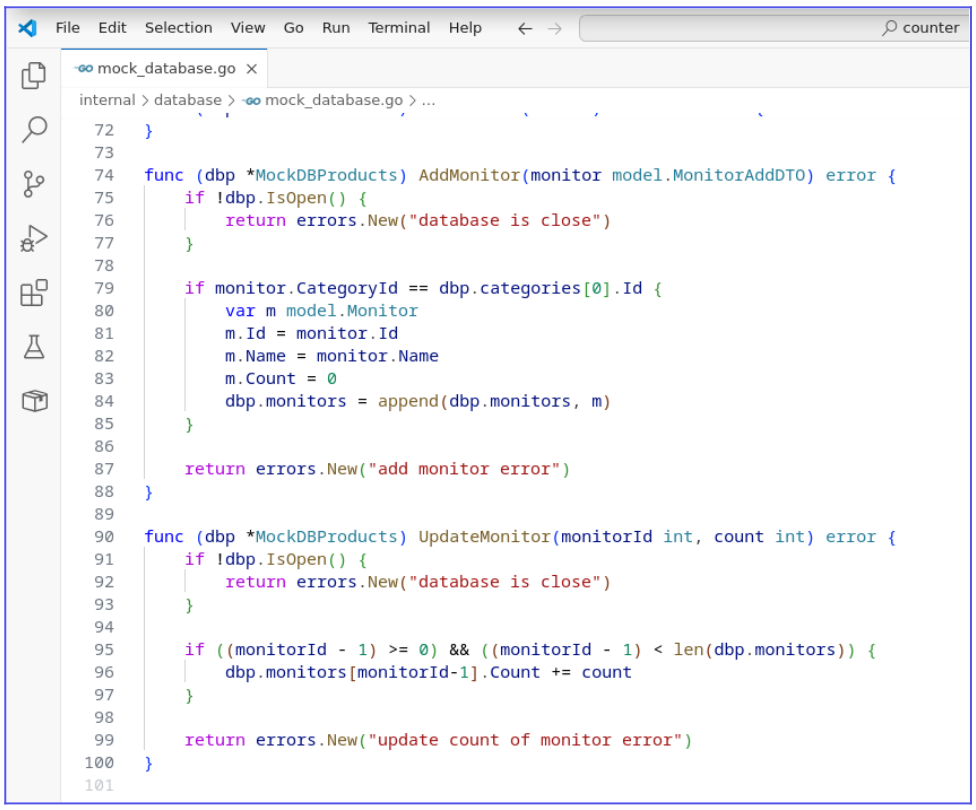


Рис. 3.13. Приклад реалізації імітаційних функцій (*mock*-функцій) інтерфейсу `IDBProducts` (продовження)

При реалізації функцій тестування використовуються пакети зі складу стандартної бібліотеки мови Go – *testing* та *httptest*. Додатково використовуються можливості відомого стороннього пакета *testify*.

Спочатку створюється об’єкт підключення до *mock*-БД та імітуються виклики певних функцій роботи з БД (рядки 15-18 та 39-42 на рис. 3.11). Причому саме при виклику методу `InitData()` відбувається ініціалізація імітованої БД деяким тестовим набором даних (рядки 36-46 на рис. 3.12).

Імітація двох таблиць БД *products* представлена двома масивами, елементи яких задані відповідними моделями даних. Масиви описані у структурі `MockDBProducts` (рядки 10-15 на рис. 3.12), яка й буде реалізовувати функції інтерфейсу `IDBProducts`, але у функціях проводиться вже імітація роботи з БД – проводиться робота саме з описаними масивами даних, а не таблицями реальної БД.

Таким чином, після ініціалізації імітованої БД переходимо до створення HTTP-запиту за GET-методом до певної кінцевої точки (рядки 20 та 44, рис. 3.11). На наступних рядках створюємо об’єкт для HTTP-відповіді. Далі викликається функція-обробник, яка надає відповідь (рядки 22 та 46, рис. 3.11), після чого йде отримання та розбір результату HTTP-відповіді. За допомогою тестових функцій підпакету *testify.assert* перевіряємо на співпадіння щодо очікуваних результатів (рядки 33-35 та 57, 58 на рис. 3.11).

Збірку, тестування та публікацію (деплой) можна описати окремими командними скриптами (рис. 3.14). Зауважимо, що скрипт публікування результатів збірки проєкту пропонується описати власноруч.

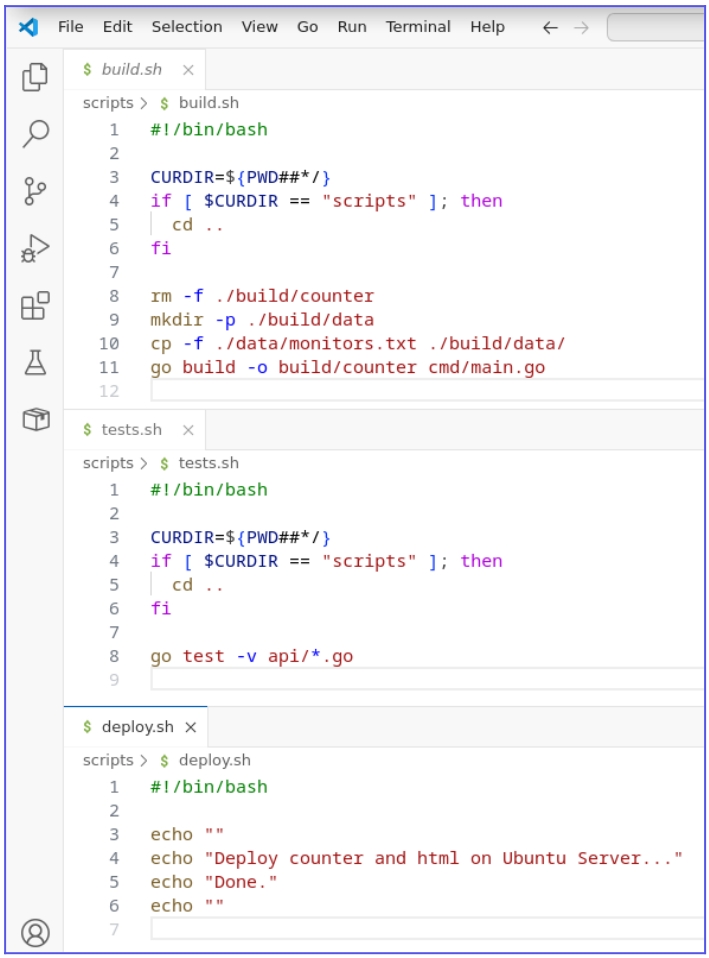


Рис. 3.14. Приклади вмісту скриптів проєкту

Після збірки та виконання процедури модульного тестування, можна виконати перевірку функціонування REST API за допомогою спеціалізованих інструментів (не тільки програми Web-браузера). До них часто відносять як професійні засоби, на кшталт Postman, так і інструменти вільно доступні та безкоштовні, наприклад, такі розширення MS Visual Studio Code, як: Postcode, REST Client. Також використовують утиліту командного рядку *curl* або подібну. Можливе використання спеціалізованого інструменту JMeter. При формуванні HTTP-запитів та аналізу відповідей із використанням мов JavaScript, TypeScript використовують, наприклад, можливості асинхронного API функції `fetch()`.

Приклади тестування відповіді серверу, який виконується, представлені на рис. 3.15 – 3.17. Використаний консольний інструмент *curl*, Web-браузер Google Chrome, можливості інтерпретатора мови Python та розширення Postcode для середовища MS Visual Studio Code.

