ХЕРСОНСЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ ТА ДИЗАЙНУ
КАФЕДРА ПРОГРАМНИХ ЗАСОБІВ І ТЕХНОЛОГІЙ

МЕТОДИЧНІ РЕКОМЕНДАЦІЇ

і контрольні завдання до виконання лабораторних робіт з дисципліни
CI/CD ІНСТРУМЕНТАРІЙ

для студентів	4 курсу, 2 курсу (за скороченим терміном навчання)
підготовки	першого (бакалаврського) рівня вищої освіти
галузі знань	12 «Інформаційні технології»
спеціальності	121 «Інженерія програмного забезпечення»
освітньо-професійних програм	Програмна інженерія, Програмне забезпечення систем
факультету	Інформаційних технологій та дизайну

Хмельницький – 2025

Методичні рекомендації і контрольні завдання до виконання лабораторних робіт з дисципліни «CI/CD інструментарій» підготовки фахівців на першому (бакалаврському) рівні вищої освіти спеціальності 121 «Інженерія програмного забезпечення».

РОЗРОБНИКИ: Хохлов В.А., к.т.н.

РЕЦЕНЗЕНТ: Вороненко М.О., к.т.н., доцент

Затверджено на засіданні кафедри програмних засобів і технологій

Протокол № 8 від «02» травня 2025 року

Завідувач кафедри

_____ доц., к.т.н. О.Є.Огнєва

УЗГОДЖЕНО:
З НАВЧАЛЬНО-МЕТОДИЧНИМ ВІДДІЛОМ

Реєстраційний номер № _____

Зміст

Інструктаж з техніки безпеки (охорони праці на робочих місцях)

1. Загальні вимоги

1.1. До роботи у комп’ютерному класі допускаються студенти, які пройшли інструктаж з техніки безпеки з відповідним записом у журналі з техніки безпеки і підписами.

1.2. Не можна заходити й перебувати у комп’ютерному класі без викладача.

1.3. Робота у комп’ютерному класі має проводитися тільки в суворій відповідності до розкладу занять і графіка самостійної роботи викладача та студентів.

1.4. Студентам заборонено відчиняти шафи живлення і комп’ютери як тоді, коли ЕОМ працюють, так і тоді, коли вони вимкнені.

2.Вимоги безпеки перед початком роботи

2.1. Заборонено заходити до класу у верхньому одязі чи приносити його з собою.

2.2. Заборонено приносити на робоче місце особисті речі, дискети і т.п., крім ручки і зошита.

2.3. На робочому місці слід сидіти так, щоб можна було, не нахиляючись користуватися клавіатурою, і водночас повністю бачити зображення на екрані дисплея.

2.4. Починати роботу можна лише за вказівкою викладача або інженера.

3.Вимоги безпеки під час роботи

3.1. Заборонено ходити по комп’ютерному класу, голосно розмовляти.

3.2. Виконувати слід тільки зазначене викладачем завдання. Категорично заборонено виконувати інші роботи.

3.3. На клавіші клавіатури потрібно натискати плавно, не припускати ударів.

3.4. Користуватися друкувальним пристроєм дозволяється тільки у присутності викладача або інженера.

3.5. Заборонено самостійно переміщувати апаратуру.

3.6. Заборонено запускати ігрові програми.

3.7. У випадку виникнення неполадок треба повідомити викладача або інженера.

3.8. Не намагатися самостійно відрегулювати апаратуру або усувати в ній неполадки.

4.Вимоги безпеки після закінчення роботи

4.1. Про хиби та неполадки, помічені під час роботи, слід зробити записи у відповідних журналах.

4.2. На робочому місці не потрібно залишати зайвих предметів.

5.Вимоги безпеки в аварійних ситуаціях

5.1. При появі незвичайного звуку або вимкнення апаратури потрібно негайно припинити роботу й довести це до відома викладача або інженера.

5.2. При появі запаху паленого слід припинити роботу, вимкнути апаратуру і повідомити про це викладача чи інженера. Коли це необхідно, допомогти гасити пожежу.

5.3. При потраплянні людини під напругу необхідно знеструмити відповідне робоче місце, надати першу долікарську допомогу і викликати "швидку".

5.4. При виникненні пожежі необхідно знеструмити клас, викликати пожежну команду і приступити до гасіння пожежі засобами, які є.

5.5. У разі недотримання студентами вимог із охорони праці та пожежної безпеки адміністрація вузу може притягти їх до дисциплінарної та адміністративної відповідальності.

Лабораторна робота №1. Основи командних оболонок sh, bash

Тема роботи: Основи командних оболонок sh, bash. Мова командної оболонки
Мета роботи: ознайомитись з командними оболонками Unix-систем. Навчитись використовувати мову sh для розробки скриптів

Теоретичні відомості

Як правило, для налаштування процесів безперервної інтеграції/доставки (CI/CD) використовуються Unix-системи або Linux. Одним з елементів таких систем є командна оболонка, що використовується для керування системою, зокрема для запуску і виконання команд. Найчастіше використовується оболонка **bash**. **bash** (скор. від «Bourne-Again shell») — це командна оболонка (або «інтерпретатор командного рядка»), яка створена в 1989 році Брайаном Фоксом з метою вдосконалення командної оболонки **sh**. Іншою популярною оболонкою є zsh. Ці оболонки підтримують спеціальну мову програмування, що дозволяє автоматизувати процеси керування системою. За деякими незначними особливостями, ці мові дуже схожі. В світі BSD-систем популярною є оболонка **csh**, мова якої схожа на мову C. В Windows bahs доступний в проекті MinGW.

Будь-який сценарій мови Bash є простим текстовим файлом, який містить ряд команд для циклів і запитів. Вони є сумішшю команд, які ми зазвичай вводимо в командному рядку (наприклад, ls або cp) і команди, які ми могли б вводити в командному рядку, але які там не передбачені.

При виклику текстового файлу як програми команди обробляються одна за одною. Все, що ми робимо в оболонці, може бути записано в сценарії, і навпаки, сценарій може бути скопійований в оболонку і таким чином виконаний.

Для розробки bash-сценаріїв може використовуватись будь-який редактор, який дозволяє збереження в простому текстовому файлі, наприклад (n)vim.

Команди bash

Команда bash — це найменша одиниця коду, яку bash може виконати. За допомогою команд ми повідомляємо шеллу, що нам потрібно, щоб він зробив. bash зазвичай приймає від користувача одну команду та повертається до нього після того, як команда буде виконана.

Розглянемо деякі команди bash.

Команда **echo** — повертає все, що ви вводите в командному рядку:

```
[(2024-02-18) vadimkhohlov] ~/Documents/хнту/2023_2024% echo
'Hello, world!'
Hello, world!
```

Команда **date** — відображає поточну дату та час:

```
[(2024-02-18) vadimkhohlov] ~/Documents/хнту/2023_2024% date
Sun 18 Feb 2024 17:56:50 EET
```

Деякі інші команди:

pwd (скор. від «print working directory») — вказує на поточний робочий каталог, в якому команди шелла шукатимуть файли;

ls (скор. від «list») — відображає вміст каталогу;

cd (скор. від «change directory») — змінює поточну директорію на задану користувачем;

mkdir (скор. від «make directory») — створює новий каталог;

mv (скор. від «move») — переміщує один або кілька файлів/каталогів з одного місця до іншого;

rm (скор. від «remove») — видаляє файли/каталоги;

cat (скор. від «concatenate») — зчитує файл та виводить його вміст.

