

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ  
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ “ЧЕРНІГІВСЬКА ПОЛІТЕХНІКА”

# ІНЖЕНЕРІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

## МЕТОДИЧНІ ВКАЗІВКИ

до виконання лабораторних робіт та самостійної роботи  
для студентів спеціальності 123 – «Комп’ютерна інженерія»

Обговорено і рекомендовано  
на засіданні кафедри  
інформаційних та комп’ютерних  
систем  
*Протокол № 1*  
*від 30 вересня 2022 р.*

Чернігів НУЧП 2022

Інженерія програмного забезпечення. Методичні вказівки до виконання лабораторних робіт та самостійної роботи для студентів спеціальності 123 – «Комп'ютерна інженерія». / Укл.: Пріла О.А., , Базилевич В.М., Андрущенко Р.Б., Чорноног О.А. – Чернігів: НУЧП, 2022. – \_\_\_ с., укр. мовою.

Укладачі: ПРІЛА ОЛЬГА АНАТОЛІЇВНА, кандидат технічних наук, доцент кафедри інформаційних та комп'ютерних систем  
БАЗИЛЕВИЧ ВОЛОДИМИР МАРКОВИЧ, завідувач кафедри інформаційних та комп'ютерних систем Національний університет “Чернігівська політехніка”, кандидат економічних наук, доцент  
АНДРУЩЕНКО РОМАН БОГДАНОВИЧ, асистент кафедри інформаційних та комп'ютерних систем  
ЧОРНОНОГ ОЛЬГА АНАТОЛІЇВНА, асистент кафедри інформаційних та комп'ютерних систем

Відповідальний за випуск: БАЗИЛЕВИЧ ВОЛОДИМИР МАРКОВИЧ, завідувач кафедри інформаційних та комп'ютерних систем Національний університет “Чернігівська політехніка”, кандидат економічних наук, доцент

Рецензент: КАЗИМИР ВОЛОДИМИР ВІКТОРОВИЧ, доктор технічних наук, професор кафедри інформаційних та комп'ютерних систем

## Зміст

ВСТУП .....	5
1 ЛАБОРАТОРНА РОБОТА № 1 ВСТАНОВЛЕННЯ ТА НАЛАШТУВАННЯ СЕРВЕРА GIT .....	6
1.1 Мета роботи .....	6
1.2 Теоретичні відомості .....	6
1.3 Порядок виконання роботи .....	22
1.4 Завдання для самостійної роботи .....	23
1.5 Структура звіту .....	23
2 ЛАБОРАТОРНА РОБОТА №2 ВИВЧЕННЯ ПРИНЦИПІВ МОДУЛЬНОГО ТЕСТУВАННЯ З ВИКОРИСТАННЯМ БІБЛІОТЕКИ JUNIT .....	24
2.1 Мета роботи .....	24
2.2 Теоретичні відомості .....	24
2.3 Порядок виконання роботи .....	28
2.4 Структура звіту .....	29
3 ЛАБОРАТОРНА РОБОТА №3 СИСТЕМИ АВТОМАТИЗОВАНОГО ЗБИРАННЯ ПРОЕКТІВ (ARASHE MAVEN) .....	30
3.1 Мета роботи .....	30
3.2 Теоретичні відомості .....	30
3.3 Порядок виконання роботи .....	38
3.4 Завдання для самостійної роботи .....	41
3.5 Структура звіту .....	43
4 ЛАБОРАТОРНА РОБОТА №4 ТЕСТУВАННЯ ПРОДУКТИВНОСТІ ВЕБ- ЗАСТОСУВАННЯ .....	44
4.1 Мета роботи .....	44
4.2 Теоретичні відомості .....	44
4.3 Порядок виконання роботи .....	45
4.4 Завдання для самостійної роботи .....	51
4.5 Структура звіту .....	51
5 ЛАБОРАТОРНА РОБОТА №5. ЗАСТОСУВАННЯ СЕРВЕРІВ НЕПЕРЕРВНОЇ ІНТЕГРАЦІЇ (CONTINUOUS INTEGRATION/ CONTINUOUS DELIVERY SERVERS) .....	52
5.1 Мета роботи .....	52
5.2 Теоретичні відомості .....	52
5.3 Порядок виконання роботи .....	53
5.4 Завдання для самостійної роботи .....	55
5.5 Структура звіту .....	55
6 ЛАБОРАТОРНА РОБОТА № 6 СИСТЕМИ КЕРУВАННЯ ВЕРСІЯМИ БАЗ ДАНИХ .....	56
6.1 Мета роботи .....	56
6.2 Теоретичні відомості .....	56

## Методичні вказівки до виконання лабораторних робіт

6.3	Порядок виконання роботи .....	60
6.4	Завдання для самостійної роботи .....	61
6.5	Структура звіту .....	61

## Вступ

Курс «Інженерія програмного забезпечення» присвячено вивченню методологій, принципів та технологій підтримки й оптимізації розробки якісного програмного забезпечення та управління програмними проєктами.

У методичних вказівках розглядаються базові теоретичні принципи та практичні аспекти адміністрування та використання систем підтримки командної розробки великих програмних проєктів, а саме: систем управління проєктами, розподілених систем управління версіями документів, інструментів автоматизованого тестування (модульного, інтеграційного та тестування навантаження), систем автоматизації зборки програмного забезпечення, систем неперервної інтеграції та систем керування версіями баз даних.