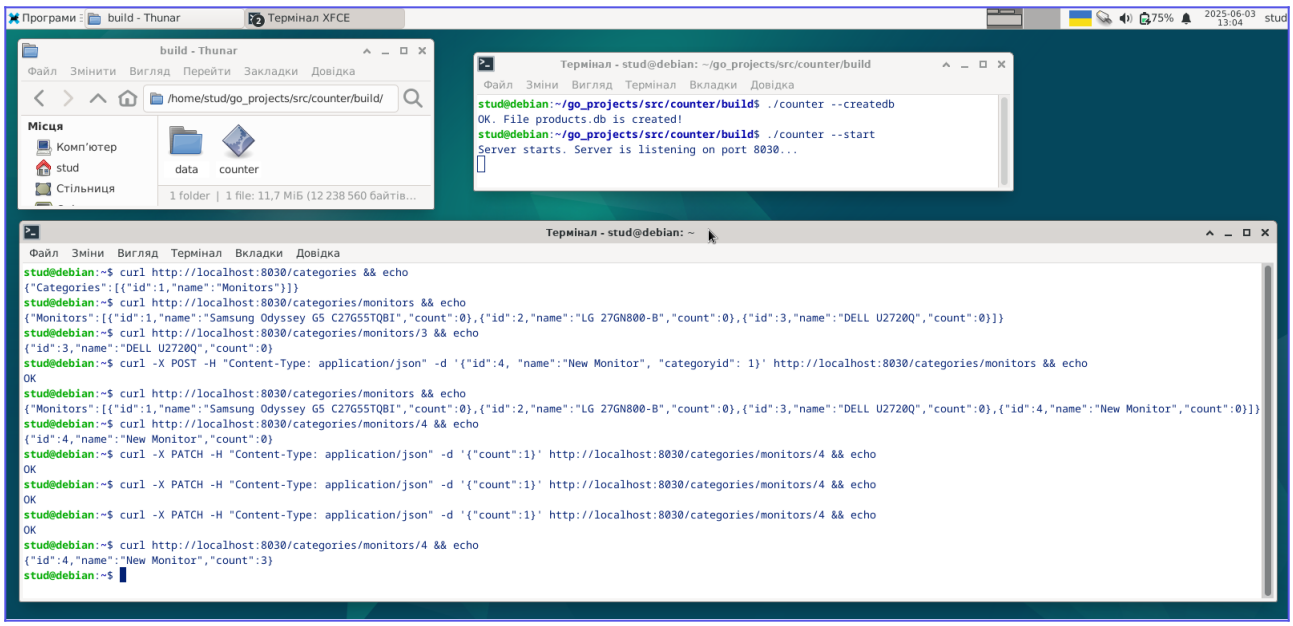


Рис. 3.15. Приклади використання інструмента *curl*

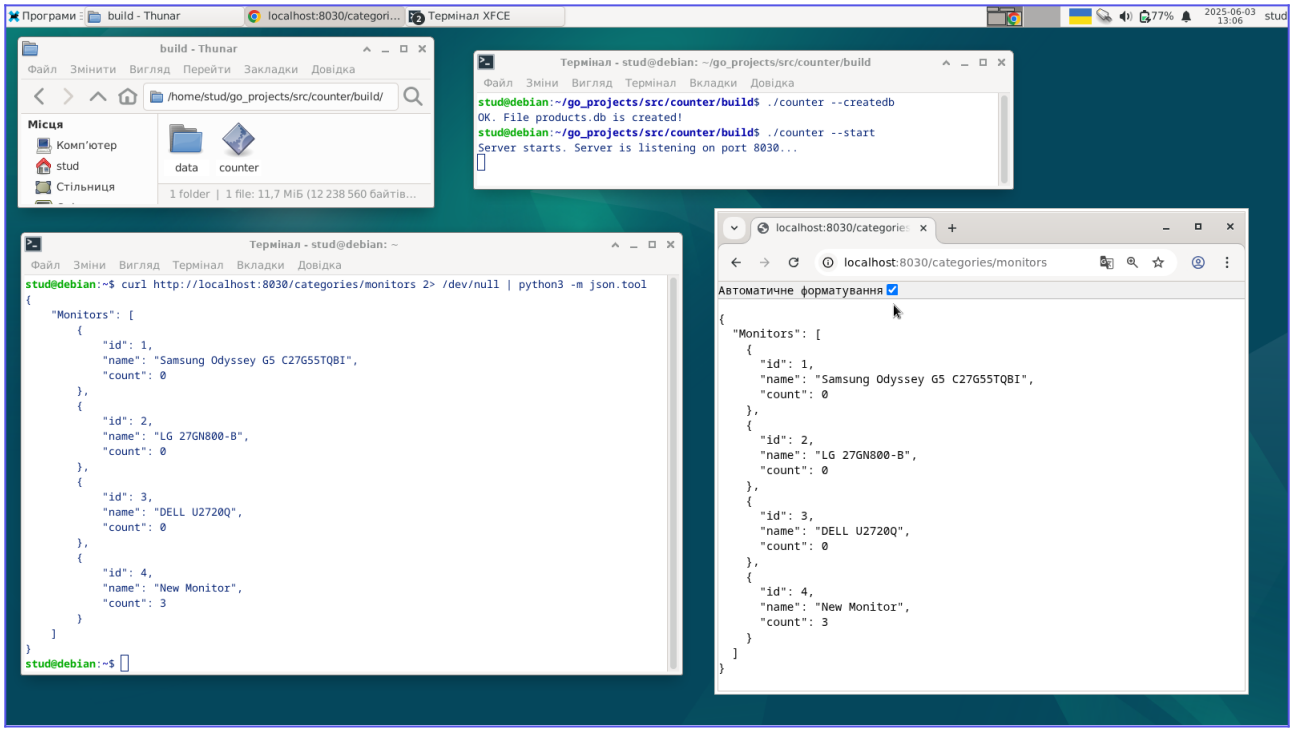


Рис. 3.16. Приклад використання інструмента *curl* разом із інтерпретатором мови *Python* та Web-браузером

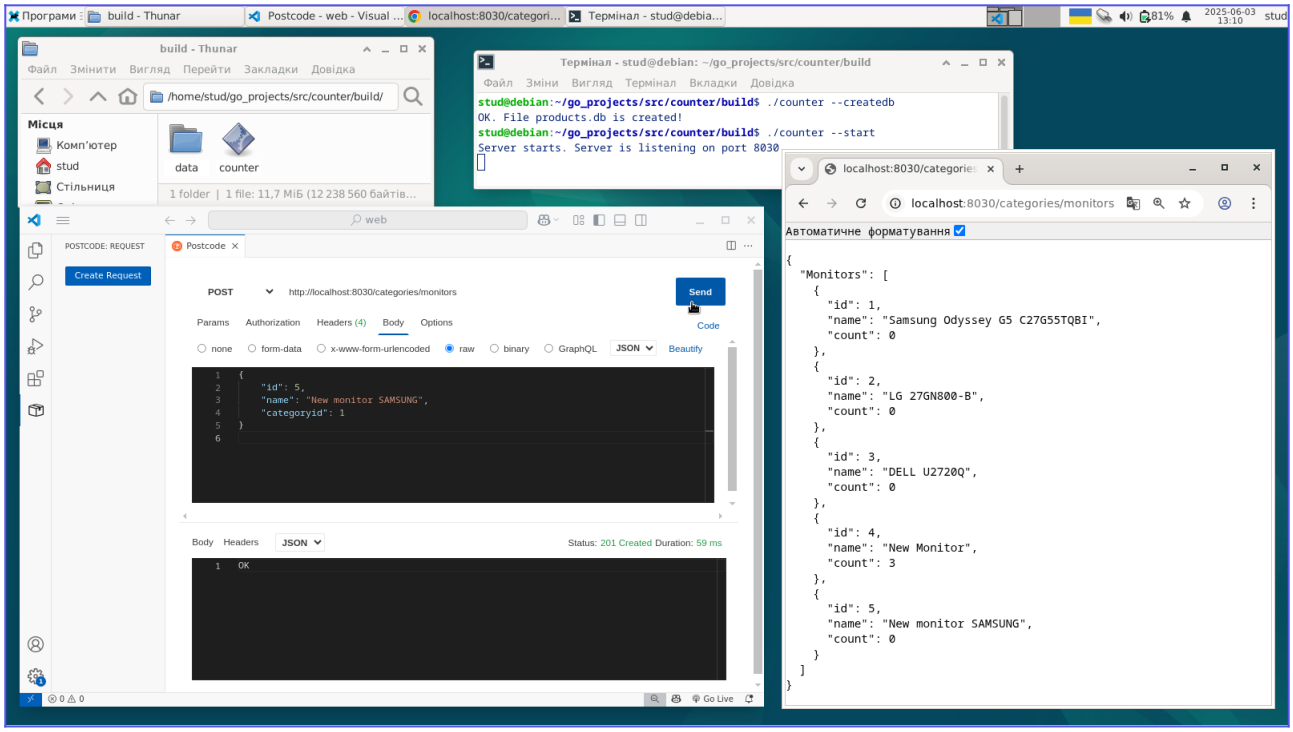


Рис. 3.17. Приклад використання розширення MS Visual Studio Code – Postcode для тестування REST API

При тестуванні засобами *curl* або *Postcode* проблем виникнути не повинно. Але при роботі із браузером можуть бути складнощі, якщо в коді не врахувати техніку роботи із *CORS* (Cross-Origin Resource Sharing).

Рядки 14-16 на рис. 3.8 нададуть можливість дозволити отримати від сервера відповідь через механізм *CORS*. Цей механізм використовує HTTP-заголовки для надання агенту клієнта дозволу на доступ до вказаних ресурсів сервера з джерела, яке відрізняється в поточний момент. Кажуть, що агент користувача робить запит із іншого джерела (cross-origin HTTP request).

В цілях безпеки браузери обмежують cross-origin-запити, які йдуть від скриптів, наприклад, із використанням відомого Fetch API.

Більш детальну інформацію щодо *CORS* можна подивитися в документації за посиланням:

https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/CORS

Зверніть увагу, що в роботі використовується метод PATCH для HTTP-запиту. Згідно документації це призводить до відправлення спочатку до серверу запиту методом OPTIONS, щоб той з'ясував, чи є безпечний PATCH запит. За замовченням будемо вважати, що запит безпечний й тому повернемо код статусу 200 на рядку 61 (рис. 3.8). Як тільки цей попередній запит завершений, буде відправлений запит методом PATCH. Обробка виконується на 65 рядку (рис. 3.8) викликом певної функції `PatchMonitor()`.

Згідно Fetch API та можливостей мови розробки JavaScript, фрагмент коду для тестової HTML-сторінки може виглядати наступним чином:

```
<!doctype html>
<html lang="en">
<body>
  <script>
    async function update_click(id) {
      const url = `http://localhost:8030/categories/monitors/${id}`;
      await fetch(url, {
        method: 'PATCH',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({count: 1})
      })
      .then(response => response.text())
      .then(data => console.log(data))
      .catch(error => console.error(error));
    }
  </script>
  . . .
  <p><a href="https://ek.ua/ua/DELL-U2720Q.htm"
      onclick="update_click(3)">DELL U2720Q</a>
  . . .
</body>
</html>
```

Звісно, що в прикладі замість *localhost* повинне бути залучене реальне доменне ім’я або IP-адреса серверу, на якому розгорнутий Web-сервіс *counter*.

Детальнішу інформацію щодо Fetch API можна подивитися за посиланням:

https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch

Додатково про політики переходу з HTML-сторінки можна почитати за посиланням:

https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Headers/Referrer-Policy

Після завершення процесів кодування, збірки та тестування можна перейти до розміщення серверного додатку *counter* в оточені Ubuntu Server.

Для цього як мінімум можна виконати наступні кроки.

1. На стороні Ubuntu Server створити каталог для розміщення *counter* та зробити власником каталогу, наприклад, користувача `stud`:

```
$ sudo mkdir /opt/counter
$ sudo chown -R stud:stud /opt/counter
```

2. На стороні клієнта з розробки *counter*, встановити утиліти командного рядку *sshpass* та *rsync*. Наприклад, для Debian-подібної ОС використовують команду:

```
$ sudo apt -y install sshpass rsync
```

У разі розробки в оточені іншої ОС, потрібно скористатися альтернативними інструментами або додатковими середовищами (наприклад, для MS Windows – оточенням MSYS2).

3. Зі сторони клієнта з розробки *counter* виконати за допомогою утиліт командного рядку *sshpass* та *rsync* копіювання на Ubuntu Server необхідних файлів проєкту *counter*. Наприклад (IP-адреса серверу наведена в якості прикладу):

```
$ sshpass -p stud \
rsync -avr /home/stud/go_projects/src/counter/build/* \
stud@192.168.119.233:/opt/counter
```

Подані кроки можна описати окремо у відповідному командному скрипті для публікування (деплою).

Завдання

1. Створити програму *counter*, яка розглянута в теоретично-практичній частині роботи, а також додатково html-сторінку для тестування розробленого REST API. Сторінка повинна містити посилання на категорію товарів – монітори, представлені в БД *products*. За допомогою javascript-коду проводиться відсилання через сервер *counter* інформації про клік за певним URL на сторінці.

2. Розширити можливості програми *counter* логуванням у вікно терміналу консолі та у зовнішній файл. Для реалізації цієї задачі створити окремий пакет *counter/internal/logger* із модулем *logger.go*. В ньому створити певні функції, що полегшують задачу логування як у консоль, так і у окремий файл. Файл логування повинен мати назву *counter.log*. Розміщення файлу за замовченням: каталог */var/log*. Для розміщення файлу в каталозі програми, *counter* запускати із додатковою опцією: “`--locallog`”. Для обробки цього аргументу програми створити додаткову функцію в модулі *main.go*.

3. Забезпечити за допомогою утиліт *sshpass* та *rsync* розгортання в оточені Ubuntu Server додатку *counter* в каталозі */opt/counter*. Процес розгортання на сервері описати окремим *bash*-скриптом у складі проєкту *counter* (наприклад,

deploy.sh). Пароль для `stud` передавати в *sshpas* через окремий файл (див. довідку *sshpass*).

У звіті описати текст програмного коду *counter* та гіпертекстову розмітку html-сторінки, навести результати роботи *counter* на стороні Ubuntu Server з логуванням, результати тестування та висновки.

Правила оформлення звіту подані у додатку А.

При захисті роботи володіти знаннями для відповіді на питання викладача за матеріалами лекцій та теоретично-практичної частини роботи.

Робота №4.
Створення власного модуля systemd

Мета роботи: набути практичних навичок щодо автоматизації розгортання власного Web-сервісу в оточені Ubuntu Server із системою ініціалізації *systemd*.

Теоретична частина

systemd – система ініціалізації, яка є в багатьох ОС для розпаралелення та прискорення процесу завантаження служб системи, а також для спрощення процесу обслуговування ОС.