Команда **man** (скор. від “manual”) — відображає довідкові сторінки, які є посібником користувача, вбудованим за замовчуванням у деякі дистрибутиви Linux та більшість Unix-систем:

```
MV(1)
General Commands Manual
MV(1)


NAME
     mv – move files


SYNOPSIS
     mv [-f | -i | -n] [-hv] source target
     mv [-f | -i | -n] [-v] source ... directory


DESCRIPTION
```

```
     In its first form, the mv utility renames the file named by
the source operand to the destination path named by the target
operand.  This form is assumed when the last operand does not name
an already existing directory.
```

Написання sh-скриптів

Програми, написані мовою шелла, називаються shell-скриптами (або shell-сценаріями) і, як правило, мають розширення файлів .sh. Сама мова містить повний набір утиліт і команд, доступних в *nix-системах, а також цикли, умовні оператори, оголошення змінних тощо.

Для створення sh-скріпта достатньо створити текстовий файл з необхідними командами і зберегти його за бажання з розширенням sh:

```
[(2024-02-18) vadimkhohlov]
~/Documents/хнту/2023_2024/subjects/ci_cd/labs% ls

hello.sh

[(2024-02-18) vadimkhohlov]
~/Documents/хнту/2023_2024/subjects/ci_cd/labs% cat hello.sh

echo 'Hello, world!'
```

Для виконання файлу необхідно запустити інтерпретатор sh (або bash, або zsh) і вказати скрипт для виконання:

```
[(2024-02-18) vadimkhohlov]
~/Documents/хнту/2023_2024/subjects/ci_cd/labs% sh hello.sh

Hello, world!
```

Інший шлях — вказати, що даний файл є виконуваним файлом, встановивши для нього атрибут +x (executive) за допомогою команди chmod:

```
[(2024-02-18) vadimkhohlov]
~/Documents/хнту/2023_2024/subjects/ci_cd/labs% ls -l

total 4

-rw-r--r-- 1 vadimkhohlov staff 21 Feb 18 18:06 hello.sh

[(2024-02-18) vadimkhohlov]
~/Documents/хнту/2023_2024/subjects/ci_cd/labs% chmod +x hello.sh

[(2024-02-18) vadimkhohlov]
~/Documents/хнту/2023_2024/subjects/ci_cd/labs% ls -l

total 4

-rwxr-xr-x 1 vadimkhohlov staff 21 Feb 18 18:06 hello.sh
```

Після цього вже можна не вказувати командний інтерпретатор:

```
[(2024-02-18) vadimkhohlov]
~/Documents/хнту/2023_2024/subjects/ci_cd/labs% ./hello.sh
Hello, world!
```

Як правило, при створенні скриптів використовуються різні мови: Bash, Python, Ruby. Щоб вказати системи який саме інтерпретатор треба викликати для обробки поточного скрипту в його першому використовується спеціальний коментар, так званий Shebang-символ:

```
[(2024-02-18) vadimkhohlov]
~/Documents/хнту/2023_2024/subjects/ci_cd/labs% cat hello.sh
#!/bin/sh
echo 'Hello, world!'
```

З символу # в скриптах починаються коментарі, які пропускаються при виконанні скрипту.

Використовувати змінні в bash дуже просто — вони оголошуються шляхом написання свого імені. Надалі при зверненні до змінної потрібно просто вказати символ $ разом з нею. Особливістю sh є те, що символи пробілів біля символу = є помилкою.

```
[(2024-02-18) vadimkhohlov]
~/Documents/хнту/2023_2024/subjects/ci_cd/labs% cat vars.sh
#!/bin/sh
var="hello, world!"
echo $var

n1=10
n2=20

n3=$((n1+n2))

echo "Sum: $n3"
echo 'Sum: $n3'
```

Результат виконання скрипту:

```
[(2024-02-18) vadimkhohlov]
~/Documents/хнту/2023_2024/subjects/ci_cd/labs% sh vars.sh
hello, world!
Sum: 30
```

```
Sum: $n3
```

Рядок коду n3=$((n1+n2)) виконує обчислення виразу. Якщо змінна з’являється у рядку у подвійних лапках “, то замість неї буде підставлено її значення (string interpolation). Рядки в одинарних лапках ‘, друкуються, як є. Рядок без дужок $ (()) не виконує обчислень, а просто формує рядок з підставлених значень:

```
n4=$n1+$n2
echo $n4
```

Результат:

```
10+20
```

Умовні оператори bash використовуються для прийняття рішень. В умовних операторах обчислюється конкретна умова. Якщо умова істинна (true), виконується перший блок коду. В протилежному випадку (false) виконується другий блок коду.

Можна обчислити одну або кілька умов усередині блоку if за допомогою операторів OR (||) та AND (&&). У bash блок if починається з ключового слова if і закінчується ключовим словом fi. Якщо задана умова помилкова, то виконується блок else.

Приклад:

```
#!/bin/sh
echo 'Eneter a number:'
read num

if [ $num -gt 10 ];
then
    echo " > 10 "
elif [ $num -eq 10 ];
then
    echo "== 10"
elif [ $num -lt 10 ];
then
    echo "< 10"
else
    echo "Wrong number"
```

```
fi
```

В квадратних дужках обчислюється вираз. Оператор -gt виконує перевірку на більше (greater than), -eq — перевірку на рівність (equal), -lt — на менше (less than); read — виконує введення з клавіатури.

Bash також підтримує такі цикли як while:

```
#!/bin/bash
VAR=1
while [ $VAR -le 10 ]
do
        echo "Значення  $VAR"
        #збільшуємо значення змінної на 1
        (( VAR++ ))
done
```

і for:

```
#!/bin/bash

#оголошуємо цикл for
for ((i=0; i<10; i++))
do
  #виводимо на екран значення змінної
  echo -n "$i "
done

echo "\n"
```

З прикладом розробки доволі складного bash-скрипту з поясненнями можна ознайомитись по посиланню: https://nklug.org.ua/lg/rus/articles/gariki.html

Завдання

Розробити sh-скрипт, що запитує у користувача його ім’я та вік і виводить на екран текст:

‘Привіт _name_, через рік тобі буде _age_ років!’