Перевірити присутність *systemd* у системі можна через виклик програми керування нею, яка входить до її складу:

```
systemctl --version
```

systemd використовує цілі (*target*), які нагадують рівні запуску в Unix-подібних та GNU/Linux-сумісних ОС, але працюють дещо інакше. Кожна ціль має назву, яка описує її призначення. Деякі цілі об'єднують в собі запуск всіх служб однієї певної цілі та декількох додаткових сервісів.

У разі використання системи ініціалізації *systemd*, як правило, всі цілі та сервіси розміщуються в каталозі */lib/systemd/system*. В сучасних версіях GNU/Linux-сумісних ОС можна зустріти символьне посилання */lib –> /usr/lib*. Тобто насправді каталог *lib* із *systemd* буде розміщений в підкаталозі */usr*.

Приклади спеціальних цілей, які підміняють рівні запуску:

```
/etc/systemd/system/default.target -> /lib/systemd/system/graphical.target
```

або:

```
/lib/systemd/system/default.target -> graphical.target
```

```
/lib/systemd/system/multi-user.target
/lib/systemd/system/poweroff.target
/lib/systemd/system/rescue.target
/lib/systemd/system/reboot.target
/lib/systemd/system/runlevel0.target -> poweroff.target
/lib/systemd/system/runlevel1.target -> rescue.target
/lib/systemd/system/runlevel2.target -> multi-user.target
/lib/systemd/system/runlevel3.target -> multi-user.target
/lib/systemd/system/runlevel4.target -> multi-user.target
/lib/systemd/system/runlevel5.target -> graphical.target
/lib/systemd/system/runlevel6.target -> reboot.target
```

Для отримання посилання на рівень запуску за замовченням, виконують команду:

```
systemctl get-default
```

А для встановлення іншого рівня виконують команду:

```
sudo systemctl set-default ім'я_цілі.target
```

Наприклад:

```
sudo systemctl set-default multi-user.target
```

Для `set-default` заборонено використовувати цілі, які призводять до перезавантаження або вимкнення ОС!!!

Керування сервісами *systemd* здійснюється певною множиною команд із складу *systemctl*. Найчастіше з них використовують:

– `systemctl enable` *`ім'я_служби`*`.service` – вмикає запуск служби після перезавантаження;

– `systemctl disable` *`ім'я_служби`*`.service` – вимикає запуск служби після перезавантаження;

– `systemctl is-enabled` *`ім'я_служби`* – перевіряє, буде чи ні служба запущена після перезавантаження;

– `systemctl start` *`ім'я_служби`*`.service` – запускає службу;

– `systemctl stop` *`ім'я_служби`*`.service` – зупиняє службу;

– `systemctl restart` *`ім'я_служби`*`.service` – перезапускає службу;

– `systemctl reload` *`ім'я_служби`* – перезавантажує конфігураційний файл служби, без переривання її роботи (якщо це підтримується службою);

– `systemctl status` *`ім'я_служби`*`.service` – виводить інформацію про стан служби.

Коли потрібно підключити певну службу (service-модуль в системі присутній), то виконують послідовність команд (на прикладі служби друку *cups.service*):

```
sudo systemctl enable cups.service
sudo systemctl start cups.service
```

Якщо виникають проблеми при роботі служби, то для діагностики виконують команди:

```
systemctl status cups.service
journalctl -xeu cups
```

Для виведення списку всіх доступних служб виконують команду:

```
systemctl list-units --type service
```

Для виведення списку всіх служб, що запущені, виконують команди:

```
systemctl list-units --type service --state running
```

або:

```
systemctl | grep service | grep running | awk '{print $1}' | less
```

або іншу.

Для виведення послідовності запуску служб із залежностями від поточної (*default.target*) цілі, використовують команду:

```
systemctl list-dependencies
```

або для певної цілі (наприклад, для цілі `timers.target`):

```
systemctl list-dependencies timers.target
```

Модулі systemd

Основною одиницею управління в *systemd* є модуль (юніт, unit). Одним із різновидів модулів є служби (сервіси або демони).

Зазвичай всі модулі *systemd* розміщені в каталогах:
– */lib/systemd/system* або */usr/lib/systemd/system* – встановлені з репозиторіїв;
– */etc/systemd/system* – основні посилання на каталог із встановленими модулями та власними модулями користувача *root*;
– */run/systemd/system* – модулі, які були створені в процесі запуску ОС.

Якщо необхідно створити підтримку запуску служби в *systemd*, яка відсутня по якимось причинам в каталозі */lib/systemd/system* або подібному, або у репозиторію, то створюють власний модуль.

Наприклад, нехай потрібно сотворити *systemd*-модуль для деякої служби *my_staff*, який буде запускати деякі командні файли. Обов'язковим при цьому є те, що служба потребує наявності цілі *network.target* Тоді можна сформувати модуль із таким вмістом:

```
# file /etc/systemd/system/my_staff.service
[Unit]
Description=Run my stuff
Requires=network.target
After=network.target
```

```
[Service]
Type=oneshot
RemainAfterExit=True
ExecStart=/some/script --with-some-parameters
ExecStart=/some/other/script

[Install]
WantedBy=multi-user.target
```

Де `Requires` – вказує ціль, після якої необхідно запустити сервіс.

`After` – означає, що мережа повинна бути повністю запущена до моменту старту.

`WantedBy` – ціль, для якої запускається сервіс.

`Type` – тип служби, наприклад, `oneshot` – скрипт виконує своє завдання та завершується.

Часто також потрібно вказати каталог, який буде вважатися робочим за замовчуванням для модуля, що запускається. Для цього існує ключове слово `WorkingDirectory` в частині `Service`. Наприклад:

```
WorkingDirectory=/myservice/data
```

Якщо потрібно, то додатково в частині `Service` можна вказати користувача або групу, від імені якого буде стартувати власний модуль. Наприклад:

```
User=admin
Group=admin
```

Якщо власна служба з якихось причин перестає працювати, то можна налаштувати *systemd* на її автоматичне перезавантаження (в частині `Service`):

```
PIDFile=/run/my_staff.pid
Restart=always
```

Для виконання скриптів після запуску графічного дисплейного менеджеру, в рядку `After`, можна використовувати *systemd-user-sessions.service*.

Для запуску власних скриптів можна також використовувати файл */etc/rc.local* (або */etc/rc.d/rc.local*, або подібний в залежності від різновиду GNU/Linux-сумісної ОС). Однак слід враховувати специфіку запуску цього скрипту в *systemd*. Цей файл не має ніяких залежностей від інших цілей. Це означає, що з врахуванням паралельного завантаження, скрипт, який вказаний у *rc.local*, може бути запущений дуже рано.

Для підтримки запуску *rc.local*, його, за відсутності, треба створити, надати атрибут на виконання, та забезпечити виконання файлу служби */lib/systemd/system/rc-local.service*.

Якщо в системі опису модулів *systemd* виникли зміни та є потреба у перезавантажені всіх модулів *systemd*, то це виконується командою:

```
sudo systemctl daemon-reload
```

Для виведення у вікно терміналу консолі списку модулів *systemd*, завантаження яких провести не вдалося, виконують команду:

```
systemctl --failed
```

Розміщення репозиторію проєкту на сервері GitLab

Різноманітні зміни можна зберегти у віддалений репозиторій проєкту на *GitLab*-сервер. Для коректної роботи із встановленим в роботі №2 *GitLab*-сервером, на клієнті (наприклад, ОС Debian GNU/Linux) виконують команди (IP-адреса 192.168.119.190 у прикладі далі – це IP-адреса віртуальної машини із запущеним сервером *GitLab*; детально про *GitLab Runner* мова буде йти у роботі №5):

```
$ sudo -s
# echo '192.168.119.190 gitlab.example.com' >> /etc/hosts
# mkdir -p /etc/gitlab-runner/certs
# openssl s_client -showcerts -connect gitlab.example.com:443 \
-servername gitlab.example.com < /dev/null 2> /dev/null | \
openssl x509 -outform PEM > /etc/gitlab-runner/certs/gitlab.example.com.crt
# cp /etc/gitlab-runner/certs/gitlab.example.com.crt /etc/ssl/certs/
```

Примітка 4.1: додавання *GitLab*-хосту із специфічним SSH-портом та подальше клонування створеного репозиторію проєкту із сервера можна виконати, наприклад, командами (під користувачем *stud*):

```
$ ssh -T git@gitlab.example.com -p 2025
$ git clone ssh://git@gitlab.example.com:2025/stud/counter.git
```

Примітка 4.2: для додавання в репозиторій проєкту файлу *.gitignore* можна скористатися командами:

```
$ cd ./counter
$ wget -O ./.gitignore \
https://raw.githubusercontent.com/github/gitignore/refs/heads/main/Go.gitignore
$ echo 'build' >> ./.gitignore
```

Завдання

Використовуючи теоретичний матеріал лекцій та поточної роботи, виконати наступні пункти завдання.

1. Створити на розгорнутому в роботі №2 сервері *GitLab* під користувачем *stud* новий public-проєкт *counter*.

2. Використовуючи git-команди, додати в репозиторій нового проєкту код основної програми з роботи №3 (без додавання html-сторінки, *bash*-скриптів, модуля *systemd*). Всі дії робити в *main*-гілці. Позначити останній коміт в гілці тегом `release1`.

3. За допомогою git-команд, створити нову гілку *version2*, в яку додати зміни, що стосуються розробленої html-сторінки в роботі №3.

4. За допомогою git-команд, створити нову гілку *version3* від гілки *version2*, в яку додати *bash*-скрипт розгортання створеної програми. Деплой програми повинен здійснюватися в каталог */opt/counter* серверу Ubuntu. Розгортання html-сторінки повинно відбуватися в оточення Web-серверу, наприклад, *Lighttpd*.

5. Створити гілку *version4* від гілки *version3*, в якій повинен міститися модуль *systemd* для запуску розробленого Web-сервісу (файли, які повинні мати відношення до цього, можна розмістити у підкаталозі */counter/systemd* проєкту).

6. Виконати злиття гілок таким чином, щоб остаточні зміни вірно потрапили в гілку *main* проєкту. Позначити останній коміт в гілці тегом `release2`. Створені гілки не видаляти.

7. Продемонструвати клонування, збірку та встановлення проєкту *counter* (`release2`) із клієнту на Ubuntu Server, успішне встановлення служби на базі *systemd* та її функціонування.

У звіт до роботи включити тексти source-кодів всіх модулів проєкту, знімки з екрану графа змін в репозиторії проєкту *counter*, результати функціонування служби *counter* в *systemd*, вірної роботи API та html-сторінки.

Правила оформлення звіту подані в додатку А.

При захисті роботи володіти знаннями для відповіді на питання викладача за матеріалами лекцій та виконаної роботи.

Робота №5.
Робота з GitLab CI/CD pipelines

Мета роботи: опанувати процес створення та використання конвеєру CI/CD на платформі *GitLab* із залученням засобу *GitLab Runner* і *shell*-середовищем для виконання.

Теоретично-практична частина

CI/CD (**C**ontinuous **I**ntegration / **C**ontinuous **D**eployment або **D**elivery) – це безперервний метод розробки ПЗ, при якому виконується процес постійного створення, тестування, розгортання та контролю ітеративних змін коду[1].

Цей ітеративний процес допомагає зменшити ймовірність розробки нового коду на основі версій, які містять помилки або є недосконалими. *GitLab CI/CD* може виявляти помилки на ранніх етапах циклу розробки та допомагати гарантувати, що код, розгорнутий у продакшені, відповідає встановленим стандартам коду.