Лабораторна робота №2. Основи систем керування версіями

Тема роботи: основи систем керування версіями. Git.
Мета роботи: ознайомитись з можливостями Git для керування версіями програмного забезпечення

Теоретичні відомості

Розробка програмного забезпечення — довготривалий процес, який, як правило, ведеться групою програмістів. Серед задач, які виникають під час цього процесу — розмежування програмістів одне від одного, об’єднання роботи різних програмістів в одну кодову базу, повернення до одного з попередніх станів ПЗ. Для розв’язання цих проблем використовується системи контролю версій (СКВ, Version Control System — VCS). Такі системи можуть бути локальними, тобто застосовуватись на одному робочому місці. Вони мають просту базу даних, яка зберігає всі зміни в файлах під контролем версій Приклад — RCS. Для організації взаємодії багатьох розробників були створені централізовані СКВ, які мають мають єдиний сервер, який містить всі версії файлів, та деяке число клієнтів, які отримують файли з центрального місця. Приклади — CVS або Subversion. Наступним етапом стала поява децентралізованих СКВ, клієнти яких є повною копією сховища разом з усією його історією. Це дозволяє з одного боку працювати з історією змін локально, навіть оффлайн, а з іншого — при необхідності обмінюватись даними з іншими розробниками. Однією з таких систем, що стала фактичним стандартом СКВ є **Git**. Основне джерело інформації по fastlane знаходиться за адресою: https://git-scm.com/book/uk/v2

Встановлення Git і виклик команд git

Як правило, git встановлюється з системою розробки, наприклад, Xcode Command Line Tools. Також можна скористатися пакетним менеджером системи, або завантажити зі сторінці: https://git-scm.com/download/. Для виконання команд Git використовується програма командного рядка git, якій вказуються команди, які потрібно виконати. Також існує велика кількість графічних клієнтів Git. Більшість сучасних систем розробки (IDE) також підтримують роботу з Git безпосередньо з меню.

Стандартний виклик Git в командному рядку виглядає наступним чином:

```
git command argumnets
```

Наприклад:

```
~/ci_cd/labs/lab2% git version
git version 2.39.2 (Apple Git-143)
```

Щоб отримати інформацію по команді git треба в якості аргументи вказати — help.

Файлове сховище, яким керує git називається сховищем або репозитарієм (repository). Для визначення статусу репозитарію використовується команда status:

```
~/ci_cd/labs/lab2% git status
fatal: not a git repository (or any of the parent
directories): .git
```

Для ініціалізації репозитарію в якомусь каталозі використовується команда init, якій треба вказати потрібний каталог або . (точка) для ініціалізації git в поточному каталозі:

```
~/ci_cd/labs/lab2% git init .
Initialized empty Git repository in
/Users/vadimkhohlov/ci_cd/labs/lab2/.git/
```

Git створить службовий каталог .git, в якому буде зберігатися службова інформація.

```
~/ci_cd/labs/lab2[main]% git status
On branch main


No commits yet


nothing to commit (create/copy files and use "git add" to track)
```

Основні концепції git

Git сприймає свої дані як низку знімків мініатюрної файлової системи. У Git щоразу, як зберігається стан проекту (в термінах Git “робиться коміт” commit), Git запам’ятовує як виглядають всі файли в той момент і зберігає посилання на цей знімок. Для ефективності, якщо файли не змінилися, Git не зберігає файли знову, просто робить посилання на попередній ідентичний файл, котрий вже зберігається.

Будь-що в Git, перед збереженням, отримує контрольну суму, за якою потім і можна на нього посилатися. Таким чином, неможливо змінити файл чи директорію так, щоб Git про це не дізнався. Механізм, який використовується

для цього контролю, називається хеш SHA-1. Він являє собою 40-символьну послідовність цифр та перших літер латинського алфавіту (a-f) і вираховується на основі вмісту файлу чи структури директорії в Git. SHA-1 хеш виглядає це приблизно так:

```
14b9da6552252987aa493b52f8696cd6d3b00372
```

Git має три основних стани, в яких можуть перебувати файли: збережений у коміті (**commited**), змінений (**modified**) та індексований (**staged**):

- Збережений у коміті означає, що дані безпечно збережено в локальній базі даних.
- Змінений означає, що у файл внесено редагування, які ще не збережено в базі даних.
- Індексований стан виникає тоді, коли позначається змінений файл у поточній версії, щоб ці зміни ввійшли до наступного знімку коміту.

У директорії Git система зберігає метадані та базу даних об’єктів проекту. Це найважливіша частина Git, саме вона копіюється при клонуванні сховища з іншого комп’ютеру.

Робоче дерево — це одна окрема версія проекту, взята зі сховища. Ці файли видобуваються з бази даних у теці Git та розміщуються на диску для подальшого використання та редагування.

Індекс — це файл, що зазвичай знаходиться в директорії Git і містить інформацію про те, що буде збережено у наступному коміті. Також цей файл називають “областю додавання” (staging area).

1. Найпростіший процес взаємодії з Git виглядає приблизно так:
2. Ви редагуєте файли у своїй робочій директорії.
3. Вибірково надсилаєте до індексу лише ті зміни, що їх ви бажаєте зберегти в наступному коміті, і лише ці зміни буде збережено в індексі.
4. Створюєте коміт: знімок з індексу остаточно зберігається в директорії Git.

Основи роботи з Git

Кожен файл робочої директорії може бути в одному з двох станів: контрольований (**tracked**) чи неконтрольований (**untracked**). Контрольовані файли — це файли, що були в останньому знімку. Вони можуть бути не зміненими, зміненими або індексованими. Якщо стисло, контрольовані файли — це файли, про які Git щось знає.

Неконтрольовані файли — це все інше, будь-які файли у робочій директорії, що не були в останньому знімку та не існують у вашому індексі.

```
~/ci_cd/labs/lab2[main]% echo "Project description" > README.md
~/ci_cd/labs/lab2[main]% touch script.sh
~/ci_cd/labs/lab2[main]% git status
On branch main

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
     README.md
     script.sh

nothing added to commit but untracked files present (use "git add" to track)
```

Команда status показує, що в репозиторії ще немає жодного коміту, а в каталозі знаходяться два неконтрольовані файли: README.md та script.sh.

Щоб почати контролювати новий файл, треба використати команду git add:

```
~/ci_cd/labs/lab2[main]% git add README.md
~/ci_cd/labs/lab2[main+]% git status
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
     new file:   README.md

Untracked files:
  (use "git add <file>..." to include in what will be committed)
     script.sh
```

Команда status показує, що файл README.md контролюються git і знаходиться в staging area.

Для збереження змін, тобто виконання коміту використовується команда commit, якій через параметр -m потрібно вказати повідомлення або пояснення до коміту:

```
~/ci_cd/labs/lab2[main+]% git commit -m "Initial commit"
[main (root-commit) 8b381e5] Initial commit
 1 file changed, 1 insertion(+)
 create mode 100644 README.md
```

8b381e5 — це початок хешу коміта.

Розгалуження

При розробці нової функціональності вважається хорошою практикою робота з копією оригінального проекту, яку називають гілкою. Гілки мають свою власну історію і окремі інші зміни до тих пор, поки ви не вирішуєте злити (об’єднати) зміни разом. Це відбувається з кількох причин:

- Уже робоча, стабільна версія коду зберігається.
- Різні нові функції можуть розроблятися паралельно різними програмістами.
- Розробники можуть працювати з власними гілками без ризику, що кодова база змінюється із-за чужих змін.
- У разі сумнівів, різні реалізації однієї і тієї ж ідеї можуть бути розроблені в різних гілках і потім зрівнятися.