Для використання *GitLab CI/CD* потрібно забезпечити наявність у корені репозиторію git-проєкту спеціального файлу `.gitlab-ci.yml`, який є файлом конфігурації CI/CD та повинен містити певні інструкції по виконанню CI/CD в YAML-розмітці (рис. 5.1).

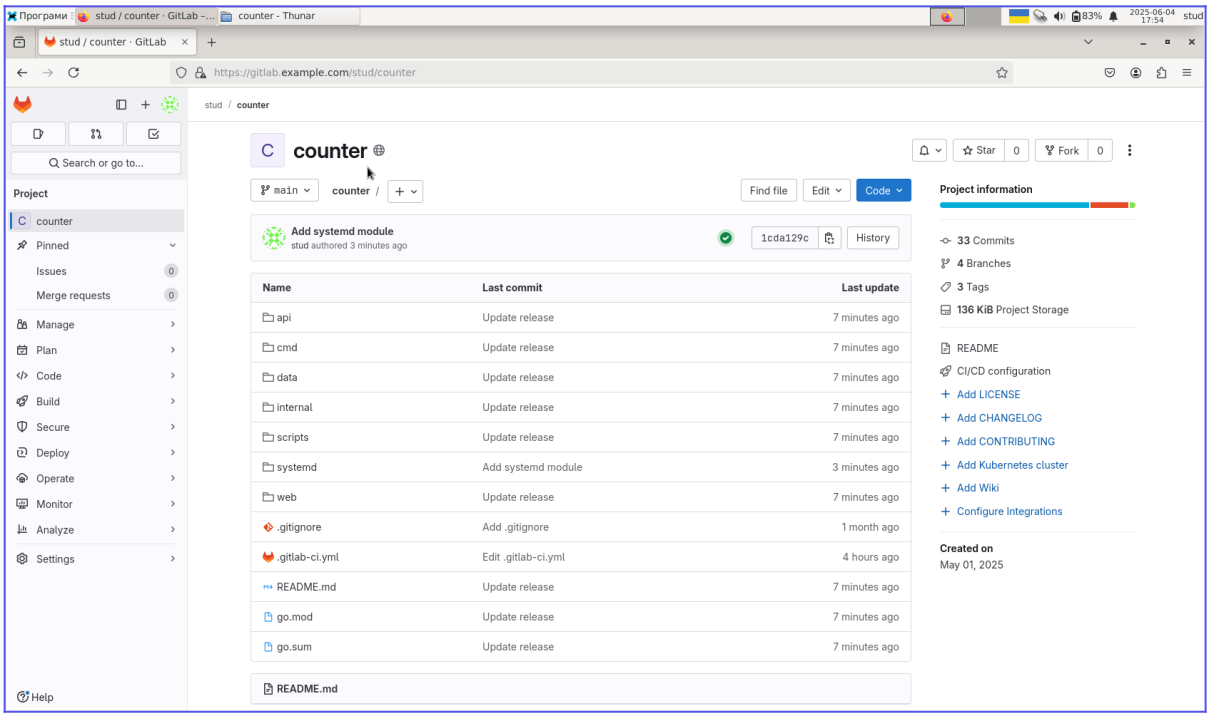


Рис. 5.1. Приклад розміщення файлу *.gitlab-ci.yml* в репозиторії проєкту

[1] В теоретично-практичній частині роботи використана офіційна документація проєкту *GitLab*: https://docs.gitlab.com/ci/.

Фундаментальним компонентом *GitLab CI/CD* є *конвеєри* – *GitLab CI/CD pipelines*. Конвеєри налаштовуються саме у файлі *.gitlab-ci.yml* за допомогою ключових слів YAML-розмітки.

Конвеєри можуть запускатися автоматично для певних подій, таких як перенесення до гілки, створення запиту на злиття або за розкладом. За потреби також можна запускати конвеєри вручну.

Конвеєри складаються з таких складових.

1. *Глобальних ключових YAML-слів* (*global YAML keywords*), які контролюють загальну поведінку конвеєрів проєктів.

2. *Завдань* (*jobs*), які містять команди для виконання завдання. Наприклад, завдання може компілювати (compile), тестувати (test) або розгортати/публікувати код (deploy). Завдання виконуються за допомогою *виконавців* (*runners, GitLab Runner*) незалежно одне від одного.

3. *Етапів* (*stages*), які визначають, як групувати завдання разом. **Етапи виконуються послідовно, тоді як завдання на етапі виконуються паралельно.** Так, наприклад, ранній етап може мати завдання, які компілюють та обробляють код, тоді як пізніші етапи можуть мати завдання, які тестують та розгортають код. Якщо всі завдання на етапі завершуються успішно, конвеєр переходить до наступного етапу. Якщо будь-яке завдання на етапі завершується невдачею, наступний етап (зазвичай) не виконується, і конвеєр завершується раніше.

Невеликий конвеєр може складатися, наприклад, з трьох етапів, що виконуються в такому порядку.

1. Етап збірки (*build stage*) із завданням під назвою компіляція (compile), яке компілює код проєкту.

2. Етап тестування (*test stage*) з декількома завданнями з назвами, що містять слово test і виконують різні тести коду. Ці тести будуть виконуватися лише за умови успішного завершення завдання компіляції.

3. Етап розгортання (*deploy stage*) із завданням з назвою, наприклад: deploy-to-production. Це завдання буде виконуватися лише за умови успішного запуску та завершення всіх завдань на етапі тестування.

GitLab Runner запускає завдання CI/CD, які описані для конвеєра. Цей засіб може працювати як окремий бінарний файл і не має вимог до певної мови програмування.

З міркувань безпеки та підвищення продуктивності, розробники *GitLab* радять виконувати встановлення *GitLab Runner* в ОС, під якою не працює сервер *GitLab*.

Вказівки та поради щодо встановлення *GitLab Runner* надані за посиланням: https://docs.gitlab.com/runner/install/. Зауважте, що ця технологія активно використовує контейнеризацію, тому до встановлення *GitLab Runner* радять розгорнути в оточені обраної ОС інструмент *docker* (встановлення в

оточені ОС Debian GNU/Linux було розглянуто в роботі №2) та подбати про встановлений локально інструментарій *Git*. Після встановлення *GitLab Runne*r його реєструють, використовуючи поради, які надані за посиланням: https://docs.gitlab.com/runner/register/.

Розглянемо мінімальні кроки для встановлення *GitLab Runner* в клієнтському оточені ОС Debian GNU/Linux. Згідно із документацією для цього різновиду ОС виконуємо у вікні терміналу консолі команди (IP-адреса 192.168.119.190 у прикладі далі – це IP-адреса віртуальної машини із запущеним сервером *GitLab*):

```
$ sudo -s
# curl -L "https://packages.gitlab.com/install/\
repositories/runner/gitlab-runner/script.deb.sh" | bash
# apt update
# apt -y install gitlab-runner
# echo '192.168.119.190 gitlab.example.com' >> /etc/hosts
# mkdir -p /etc/gitlab-runner/certs
# openssl s_client -showcerts -connect gitlab.example.com:443 \
-servername gitlab.example.com < /dev/null 2>/dev/null | \
openssl x509 -outform PEM > /etc/gitlab-runner/certs/gitlab.example.com.crt
# cp /etc/gitlab-runner/certs/gitlab.example.com.crt /etc/ssl/certs/
```

Примітка 5.1: за замовченням сертифікат безпеки у сервера *GitLab* дійсний протягом одного місяця. Якщо термін сертифікату сплив, то його обов'язково потрібно згенерувати знову. Це важливо, оскільки *GitLab Runner* не буде нормально працювати із застарілим сертифікатом безпеки. Існує кілька способів генерування сертифікату безпеки сервера *GitLab*. Один з них полягає у виконанні в оточені контейнера із запущеним сервером *GitLab* наступної послідовності команд.

```
$ cd /home/stud/gitlab
$ sudo -s
# docker compose up -d
```

Очікуємо на старт сервера *GitLab* та далі продовжуємо виконувати команди (де `ID_GITLAB_CONTAINER` – це ідентифікатор docker-контейнера із сервером *GitLab*):

```
# docker ps
# docker exec -it ID_GITLAB_CONTAINER /bin/bash
mkdir /root/newcert && cd /root/newcert
openssl genrsa -out ca.key 4096
openssl req -x509 -new -nodes -key ca.key \
-sha256 -days 1024 -out ca.crt -subj '/CN=gitlab.example.com'
openssl req -new -nodes \
-out gitlab.example.com.csr -newkey rsa:4096 \
-keyout gitlab.example.com.key -subj '/CN=gitlab.example.com'
```

```
cat > run.extfile << EOF
basicConstraints=critical,CA:TRUE
keyUsage = digitalSignature, nonRepudiation, keyEncipherment, dataEncipherment
subjectAltName = @alt_names
[alt_names]
DNS.1 = gitlab.example.com
EOF
openssl x509 -req -in gitlab.example.com.csr \
-CA ca.crt -CAkey ca.key -CAcreateserial \
-out gitlab.example.com.crt -days 365 -sha256 -extfile run.extfile
```

Далі вносять правки у файл */etc/gitlab/gitlab.rb* – прибирають коментарі та формують такі рядки:

```
external_url "https://gitlab.example.com"
nginx['redirect_http_to_https'] = true
letsencrypt['enable'] = false
```

Після цього розміщують key- та crt-файли у *ssl*-каталозі *GitLab*, попередньо подивившись, що каталог порожній:

```
# cp -R /etc/gitlab/ssl /etc/gitlab/ssl.old
# rm -f /etc/gitlab/ssl/*
# cp -f gitlab.example.com.{key,crt} /etc/gitlab/ssl/
```

Далі виконують реконфігурування серверу *GitLab* та його перезавантаження:

```
# gitlab-ctl reconfigure
# exit
# docker compose down
# docker compose up -d
```

Таким чином, буде згенерований самопідписаний сертифікат безпеки терміном на один рік.

Примітка 5.2: перевірити дати дії сертифікату безпеки сервера *GitLab* можна, наприклад, командою (або, як альтернатива, через Web-браузер):

```
echo "Q" | openssl s_client -connect gitlab.example.com:443 \
-showcerts 2> /dev/null | openssl x509 -noout -dates
```

Примітка 5.3: після оновлення сертифікату безпеки на сервері *GitLab*, потрібно оновити файли у каталозі */etc/gitlab-runner/certs* та */etc/ssl/certs*, повторивши описану вище послідовність команд на клієнті.

Відкрити Web-браузер та здійснити вхід під користувачем *stud* на сервер за посиланням: https://gitlab.example.com/.

В оточенні користувача *stud* обрати меню *Preferences* та далі пункт *SSH Keys*. Перевірити наявність SSH-ключа з оточення користувача той ОС, в якій буде запускатися *GitLab Runner*. Якщо ключ відсутній, то згенерувати його на стороні клієнта та отримати командами:

```
$ ssh-keygen -o -t ed25519
$ cat /home/stud/.ssh/id_ed25519.pub
```

Додати SSH-ключ в налаштування оточення користувача *stud* на сервері *GitLab* (якщо цього не було зроблено раніше при виконанні попередньої роботи).