Основна гілка у кожному репозиторії називається master або main, але це ім’я можна змінити.

Створення нової гілки виконується командою git branch <branch_name>:

```
git branch develop
```

Щоб переглянути всі наявні гілки застосовується команда branch без аргументів:

```
~/ci_cd/labs/lab2[main]% git branch
  develop
* main
```

В даному сховищі дві гілки, символом * помічена активна.

Для перемикання на іншу гілку виконується команда git switch (або git checkout у старих версіях Git):

```
~/ci_cd/labs/lab2[main]% git switch develop
Switched to branch 'develop'
/ci_cd/labs/lab2[develop]% git branch
* develop
  main
```

Після перемикання на нову гілку в ній можна працювати з поточною копією сховища: редагувати інсуючи файли, додавати нові:

```
~/ci_cd/labs/lab2% echo -n >> README.md
~/ci_cd/labs/lab2% echo "Version 1.0" >> README.md
~/ci_cd/labs/lab2% touch script2.sh
~/ci_cd/labs/lab2[develop!]% git status
On branch develop
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working
directory)
     modified:   README.md

Untracked files:
  (use "git add <file>..." to include in what will be committed)
     script2.sh

no changes added to commit (use "git add" and/or "git commit -a")
~/ci_cd/labs/lab2[develop!]% git add README.md
~/ci_cd/labs/lab2[develop+]% git status
On branch develop
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
     modified:   README.md

Untracked files:
  (use "git add <file>..." to include in what will be committed)
```

```
     script2.sh
~/ci_cd/labs/lab2[develop+]% git add script2.sh
~/ci_cd/labs/lab2[develop+]% git status
On branch develop
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
     modified:   README.md
     new file:   script2.sh
~/ci_cd/labs/lab2[develop+]% git commit -m "bump to version 1.0"
[develop 4bfbb3a] bump to version 1.0
 2 files changed, 1 insertion(+)
 create mode 100644 script2.sh
```

Для копіювання змін в одній гілці в іншу використовується команда git merge:

```
~/ci_cd/labs/lab2[develop]% git switch main
Switched to branch 'main'
~/ci_cd/labs/lab2[main]% git merge develop
Updating bfebea1..4bfbb3a
Fast-forward
 README.md  | 1 +
 script2.sh | 0
 2 files changed, 1 insertion(+)
 create mode 100644 script2.sh
```

Завдання

1. Створити новий репозитарій
2. Додати в репозитарій кілька файлів
3. Створити нову гілку
4. В створенній гілці змінити один з файлів з п.2
5. Злити зміни зі створеної гілці в основну
6. Відобразити команди з пп.1-5 у звіті лабораторної роботи

Лабораторна робота №3. Використання fastlane для автоматизації процесів зборки додатків

Тема роботи: використання fastlane для автоматизації процесів зборки додатків

Мета роботи: ознайомитись з можливостями інструментарію fastlane для математизації зборки мобільних додатків

Теоретичні відомості

Підготовка додатків до публікації в Google Play або App Store вимагає виконання одноманітних дій: створення архіву додатка, оновлення метаданих (наприклад інформації про нову версію), оновлення скриншотів, підпису архіву цифровим ключем.

Одним з інструментів автоматизації процесів публікації мобільних додатків є **fastlane**. Основне джерело інформації по fastlane знаходиться за адресою: https://docs.fastlane.tools/

Встановлення fastlane

Встановити fastlane можна кількома способами.

В Mac Os мож скористатися пакетним менеджером brew:

```
brew install fastlane
```

В Linux — відповідним пакетним менеджером конкретного дистрибутиву.

Ця система написана мовою Ruby, тому найпростіше установити її за допомогою gem:

```
gem install fastlane
```

Після встановлення перевірити доступність fastlane можна наступною командою:

```
[(2024-02-20) vadimkhohlov] ~/work/heroes2% fastlane --version
fastlane installation at path:
/usr/local/Cellar/fastlane/2.212.2/libexec/gems/fastlane-2.212.2/bin/fastlane
-----------------------------
[✔]
fastlane 2.212.2
```

Налаштування fastlane для Android-проєкту

Щоби додати в поточний проект підтримку fastlane необхідно в кореневому каталозі проєкту виконати команду:

```
fastlane init
```

після чого на запит команди ввести назву пакета проекту. Наступним кроком fastlane запитає шлях до json-файлу, що містить інформацію про акаунт розробника. Даний етап можна пропустити і виконати налаштування пізніше. Також можна пропустити вивантаження метаінформації про проект в Google Play або Apple Store.

Результатом виконання команди init буде створення в поточному каталозі каталогу fastlane, що містить наступні файли:

Appfile — містить інформацію про пакет додатка і інформацію, необхідну для підпису (signing) архіву.

Fastfile — містить команди автоматизації (lanes)

```
[(2024-02-21) vadimkhohlov] ~wx/MuBarometer/app/fastlane[master]% cat Appfile
package_name "org.xbasoft.mubarometer"
```

Fastfile — це фактично ruby-програма, що містить, набір команд (lane). Команда-lane, що визначає lane_name, має наступний синтаксис:

```
desc "Lane description"
lane :lane_name do
    action1
    action2
     ...
     actionN
end
```

Також потрібно вказувати платформу (andoid або ios) для якої будуть ці команди виконуватись. Таким чином, приклад Fastfile може бути наступним:

```
default_platform(:android)

platform :android do
 desc "Runs all the tests"
 lane :test do
```

```
    gradle(task: "test")
  end

  desc "Submit a new Beta Build to Firebase App Distribution"
  lane :beta do
    gradle(task: "clean assembleRelease")
    firebase_app_distribution

    # sh "your_script.sh"
    # You can also use other beta testing services here
  end

  desc "Deploy a new version to Google Play"
  lane :deploy do
    gradle(task: "clean assembleRelease")
    upload_to_play_store
  end
end
```

Опис lane не є обов’язковим.

Fastlane містить велику кількість вбудованих команд (action): gradle, firebase_app_distribution, upload_to_play_store, тощо. Список команд: https://docs.fastlane.tools/actions/. Також стороні розробники створюють свої плагіни з додатковими командами. Список наявних плагінів: https://docs.fastlane.tools/plugins/available-plugins/. При необхідності можна запускати shell-скрипти наступним чином:

```
desc "Run a shell-script"
  lane :shell_test do
    sh "your_script.sh"
  end
```

Запускається lane наступним чином:

```
fastlane shell_test
```

Приклад Fastfile:

```
[(2024-02-21) vadimkhohlov] ~wx/mubarometer/app[master]% cat
fastlane/Fastfile
```

```
default_platform (:android)


platform :android do
    desc "Simle lane"
    lane :hello do
        puts "Hello, world!"
    end
end
```

Результат роботи:

```
[(2024-02-21) vadimkhohlov] ~wx/mubarometer/app[master]% fastlane
hello
[✔] 🚀
[22:20:17]: Get started using a Gemfile for fastlane
https://docs.fastlane.tools/getting-started/ios/setup/#use-a-
gemfile
[22:20:19]: ------------------------------
[22:20:19]: --- Step: default_platform ---
[22:20:19]: ------------------------------
[22:20:19]: Driving the lane 'android hello' 🚀
[22:20:19]: Hello, world!


+------+------------------+-------------+
|           fastlane summary            |
+------+------------------+-------------+
| Step | Action           | Time (in s) |
+------+------------------+-------------+
| 1    | default_platform | 0           |
+------+------------------+-------------+


[22:20:19]: fastlane.tools finished successfully 🎉
```

Розглянемо можливості fastlane для автоматизації процесу розробки Android-додатків. Для виконання gradle-завдань використовується action gradle в форматі:

```
gradle(task: "clean assembleRelease")
```

Однією з задач при розробці ПЗ є керування версіями систем, що розробляються. Для Android-додатків версія вказується в файлі app/build.gradle в полях versionCode і versionName. Для fastlane існує плагін increment_version_code. Щоб його встановити необхідно виконати команду:

```
fastlane add_plugin increment_version_code
```

Після цього можно створювати lane:

```
desc "Increment version code"
lane :increment_vc do
     increment_version_code(gradle_file_path: "./app/build.gradle")
end
```

і запускати його:

```
fastlane increment_vc
```

Також fastlane дозволяє підготовлювати додатки для публікації в Google Play або Apple Store, керувати цифровими ключами, генерувати скриншоти та виконувати інші операції по розробці і дистрибуції Andoid- і iOS-додатків.

Завдання

1. При необхідності створити базовий Android-проект (в Andoid Studio або Visual Studio Code).
2. Встановити fastlane.
3. Активувати fastlane для обраного Android-проекту
4. Створити команди для:
   - зміни версії додатка
   - виклику shell-скрипту з роботи №1
   - підготовки release-архіву додатку

Лабораторна робота №4. Додаткові можливості Git.

Тема роботи: додаткові можливості Git
Мета роботи: ознайомитись можливостями Git для роботи з віддаленими репозитаріями а також колективної роботи

Теоретичні відомості

Git є децентралізованою СКВ, тобто не вимагає окремого сервера для зберігання сховища. Однак, над великими проектами, як правило, працюють групи розробників. Для організації взаємної роботи над кодовою базою таких проектів використовуються віддалені сховища. **Віддалені сховища** — це версії проекту, що розташовані в Інтернеті, або десь у мережі. Їх може бути декілька, кожне зазвичай або тільки для читання, або для читання та змін. Співпраця з іншими вимагає керування цими віддаленими сховищами, надсилання (**pushing**) даних до них та отримання (**pulling**) даних з них, коли необхідно зробити внесок.

Робота з віддаленими сховищами

Для початку роботи з віддаленим сховищем його необхідно скопіювати в локальне за допомогою клонування. Клонування виконується командою **git clone**:

```
git clone https://github.com/xvadim/number_facts2.git

Cloning into 'number_facts2'...
remote: Enumerating objects: 291, done.
remote: Counting objects: 100% (291/291), done.
remote: Compressing objects: 100% (195/195), done.
remote: Total 291 (delta 66), reused 284 (delta 59), pack-reused 0
Receiving objects: 100% (291/291), 290.67 KiB | 1.35 MiB/s, done.
Resolving deltas: 100% (66/66), done.
```

Git підтримує різні протоколи для доступу до мережевих ресурсів: https, ssh, тощо.

Команда додає локально віддалене сховище з назвою **origin**. Щоб побачити, які віддалені сервера налаштовані, використовується команда **git remote**:

cd number_facts2

git remote -v

origin https://github.com/xvadim/number_facts2.git (fetch)

origin https://github.com/xvadim/number_facts2.git (push)

Щоб додати нове віддалене Git сховище під заданим ім’ям, на яке можна буде легко посилатись, використовується команда **git remote add** <ім’я> <посилання>:

```
git remote add nf https://github.com/xvadim/number_facts2.git
git remote -v

nf   https://github.com/xvadim/number_facts2.git (fetch)
nf   https://github.com/xvadim/number_facts2.git (push)
origin    https://github.com/xvadim/number_facts2.git (fetch)
origin    https://github.com/xvadim/number_facts2.git (push)
```

Після цього вказане ім’я можна буде використовувати замість повного посилання.

Для отримання данних з віддалених проектів використовується команда **git fetch <remote>**. За замовчання використовується origin в якості посилання:

```
 git fetch -v

POST git-upload-pack (202 bytes)
From https://github.com/xvadim/number_facts2
 = [up to date]      main       -> origin/main
```

Прапорець -v змушує команди друкувати додаткову інформацію під час виконання.

Ця команда заходить на віддалений проект та забирає звідти усі дані, котрих локально досі нема. Після цього, у вас будуть посилання на всі гілки з того сховища, які ви можете зливати або оглядати в будь-який час.

Якщо поточна гілка налаштована слідкувати за віддаленою гілкою, можна виконати команду **git pull** щоб автоматично отримати зміни та злити віддалену гілку до поточної гілки. Команда git clone автоматично налаштовує локальну гілку main слідкувати за віддаленою гілкою main (хоча вона може називатись і по іншому) на віддаленому сервері, з якого зробили клон. Виконання git pull зазвичай дістає дані з серверу, з якого зробили клон, та намагається злити її з кодом, над яким зараз виконується робота.

Для відсилання локальних змін на віддалене сховище використовується команда git push <сховище> <гілка>:

```
vimr pubspec.yaml


git status


On branch main
Your branch is up to date with 'origin/main'.


Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working
directory)
     modified:   pubspec.yaml


no changes added to commit (use "git add" and/or "git commit -a")


g commit -am "Bumb build number"
[main e32c044] Bumb build number
 1 file changed, 1 insertion(+), 1 deletion(-)


git push origin main
```

Команда git remote show <сховище> відображає інформацію про віддалене сховище:

```
 git remote show origin


* remote origin
  Fetch URL: https://github.com/xvadim/number_facts2.git
  Push  URL: https://github.com/xvadim/number_facts2.git
  HEAD branch: main
  Remote branch:
    main tracked
  Local branch configured for 'git pull':
    main merges with remote main
```

```
  Local ref configured for 'git push':
    main pushes to main (fast-forwardable)
```

Використання GitHub для організації колективної роботи

Одним з найпопулярніших сервісів для зберігання віддалених git-сховищ є GitHub. Він дозволяє створювати приватні і публічні сховища й організовувати колективну роботу над ними. Багато opensource-проектів, на приклад Flutter, використовують GitHub.

Для роботи з GitHub можна використовувати як спеціальні додатки з графічним інтерфейсом (наприклад, Github Desktop), так і команди git.