Перейти до проєкту *counter*. В меню *Settings* обрати пункт меню *CI/CD*. На сторінці розкрити частину *Runners* та натиснути на кнопку *New project runner*. В поле тегів *Tags* вказати, наприклад, *counter* та встановити опцію *Run untagged jobs*. В поле *Runner description* вказати, наприклад, *ForCounter* та натиснути на кнопку *Create runner*. Після вдалого створення *GitLab* виведе на сторінці команду реєстрації *GitLab Runner* на клієнті. Потрібно обов'язково зберегти токен аутентифікації (*authentication token*)!

Після цих дій знову перейдіть до відкритої *root*-консолі та введіть команду:

```
# gitlab-runner register
```

Далі в інтерактивному режимі дають відповіді в процесі реєстрації (рис. 5.2), а саме:

– URL серверу *GitLab*: https://gitlab.example.com/;

– токен для реєстрації (в нових версіях *GitLab* – це токен аутентифікації): вводять збережений вище токен (або копіюють через інтерфейс – рис. 5.2);

– ім'я виконавця (*name for the runner* або *description*): наприклад, *ForCounter*;

– теги: наприклад, *counter*;

– середовище для виконання: наприклад, *shell*.

Якщо все вище було вірно налаштовано та виконано, то на цьому реєстрація *GitLab Runner* завершується. У підсумку буде сформовані спеціальні конфігураційні встановлення у файлі */etc/gitlab-runner/config.toml*. Файл можна продублювати й у користувача *stud*.

```
$ sudo -s
# cp /etc/gitlab-runner/config.toml /home/stud/.gitlab-runner/
# chown stud:stud /home/stud/.gitlab-runner/config.toml
```

Після внесення та збереження змін, перезавантажте відповідну службу:

```
sudo systemctl resrart gitlab-runner
```

Коли всі основні налаштування та реєстрація *GitLab Runner* проведені, можна перейти до етапу оформлення вмісту файлу *.gitlab-ci.yml*. Приклади шаблонів цього файлу для різних мов розробки та технологій можна знайти за посиланням (розробники *GitLab* оголосили цю частину проєкту, як *deprecated*, оскільки не будуть більше її оновлювати):

https://gitlab.com/gitlab-org/gitlab/-/tree/master/lib/gitlab/ci/templates

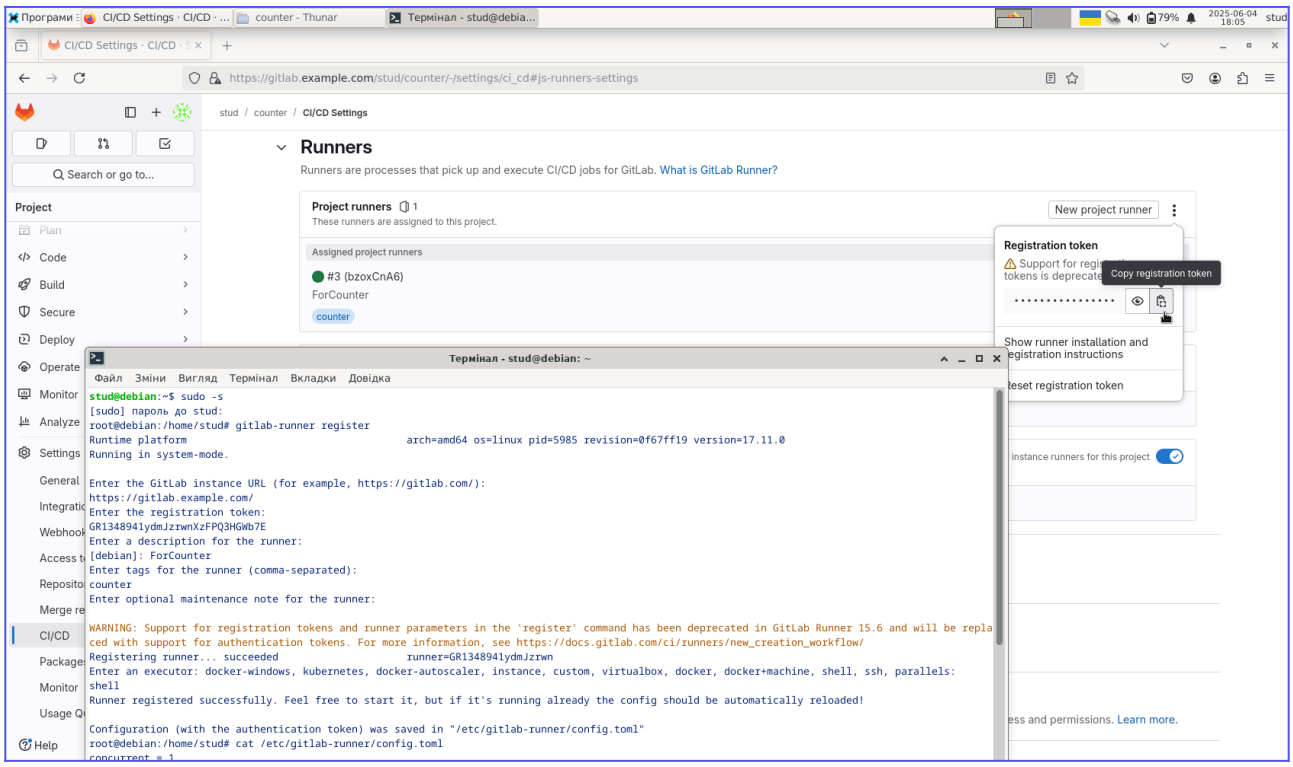


Рис. 5.2. Реєстрація *GitLab Runner* на клієнті в ОС Debian GNU/Linux

Розглянемо більш детальніше приклад побудови цього файлу. Оформлення можна почати із переліку *pipeline*-етапів (stages). Типовими назвами етапів є `build, test, deploy`:

```
stages:
  - build
  - test
  - deploy
```

Оскільки *pipeline* містить завдання (*jobs*), в *.gitlab-ci.yml* це записують, наприклад, наступним чином:

```
build-job:
test-job1:
test-job2:
deploy-prod:
```

Тобто `build-job, test-job1, test-job2, deploy-prod` – це приклади назв завдань. Кожне завдання повинно мати відношення до певного етапу. Тому після зазначення назв завдань, прописують назви етапів. Наприклад:

```
stages:
  - build
  - test
  - deploy

build-job:
  stage: build

test-job1:
  stage: test

test-job2:
  stage: test

deploy-prod:
  stage: deploy
```

Таким чином бачимо, що, наприклад, до етапу `test` мають відношення два завдання: `test-job1` та `test-job2`.

У кожного завдання далі потрібно вказати, як мінімум, частину `script`, в якій прописати конкретну команду чи команди для виконання. Наприклад:

```
stages:
  - build
  - test
  - deploy

build-job:
  stage: build
  script: echo "Execute build process..."

test-job1:
  stage: test
  script:
    - echo "Execute test-job1..."
    - echo "Start test-job1..."

test-job2:
  stage: test
  script:
    - echo "Execute test-job2..."
    - echo "Start test-job2..."

deploy-prod:
  stage: deploy
  script:
    - echo "Execute deploy process..."
    - echo "Deploy files to destination..."
```

При реєстрації *GitLab Runner* вказувався в якості середовища виконання – *shell*. Тому саме в командній оболонці для користувача *gitlab-runner* будуть викликані вказані команди.

Після того, як зміни у *.gitlab-ci.yml* внесені, можна перейти до коміту. Зміни можна робити в різних оточеннях *GitLab*, наприклад, у спеціалізованому інструменті *Pipeline Editor*. Після оформлення змін, можна перейти на закладку *Validate* та скористатися перевіркою сформованого файлу, натиснувши на кнопку *Validate pipeline*. Після успішної валідації можна виконати коміт.

Якщо все було зроблено вірно, то *GitLab* переходить до виконання нового *pipeline*-проєкту. При виконанні завдань (jobs), будуть показані відповідні статуси (рис. 5.3). Їх можна завжди подивитися, обравши пункт меню *Pipelines* в режимі роботи з репозиторієм проєкту.

При обранні певного статусу виконання, робиться перехід до процесу виконання *pipeline* і у графічному вигляді можна подивитися на поточний стан виконання завдань (рис. 5.4).

Після обрання певного завдання виконується перехід до детальних логів виконання цього завдання, або до помилок та зауважень, якщо такі будуть (рис. 5.5).