При роботі з віддаленими сховищами Git може використовувати різні мережеві протоколи: https, ssh, тощо. При роботі з приватними сховищами, які захищені паролем під час використання протоколу https доведеться вводити логін та пароль. Протокол ssh, в свою чергу, більш безпечний і дозволяє налаштувати ключі для своєї роботи замість вводу логіна та пароля. Остання особливість корисна для налаштування конвеєрів CI/CD.

Протокол ssh використовує пару з приватного й публічного ключів. Приватний зберігається локально і має бути захищений. Публічній ключ можна копіювати на потрібні сервіси для роботи з ними.

Для створення пари ключів використовується команда:

ssh-keygen -t ed25519 -C "your_email@example.com"

На запитання команди треба вказати назву файлів, в яких будуть зберігатися ключі і секретну фразу (при необхідності). В Linux та macOS ключі, як правило, зберігається в каталозі ~/.ssh:

```
ls -l ~/.ssh/xvadim-GitHub*


-rw------- 1 vadimkhohlov staff 411 Aug  2  2021
/Users/vadimkhohlov/.ssh/xvadim-GitHub
-rw-r--r-- 1 vadimkhohlov staff  97 Aug  2  2021
/Users/vadimkhohlov/.ssh/xvadim-GitHub.pub
```

Публічний ключ має розширення .pub і доступний всім для читання. Приватний ключ розширення на має і доступний лише власнику.

Після створення ключів, треба оновити файл ~/.ssh/config і додати запис про створені ключи і відповідний ресурс для роботи з яким ці ключі були створені:

cat ~/.ssh/config

IdentityFile ~/.ssh/vadim.khohlov

Host xvadim-GitHub

HostName github.com

User xvadim

PreferredAuthentications publickey

IdentityFile /Users/vadimkhohlov/.ssh/xvadim-GitHub

Налаштування ssh і створення ключів в системи Windows описано, зокрема на цьому ресурсі. GitHub має власну секцію з інформацією про створення ключів.

Створений публічний ключ необхідно додати на GitHub в своєму профілі в секції «SSH and GPG keys».

Після налаштування ssh можна використовувати цей протокол для роботи з віддаленими сховищами:

```
ssh-add ~/.ssh/xvadim-GitHub
Identity added: /Users/vadimkhohlov/.ssh/xvadim-GitHub
(xvadima@ukr.net)
git clone git@github.com:xvadim/number_facts2.git
Cloning into 'number_facts2'...
remote: Enumerating objects: 291, done.
remote: Counting objects: 100% (291/291), done.
remote: Compressing objects: 100% (195/195), done.
remote: Total 291 (delta 66), reused 284 (delta 59), pack-reused 0
Receiving objects: 100% (291/291), 290.67 KiB | 878.00 KiB/s,
done.
Resolving deltas: 100% (66/66), done.

cd number_facts2
vimr pubspec.yaml
git commit -am "Bump build number"

[main ab9c6b9] Bump build number
```

```
 1 file changed, 1 insertion(+), 1 deletion(-)


git push origin
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 12 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 288 bytes | 288.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (2/2), completed with 2 local
objects.
To github.com:xvadim/number_facts2.git
   f1e2ad0..ab9c6b9  main -> main
```

Команда shh-add додає вказаний ключ до списку ключів, які команда ssh використовує в своїй роботі. Як правило, її треба використовувати лише раз після запуску системи.

Завдання

1. Створити новий репозитарій на GitHub
2. Склонувати репозитарій на локальну машину
3. Зробити окрему гілку
4. Додати в репозитарій кілька файлів
5. Відправити зміни в віддалений репозитарій
6. Додати до репозитарію на GitHub користувача xvadim.
7. Зробити pull-request і додати xvadim до списку ревьюверів
8. Отримати схвалення PR
9. Виконати злиття створеної гілки з основною
10. Відобразити команди з пп.1-9 у звіті лабораторної роботи

Лабораторна робота №5. Github actions

Тема роботи: Github actions.
Мета роботи: ознайомитись з можливостями GitHub для організації процесів CI/CD

Теоретичні відомості

GitHub має власну платформу CI/CD, яка дозволяє автоматизувати процеси зборки, тестування та доставки додатків, - GitHub Actions.

Ви можете створювати робочі процеси, які запускають тести щоразу, коли ви надсилаєте зміни до свого репозиторію, або які розгортають додаток після pull requests. Такі процесу описуються за допомогою спеціальних workflow-файлів. Це yaml-файли, які мають розміщуватись в каталозі .github/workflows репозитарію. Розглянемо приклад workflow-файла для створення flutter-додатка:

```
name: Create Android adHoc build

on:
  # Run from 'Actions' menu manually
  workflow_dispatch:

jobs:
  create-adhoc-build:
    runs-on: ubuntu-latest
    steps:
      - uses:  actions/checkout@v3
      - uses: actions/setup-java@v3
        with:
          distribution: 'zulu'
          java-version: '12.x'

      - uses: subosito/flutter-action@v2
        with:
          flutter-version: '3.7.12'
          channel: 'stable'
```

```
    - name: Get dependencies
      run: flutter pub get

    - name: Start adHoc build
      run: flutter build apk --debug

    # Upload a build to gihub storage
    - uses: actions/upload-artifact@v1
      with:
        name: APK for QA
        path: build/app/outputs/apk/dev/debug/app-dev-debug.apk
```

Секція **name** задає ім’я цього процесу, що буде видно в інтерфейсі GitHub.

on задає події, у відповідь на які буде запускатися вказаний процес. В даному випадку використовується workflow_dispatch — запуск процесу вручну з інтерфейсу GitHub. Інші можливі події:

- push — процес запуститься після коміту в репозитарій
- pull_request_review — процес запуститься після того, як хтось виконає review пул-реквеста

Список інших можливих подій перелічено за цим посиланням: https://docs.github.com/en/actions/using-workflows/events-that-trigger-workflows

Процеси складаються з набору задач (**jobs**), кожна з яких має унікальне ім’я. В даному випадку використовується одна задача — create-adhoc-build. Задача запускається в середовищі з операційною системою, яка вказується в параметрі runs-on.

Задачі складаються з послідовності кроків (**steps**). Кожен крок може або використовувати вже існуючи дії (**actions**), або запускати нові команди. Параметр uses вказує, що крок має запустити вказану дію. Дії, що починаються з actions/ вбудовані в GitHub actions. Також можна створювати власні дії, розмізщувати їх на GitHub і використовувати їх, вказуючи репозитарій і потрібну дію.

Дії, що використовуються в даному процесі:

- actions/checkout@v3 — стандартна дія, що виконує клонування поточного репозиторія. Через @ вказується версія дії.
- actions/setup-java@v3 — стандартна дія встановлення на цільову машину системи java. За допомогою with вказуються додаткові параметри дії.
- subosito/flutter-action@v2 — дія для встановлення дистрибутиву flutter

Більш детально про використання дій написано за цим посиланням: https://docs.github.com/en/actions/learn-github-actions/finding-and-customizing-actions

Також кроки запускати команди. Кожна команда має ім’я, яке друкується під час виконання процесу і набір команд операційної системи, які треба виконати (run). Приклади команд:

- flutter pub get — оновлення залежностей для flutter-проєкта
- echo „Hello“ — друк тексту під час виконання процесу.