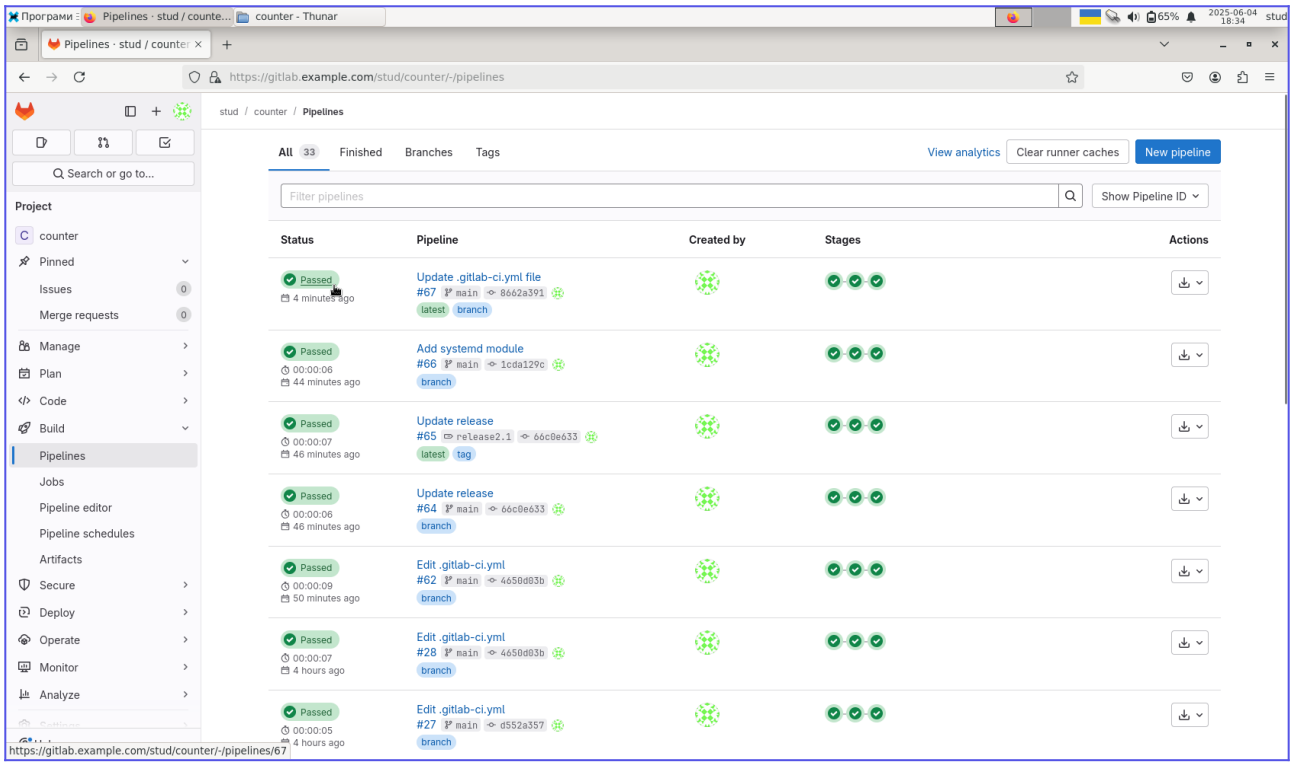


Рис. 5.3. Перелік створених CI/CD конвеєрів та їх статуси виконання

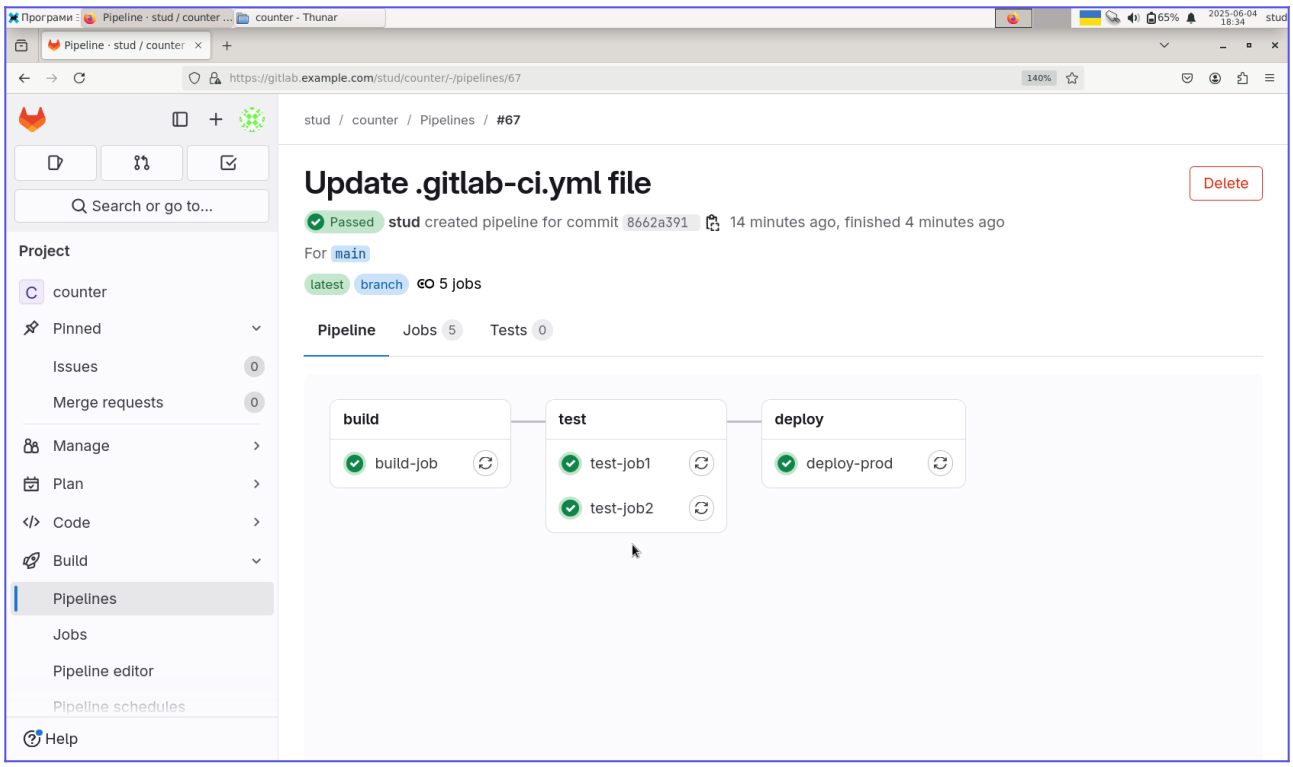


Рис. 5.4. Розгорнута діаграма зі статусами виконання завдань етапів обраного CI/CD конвеєра

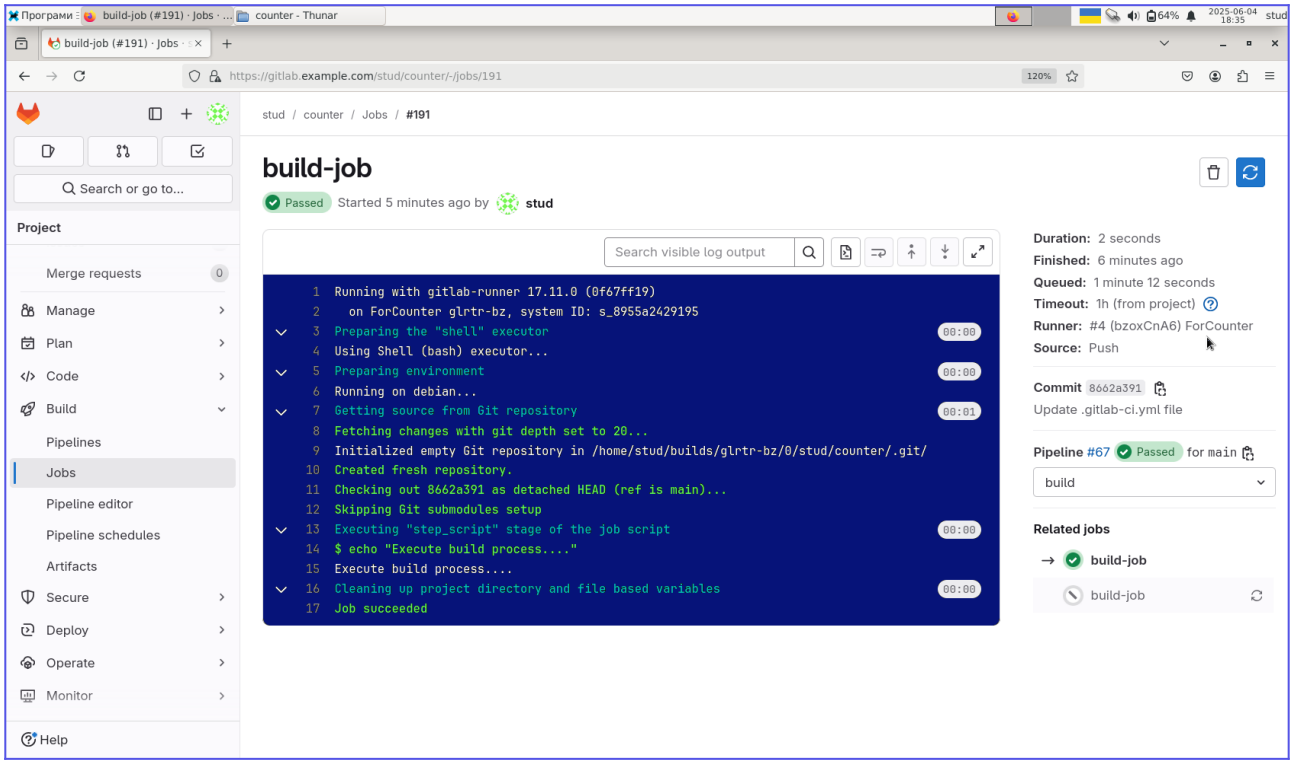


Рис. 5.5. Лог виконання обраного завдання та його статус

Як побачити, де буде виконуватися та в якому поточному каталозі завдання. Самий простий варіант – це прописати певні команди в `script`. Наприклад:

```
. . .
script:
  - echo "Execute build process..."
  - echo $USER
  - ls -l
  - pwd
. . .
```

Таким чином, у лозі виконання відповідного завдання можна побачити результат – рис. 5.6.

Тобто бачимо, що виконання команд завдання йде в командному процесорі від користувача *gitlab-runner*. В його домашньому каталозі буде створений підкаталог *builds* із тимчасовим підкаталогом для обслуговування конвеєру проєкту користувача *stud*. В ньому буде розміщений клонований репозиторій проєкту *counter* (рис 5.6).

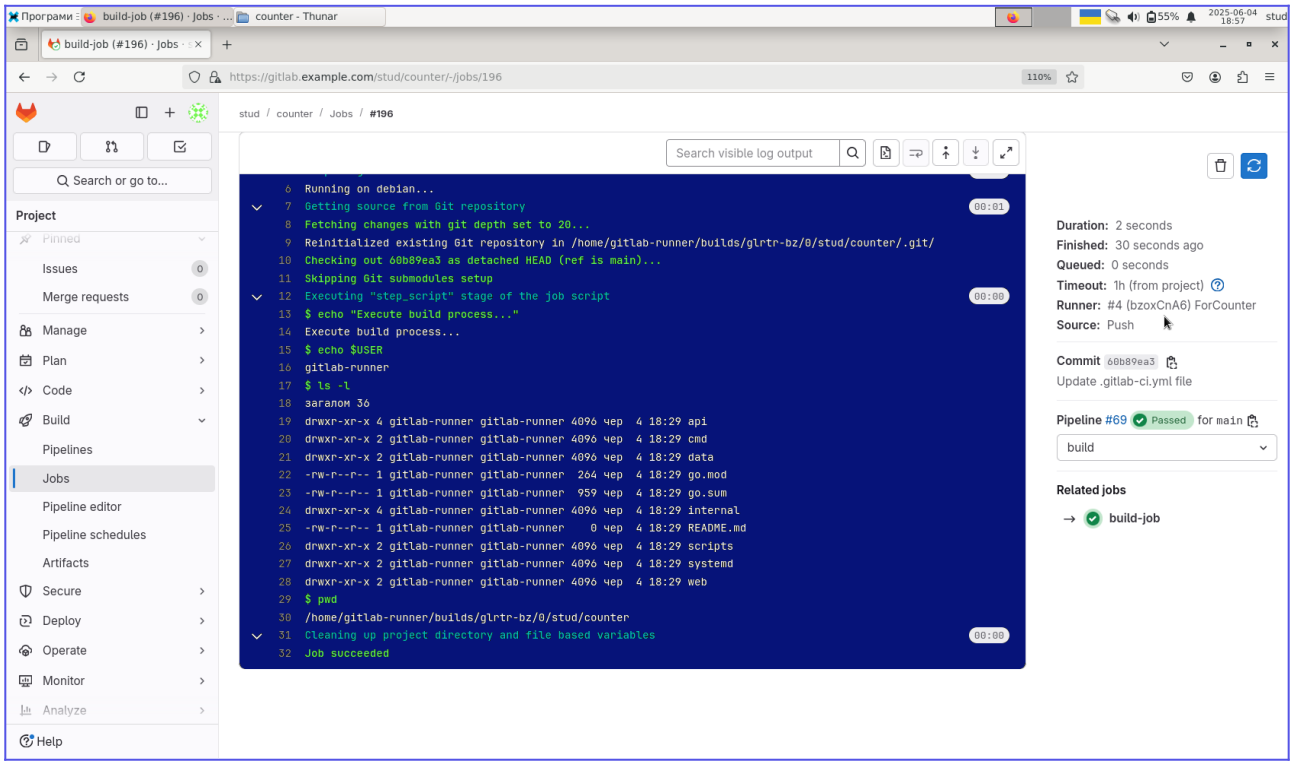


Рис. 5.6. Результат виконання команд завдання для перевірки розміщення файлів проєкту в оточені користувача *gitlab-runner*

Якщо інструменти та бібліотека Go були раніше в роботі № 3 встановлені в каталог */opt/go*, то тоді для збірки можна зробити, наприклад, таке завдання:

```
. . .
build-job:
  stage: build
  script:
    - echo "Execute build process..."
    - export GOROOT=/opt/go
    - export PATH=$PATH:$GOROOT/bin
    - go version
    - ./scripts/build.sh
. . .
```

Таким чином, компілятор мови Go стане також доступним для користувача *gitlab-runner* (якщо встановлення Go проводилося за описом із роботи №3) та в подальшому буде виконаний *bash*-скрипт збірки проєкту.

Завдання

1. Встановити та налаштувати *GitLab Runner* на клієнті, де йде розробка Web-сервісу *counter*.

2. Створити конвеєр CI/CD з трьох етапів: build, test, deploy. Кожний з етапів повинен містити по одному завданню, яке виконує відповідні цим етапам команди збірки та тестування розробленого Web-сервісу *counter* і подальше розгортання необхідних файлів проєкту в оточення Ubuntu Server (див. рис. Б.1 з додатку Б).

3. Запропонувати четвертий етап конвеєру – post-deploy, який повинен виконуватися тільки після вдалого виконання етапу deploy (залучити YAML-слово `needs` – https://docs.gitlab.com/ci/yaml/#needs). Запропонувати на цьому етапі завдання, яке буде тестувати вірну роботу API розгорнутого Web-сервісу на базі Bash/Python та утиліти *curl*.

У звіт до роботи включити вміст файлу налаштування конвеєру CI/CD, вміст використаних додаткових скриптів та результати роботи завдань створеного конвеєру.

Правила оформлення звіту подані в додатку А.

При захисті роботи володіти знаннями для відповіді на питання викладача за матеріалами лекцій та виконаної роботи.

Критерії оцінювання виконання робіт

Виконання завдань, які описані в практичних частинах робіт, та захист результатів, що подані здобувачем у звітах, є обов'язковим.

В залежності від складності практичної частини роботи, на виконання завдань відводиться певна кількість годин, обсягом, що представлений в робочій програмі дисципліни.

Після виконання практичної частини роботи, здобувач складає звіт, в якому повинен описати хід виконання завдання та надати висновки щодо отриманих результатів роботи. Вимоги по оформленню звітності подані в додатку А.

Захист звіту з виконаної роботи проводиться під час проведення заняття у встановлений розкладом основний час або у додатковий час, якій погоджений із викладачем.

Узагальнені критерії оцінки виконаної роботи наведені у таблиці:

Критерії оцінювання виконання та захисту роботи здобувачем	*Рейтингова оцінка*	*Інституційна оцінка*
Результати виконання роботи вірні згідно вимог, описаних в завданні та наданих викладачем в ході лекційних занять. Звіт оформлений згідно вимог, описаних в додатку А. На всі поставлені питання викладача при захисті роботи надані вірні відповіді. Робота захищена.	90 – 100	відмінно
Результати виконання роботи вірні згідно вимог, описаних в завданні та наданих викладачем в ході лекційних занять. Звіт оформлений згідно вимог, описаних в додатку А. На 80% поставлених питань викладача при захисті роботи були надані вірні відповіді. Робота захищена.	80 – 89	добре
Результати виконання роботи вірні згідно вимог, описаних в завданні та наданих викладачем в ході лекційних занять. Звіт оформлений згідно вимог, описаних в додатку А. На 70% поставлених питань викладача при захисті роботи були надані вірні відповіді. Робота захищена.	74 – 79	
Результати виконання роботи містять помилки. Є відхилення від поставлених вимог оформлення звітності. Під час захисту здобувач надав 50% вірних відповідей на поставлені питання викладача. Робота захищена.	60 – 73	задовільно
Результати виконання роботи невірні або звіт відсутній, або здобувач не може під час захисту результатів роботи відповісти на питання викладача. Робота не була захищена.	0 – 59	незадовільно

Перелік рекомендованих джерел

1. Richard Petersen. Ubuntu 24.04 LTS Server: Administration and Reference. – Surfing Turtle Press, 2025. – 720 p. ISBN-10: 1949857549, ISBN-13: 978-1949857542.

2. Richard Blum, Christine Bresnahan. Linux Command Line and Shell Scripting Bible, 4th Edition. – Wiley, 2021. – 832 p. ISBN-10: 1119700914, ISBN-13: 978-1119700913.

3. Noah Gift. Python for DevOps: Learn Ruthlessly Effective Automation, 1st Edition / Noah Gift, Kennedy Behrman, Alfredo Deza, Grig Gheorghiu. – O'Reilly Media, 2020. – 506 p. ISBN-10: 149205769X, ISBN-13: 978-1492057697.

4. Jon Bodner. Learning Go: An Idiomatic Approach to Real-World Go Programming, 2nd Edition. – O'Reilly Media, 2024. – 491 p. ISBN-10: 1098139291, ISBN-13: 978-1098139292.

5. Mat Ryer. Go Programming Blueprints, 2nd Edition. – Packt Publishing, 2016. – 394 p. ISBN-10: 1786468948, ISBN-13: 978-1786468949.

6. Naren Yellavula. Building RESTful Web services with Go. – Packt Publishing, 2017. – 316 p. ISBN-10: 1788294289, ISBN-13: 978-1788294287.

7. Scott Chacon, Ben Straub. Pro Git, 2nd Edition. URL: https://git-scm.com/book/uk/v2. (дата звернення: 23.04.2025).

8. GitLab Docs. CI/CD pipelines. https://docs.gitlab.com/ci/pipelines/. (дата звернення: 23.04.2025).

9. Christopher Cowell, Nicholas Lotz, Chris Timberlake. Automating DevOps with GitLab CI/CD Pipelines: Build efficient CI/CD pipelines to verify, secure, and deploy your code using real-life examples. – Packt Publishing, 2023. – 348 p. ISBN-10: 1803233001, ISBN-13: 978-1803233000.

Додаток А.
Вимоги до оформлення звітності

Звіт оформлюється на листах формату А4 та включає титульний аркуш (зразок оформлення представлений на рис. А.1) та опис роботи за представленим нижче зразком.

У разі захисту звіту з виконання роботи при аудиторній формі навчання, звіт друкується та надається викладачу на розгляд.

Для захисту виконаної роботи у дистанційній (online) формі навчання, звіт подається до захисту в електронному форматі PDF (Portable Document Format) та надсилається викладачеві через засоби online-спілкування для розгляду.

Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»
Факультет інформаційних технологій
Кафедра інформаційних технологій та комп'ютерної інженерії

Звіт
з роботи № __
дисципліни "Технології DevOps"

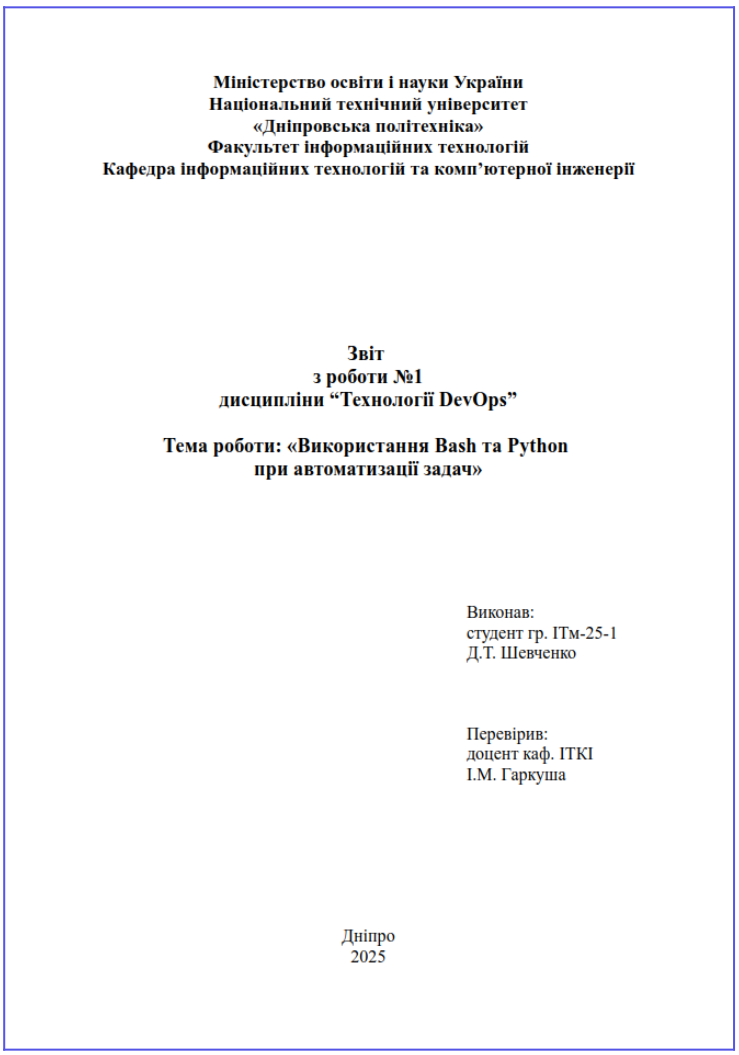
Тема роботи: « _____ »

Виконав:
студент гр. *шифр групи*
Ініціали та прізвище студента

Перевірив:
посада каф. ІТКІ
Ініціали та прізвище викладача

Дніпро
Поточний рік складання звіту

Рис. А.1. Форма оформлення титульного аркуша звіту



Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»
Факультет інформаційних технологій
Кафедра інформаційних технологій та комп'ютерної інженерії

Звіт
з роботи №1
дисципліни "Технології DevOps"

Тема роботи: «Використання Bash та Python
при автоматизації задач»

Виконав:
студент гр. ІТм-25-1
Д.Т. Шевченко

Перевірив:
доцент каф. ІТКІ
І.М. Гаркуша

Дніпро
2025

Рис. А.2. Приклад оформленого титульного аркуша звіту

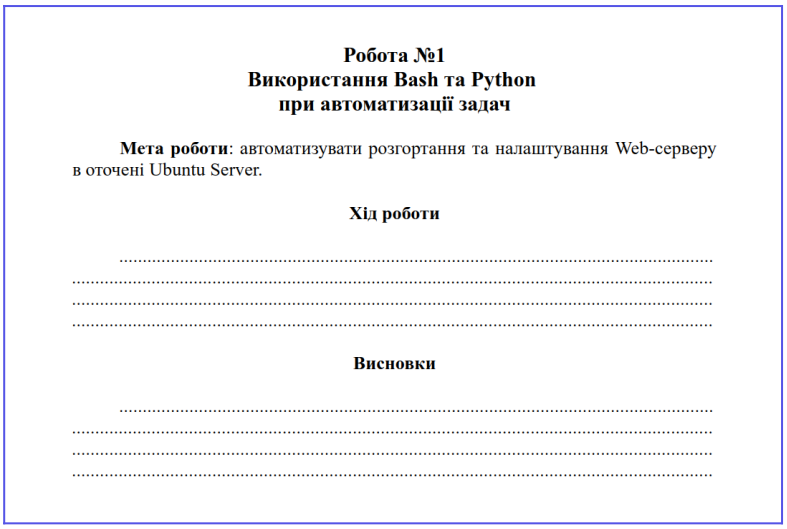
Опис виконаної роботи в звіті розпочинається з наступного аркуша. В горі сторінки, по центру, вказується номер роботи та її тема. Після цього вказується мета роботи. Через рядок по центру сторінки йдуть слова *Хід роботи* і далі через рядок розпочинається основний текст опису виконаної роботи.

Після завершення опису роботи, через рядок, йде частина, яка присвячена висновкам по проведеній роботі. Вона починається зі слова *Висновки*, розміщеного по центру листа та через рядок починається текст з висновками по роботі.