Більш детально робота з командами описана за посиланням: https://docs.github.com/en/actions/using-workflows/workflow-commands-for-github-actions

Завдання

В репозитарії на GitHub створити два робочі процеси.

Один має запускатися вручну і виводити інформацію про операційну систему, що використовується. В Unix-подібних системах таку інормацію можна отримати за допомогою команди uname:

```
uname -v
```

```
Darwin Kernel Version 23.4.0: Fri Mar 15 00:11:05 PDT 2024;
root:xnu-10063.101.17~1/RELEASE_X86_64
```

Другий процес має запускатися при пуші в репозитарій і друкувати поточну системну дату.

Лабораторна робота №6. Встановлення і налаштування GitLab

Тема роботи: встановлення і налаштування GitLab
Мета роботи: навчитись встановлювати і використовувати GitLab

Теоретичні відомості

GitLab — це веб-сервіс для керування репозиторіями Git, який надає повний DevOps-цикл: від зберігання коду до CI/CD, моніторингу, управління задачами та розгортання.
Основні компоненти GitLab:

- Git-репозиторії
- Web-інтерфейс для керування проєктами
- Система CI/CD
- Інтеграція з Docker/Kubernetes
- Управління доступом

Система для встановлення GitLab має задовольняти наступним вимогам:

7. Операційна система: Ubuntu, Debian, CentOS, RHEL
8. RAM: мінімум 4 GB (рекомендовано 8 GB+)
9. CPU: 2 ядра+
10. Диск: SSD, 10 GB+ вільного місця

GitLab не підтримує встановлення на Windows. За необхідності використання Windows як host-системи можливим рішенням є використання віртуальної машини з однією з підтримуваних Linux-систем або Docker’а. Також для використання протоколу https (secure http) необхідно мати зареєстроване домене ім’я.

Встановлення GitLab на Ubuntu

В першу чергу необхідно встановити необхідні пакети:

```
sudo apt install -y curl openssh-server ca-certificates tzdata perl
```

Наступний крок - додавання GitLab репозиторію. Існує два дістрібутиви GitLab: Enterprise Edition (gitlab-ee) та Community Edition (gitlab-ce). Додавання репозиторію виконується командою:

```
curl
https://packages.gitlab.com/install/repositories/gitlab/gitlab-
ce/script.deb.sh | sudo bash
```

Для встановлення Enterprise Edition слід використовувати ім’я gitlabl-ee.

Після додавання репозиторію встановлення GitLab виконується стандартною командою Ubuntu apt:

```
sudo EXTERNAL_URL="http://your_domain.com" apt install gitlab-ce
```

Для локального використання в якості EXTERNAL_URL можна вказати http://localhost.

Після встановлення необхідно запустити початкову конфігурацію:

```
sudo gitlab-ctl reconfigure
```

Початкове налаштування GitLab

Налаштування й робота з GitLab виконується за допомогою web-бразуреа.

Для початкового налаштування необхідно відкрити сторінку http://your_domain.com:

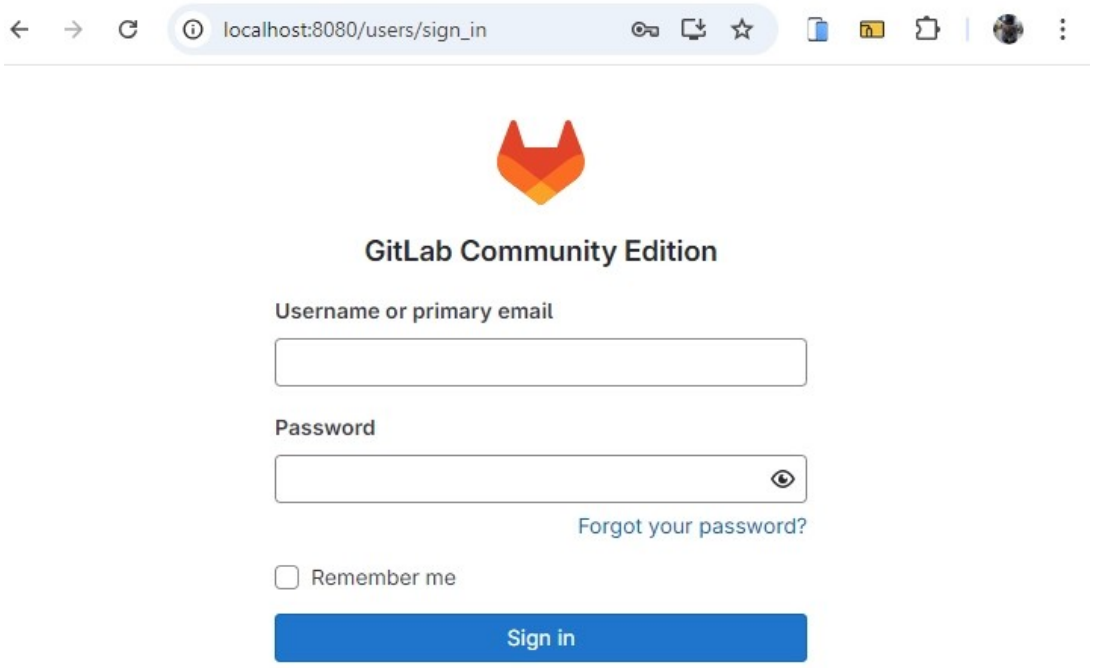

Рис. 6.1 Login-сторінка GitLab

Для входу в систему вперше GitLab надає користувача root за замовчуванням і його пароль. Стандартний пароль можна знайти в /etc/gitlab/initial_root_password.

Як правило, під час налаштувань виконуються наступні задачі:

5. Створення групи (organization)
6. Додавання користувачів
7. Створення першого репозиторію
8. Налаштування доступу (role-based permissions)

Робота з git-репозиторіями

Для створення нового репозиторію необхідно авторизуватися і натиснути кнопку «New Project». Після чого слід обрати один з можливих варіантів:

- Create blank project — створити з нуля
- Import project — імпортувати з GitHub чи іншого джерела
- Template — використати шаблон

Вибравши необхідний варіант потрібно буде вказати додаткову інформацію про створюваний проєкт:

11. Назву проєкту
12. Опис (необов'язково)
13. Рівень видимості:
    1. Private — лише для вас/команди
    2. Internal — для зареєстрованих користувачів
    3. Public — для всіх

GitLab дозволяє працювати зі створеним проєктом за допомогою веб-інтерфейсу або клонувати його в потрібне місце. Клонування виконується наступною командою:git clone https://your_domain.com/your-username/your-project.git

Інший можливий варіант ініціалізації створеного проєкту - завантаження існуючого локального репозиторію:

```
cd your_project
git init
git remote add origin https://your_domain.com/your-username/your-project.git
git add .
```

```
git commit -m "Initial commit"
git push -u origin master
```

Після клонування робота з проєктом виконується стандартним чином

Завдання

1. Встановити GitLab.
2. Створити проєкт і клонувати його.
3. Створити в проєкті файли, зробити коміт і відправити зміни в gitlab.