Приклад загального вмісту опису роботи представлений на рис. А.3.

Написання основного тексту ходу роботи та висновків йде **від третьої особи**!!! Тобто, є невірним використання у звіті висловів, на кшталт: "В ході роботи я розробив скрипт мовою Python". Замість такого вислову вірно буде написати: "В ході роботи розроблений скрипт мовою Python". Або замість вислову, на кшталт: "Під час виконання роботи ми опанували розгортання та використання Web-серверу", вірно буде написати:

“Під час виконання роботи опановано розгортання та використання Web-серверу”.



Робота №1
Використання Bash та Python
при автоматизації задач

Мета роботи: автоматизувати розгортання та налаштування Web-серверу в оточені Ubuntu Server.

Хід роботи

.....
.....
.....
.....

Висновки

.....
.....
.....
.....

Рис. А.3. Приклад загального подання опису роботи

Текст звіту оформляється шрифтом Times New Roman з розміром 14, міжрядковий інтервал 1 – 1,5. Абзацний відступ – 5 символів.

Шрифтом з розміром 10 оформлюють тексти кодів програм та скриптів, гіпертекстова розмітка html-документів. При цьому бажано використовувати один з наступних моноширинних шрифтів: Source Code Pro, Monospace, Consolas, DejaVu Sans Mono, Courier New, Monospaced, Menlo, SF Mono або подібні.

Вирівнювання тексту опису в звіті – по ширині сторінки.

Вирівнювання в звіті тексту коду програм, скриптів, гіпертекстової розмітки документів – ліворуч.

Вирівнювання рисунків та підписів до них – по центру сторінки.

Вирівнювання таблиць (якщо такі будуть) – по центру сторінки.

Якщо текст коду програми не є великим за обсягом, то його можна подати у вигляді знімку з екрану, причому фон повинен бути не чорного кольору, а текст коду чи розмітки повинен добре виділятися. Приклад такого знімку з екрану представлений на рис. А.4. Такий рисунок потрібно розмістити по центру сторінки звіту та надати по центру підпис для нього. Наприклад:

Рисунок 1 – Текст коду програми за пунктом 2 завдання роботи №4

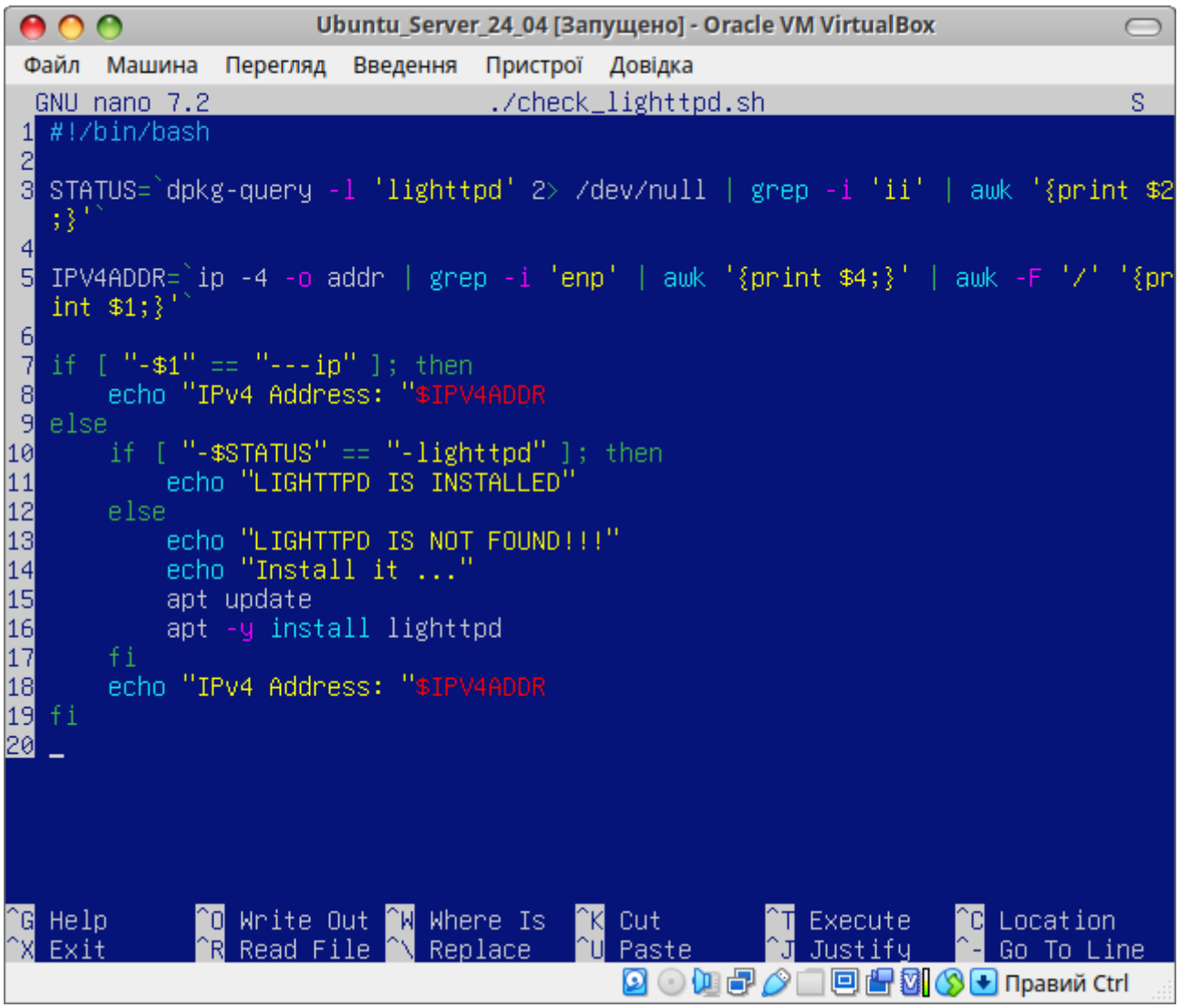


Рис. А.4. Приклад знімка з екрану скрипту для звіту

Якщо в тексті звіту наводиться рисунок, то на нього обов’язково повинно бути посилання з тексту опису ходу роботи та пояснення, щодо певних елементів коду або гіпертекстової розмітки, які представлені на вказаному рисунку. Наприклад, потрібно, вказуючи номер рядку, прокоментувати певну використану функцію та коротко описати її особливості.

При складанні тексту висновків, потрібно вказати на результати щодо досягнення поставленої мети роботи. Вказати особливості виконання роботи та складнощі. При цьому доречно залучати вислови на кшталт: “В роботі розглянуті...”, “...виконані...”, “Отриманий результат дозволив...”, “Використання умов дозволило...” та ін.

Додаток Б.
Діаграма розгортання компонентів практикуму

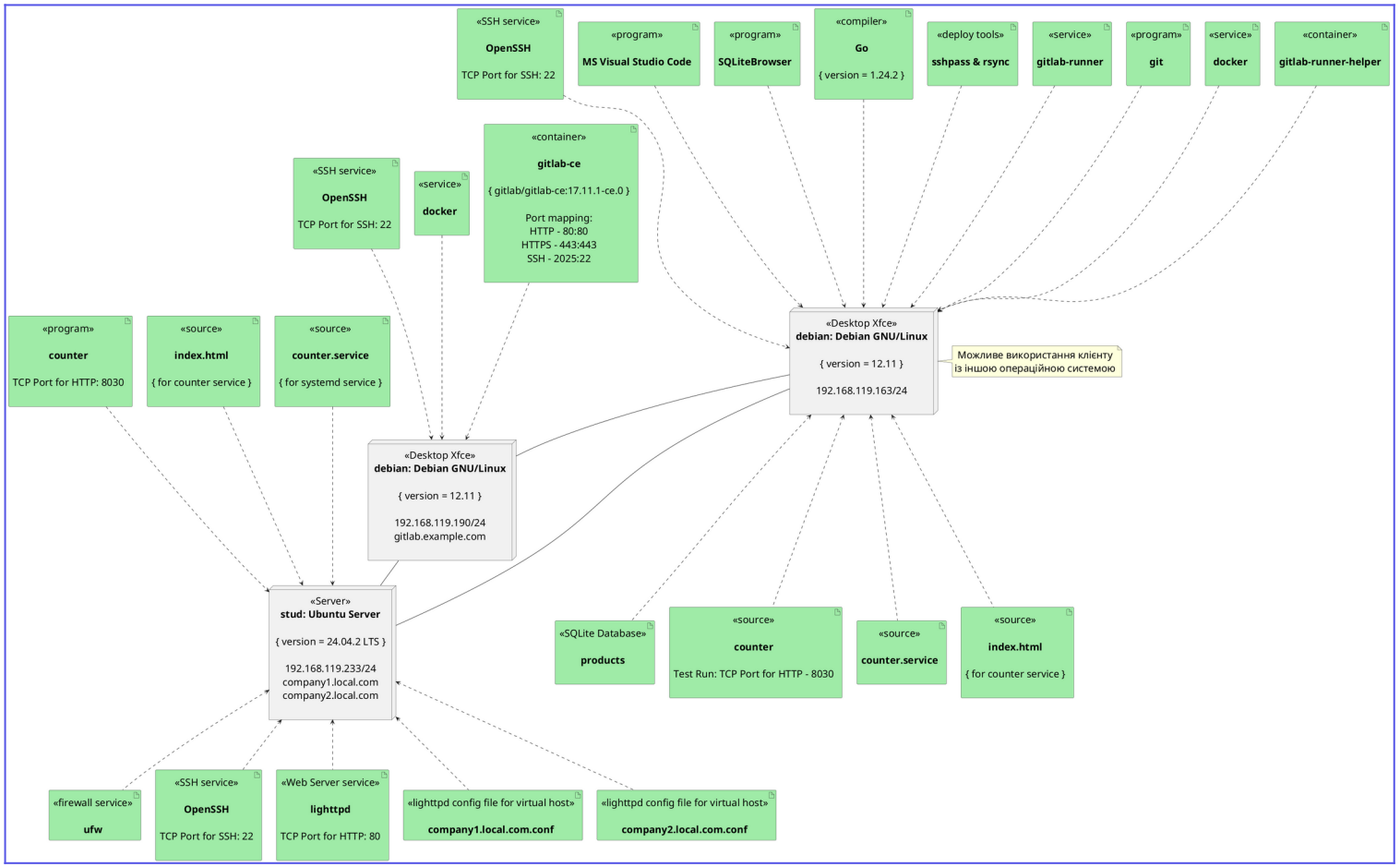


Рис. Б.1. Діаграма розгортання (deployment view) всіх складових вузлів після завершення виконання робіт, яка представлена мовою UML (Unified Modeling Language). Зазначені IP-адреси вказані в якості прикладу та можуть відрізнятися в залежності від мережевих налаштувань

Навчальне видання

Гаркуша Ігор Миколайович

ТЕХНОЛОГІЇ DEVOPS

Методичні рекомендації до виконання лабораторних робіт
для здобувачів ступеня магістра освітньо-професійної програми
«Інформаційні системи та технології»
спеціальності F6 Інформаційні системи і технології

Видано в авторській редакції.

Електронний ресурс.
Підписано до видання 02.09.2025. Авт. арк. 4,2.

Національний технічний університет «Дніпровська політехніка».
49005, м. Дніпро, просп. Дмитра Яворницького, 19.