Лабораторна робота №7. Додаткові можливості мови shell

Тема роботи: додаткові можливості мови shell.
Мета роботи: ознайомитись з додатковими можливостями bash для написання сценаріїв CI/CD.

Теоретичні відомості

Bash підтримує масиви і дозволяє використовувати цикл for для їх обробки:

```
files=(file1.txt file2.txt file3.txt)
for file in "${files[@]}"; do
  echo "Processing $file"
done
```

Розширені умови оператора if з [[дозволяють будувати складні вирази:

```
if [[ "$filename" == *.log && -s "$filename" ]]; then
  echo "$filename is a non-empty log file"
fi
```

Bash містить набір операторів для перевірки різних властивостей файлів.

Змінні середовища (environment variables) — це пари ключ-значення, що зберігають налаштування операційного середовища, доступні всім процесам, запущеним із Bash.

Створення змінної відбувається наступним чином:

```
export ENVIRONMENT=production
```

Використання змінної:

```
echo "Current environment: $ENVIRONMENT"
```

Більш зручний і надійний спосіб — використання спеціального файлу (як правило .env), в якому визначаються змінні, які використовуються після читання файлу.

Приклад файлу .env:

```
DB_HOST=localhost
DB_USER=admin
```

Даний файл може бути прочитаний наступним чином:

```
set -a
source .env
```

```
set +a
```

Команда set -a призводить до того, що прочитанні змінні автоматично експортуються. Команда source читає вказаний файл і інтерпретує його як bash-скрипт.

Використання файлів .env — це стандартний спосіб зберігання чутливої інформації, такої як паролі. Як правило, в репозиторії проєкту зберігається приклад .env-файлу, який треба заповнити реальними даними при розгортанні проєкту. Хмарні сервіси, такі як git, дозволяють задавати в налаштуваннях змінні середовища для зберігання чутливої інформації.

Завдання

1. Створити скрипт, що перевіряє властивості вказаних файлів:

- файл існує
- файл доступний для читання
- файл доступний для запису
- файл має ненульовий розмір
- файл є виконуваним

Імена файлів мають задаватися за допомогою масиву.

2. Створити .env файл із конфігурацією:

URL=some_internet_address

APP_PORT=3000

Написати Bash-скрипт, який читає .env, запускає просту перевірку curl на порт APP_PORT.

Лабораторна робота №8. Встановлення і налаштування Jenkins

Тема роботи: встановлення і налаштування Jenkins.
Мета роботи: встановити Jenkins й навчитись створювати Jenkins pipelines.

Теоретичні відомості

Jenkins - це сервер безперервної інтеграції з відкритим кодом, написаний на Java для організації ланцюга дій для досягнення процесу безперервної інтеграції в автоматизований спосіб. Дженкінс підтримує повний життєвий цикл розробки програмного забезпечення від створення, тестування, документування програмного забезпечення, розгортання та інших етапів життєвого циклу розробки програмного забезпечення.

Jenkins можна встановити на систему під керуванням Windows або будь-якою Unix-подібною ОС. Для встановлення необхідна наступна комбінація апаратного і програмного забезпечення:

- Сервер із Unix-подібною ОС або Windows
- Користувач sudo без права root (або адміністратор)
- SSH-доступ до сервера
- Доступ до командного рядка
- Принаймні 256 МБ оперативної пам’яті
- 4+ ГБ оперативної пам’яті для групового використання
- 1 ГБ дискового простору для індивідуального використання або 10 ГБ, якщо Jenkins працює в контейнері Docker
- 50+ ГБ дискового простору для групового використання
- Java Development Kit 11 або 17
- NGINX або Apache, встановлений та налаштований на сервері

Встановити Jenkins можна за допомогою системного пакетного менеджера. Також можливо завантажити й встановити безпосередньо з сайту Jenkins https://www.jenkins.io/download/ обравши відповідну опцію.

Pipeline — це набір етапів (stages), які визначають повний процес розробки та доставки коду. Він дає змогу автоматизувати: клонування, тестування, збірку, деплой. Пайплайни описуються за допомогою скриптів Groovy DSL.

Jenkins підтримує два типи конвеєрів: скриптовий (Scripted Pipeline) та декларативний (Declarative Pipeline). Перший варіант використовує Groovy як мову програмування і є гнучким, але складнішим. Другий - простий, читається як YAML і використовує спеціальні файли Jenkinsfile.

Приклад Jenkinsfile конвеєра, що виконує клонування вказаного репозитарію і виконує послідовно скрипти для компіляції проєкту, тестування та доставки.

```
pipeline {
    agent any
    stages {
        stage('Clone') {
            steps {
                git 'https://github.com/user/repo.git'
            }
        }
        stage('Build') {
            steps {
                sh './build.sh'
            }
        }
        stage('Test') {
            steps {
                sh './run_tests.sh'
            }
        }
        stage('Deploy') {
            steps {
                sh './deploy.sh'
            }
        }
    }
}
```

Для додавання його в Jenkins за допомогою графічного інтерфейсу (UI) потрібно виконати наступні кроки:

1. Зайти в Jenkins і натиснути кнопку «New Item»
2. Вказати назву. Наприклад, MyPipeline
3. Вказати тип: Pipeline
4. В розділі Pipeline Script вставити наведений код
5. Зберегти і натисніть Build Now

Для використання цього конвеєра разом з репозиторієм коду необхідно виконати наступні кроки:

1. Створити даний файл Jenkinsfile у корені Git-проєкту
2. В Jenkins:
    1. Натиснути кнопку «New Item» та вибрати опцію «Pipeline»
    2. В Pipeline definition вибрати "Pipeline script from SCM"
    3. Вказати тип SCM — Git та вставити репозиторій
3. Зберегти і натиснути Build Now

Завдання

1. Встановити Jenkins.
2. Створити pipeline, що клонує будь-який відкритий репозиторій, наприклад, https://github.com/xvadim/number_facts і друкує два останні коміта проєкту.

Рекомендована література

- Арнольд Роббінс. Bash. Кишеньковий довідник системного адміністратора / Арнольд Роббінс — К.: “Науковий світ”, 2022. - 152 с.
- Prem Kumar Ponuthorai. Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development / Prem Kumar Ponuthorai, Jon Loeliger - O'Reilly, 2022. – 546 с.
- Mohamed Labouard. Pipeline as Code: Continuous Delivery with Jenkins, Kubernetes, and Terraform, Mohamed Labouardy / Mohamed Labouard — Manning, 2021. - 528 с.
- Git Portal – Режим доступу: https://git-scm.com/book/uk/v2
- GitLab Portal – Режим доступу: https://about.gitlab.com/
- Jenkins Portal – Режим доступу: https://www.jenkins.io/