Засоби, що розглядаються, орієнтовані на використання agile-принципів розробки програмного забезпечення. Методичні вказівки орієнтовані на використання вільного програмного забезпечення.

Методичні вказівки можуть використовуватися при вивченні таких дисциплін як «Технології прикладного програмування», «Web-програмування та дизайн», «Технології проектування програмних систем» та ін. з метою налаштування та використання інфраструктури для командної розробки програмних проєктів.

## **1 Лабораторна робота № 1**

### **Встановлення та налаштування сервера git**

#### **1.1 Мета роботи**

Дослідження архітектури, синтаксису та принципів роботи з розподіленими системами керування версіями документів на прикладі git.

Створення та налаштування серверу git-репозиторію.

#### **1.2 Теоретичні відомості**

##### **Системи контролю версій**

Системи контролю версій – програмне забезпечення, що допомагає керувати змінами в програмному проєкті. Система контролю версій відстежує будь-які модифікації проєкту та зберігає їх у спеціальній базі даних таким чином, щоб у будь-який момент можна було:

- переглянути історію змін
- дізнатись дату/час та автора, який вніс відповідні зміни в проєкт
- переглянути старі версії проєкту
- відмінити будь-які зміни, які, наприклад, привели до помилок, повернувшись до більш старої версії проєкту
- зберігати декілька паралельних версій проєкту
- проводити злиття декількох паралельних версій проєкту
- інтегрувати системи контролю версій з іншим програмним забезпеченням для зменшення впливу людського фактору на процеси розгортання проєкту і т.д.

Системи контролю версій відіграють дуже важливу роль у командній розробці. Розробники зазвичай працюють в командах, постійно доповнюють, модифікують код проєкту, який в свою чергу організовується у файловій системі у вигляді дерева папок з файлами. Члени команди розробки можуть працювати *одночасно* над різними функціями проєкту, або одночасно як і над новими функціями, так і над виправленням помилок в поточній версії проєкту. Внесення змін в цьому разі, очевидно, призводить до змін файлів проєкту та змін до дерева каталогів проєкту.

Одночасна робота над проєктом іноді призводить до конфліктів рішень, оскільки зміни, внесені декількома учасниками, можуть бути не сумісними одна з одною. Система контролю версій допомагає вирішувати подібні проблеми, відстежуючи кожну зміну кожного учасника, надаючи інструменти відстеження та виправлення конфліктів. Проблеми несумісності рішень необхідно виявляти та вирішувати впорядковано, не блокуючи робіт інших команд. Крім того, будь-яка зміна в робочий проєкт може призвести до нових помилок, тому не можна відразу

публікувати нову версію програмного забезпечення до тих пір, поки вона не буде ретельно перевірена. Тому процеси тестування та розробки програмного забезпечення відбуваються паралельно, поки нова версія не буде відшліфована.

Системи контролю версій при правильному використанні забезпечують плавний і безперервний потік змін до коду від декількох учасників проекту, не блокуючи один одного.

Якщо при розробці проекту не використовувати системи контролю версій, то це може призвести до наступних проблем:

- втрата інформації: хто, коли і як вніс відповідні зміни в проект
- внесення конфліктуючих змін, які призводять до а) блокування подальшої розробки; б) неправильної роботи відразу декількох функцій проекту, зміни до яких конфліктують
- складності відстеження причин, які призвели до помилок в програмному забезпечення
- наявності декількох реалізацій функції програмного забезпечення у файлах проекту, коли робочою є лише одна із них, а всі інші тільки нагромаджують код проекту роблячи його менш зрозумілим і т.д.

### **Види систем контролю версій**

Системи контролю версій можна умовно поділити на наступні категорії:

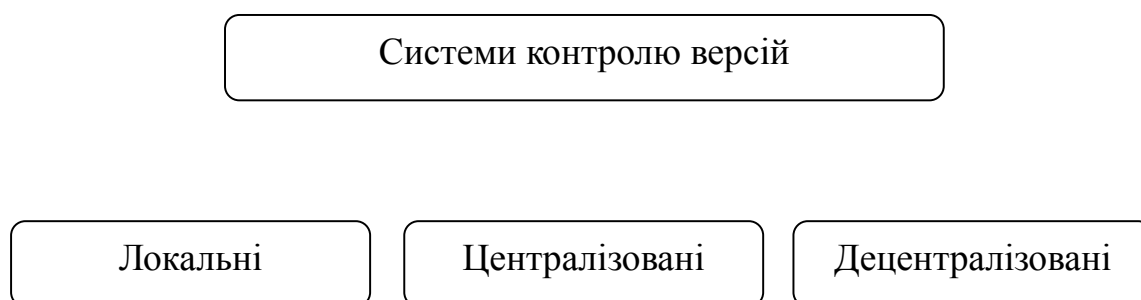


Рисунок 1.1 – Категорій систем контролю версій

Найпростішим способом контролю версій є копіювання файлів з приписом версії до каталогу з цими файлами або до самих файлів. Наприклад: «Звіт1», «Звіт2», «Звіт\_фінальний\_1» і т.д. Однак такий спосіб не є автоматизованим та призводить до багатьох помилок. Типовим рішенням в даному випадку є використання локальної системи контролю версій, яка використовує спеціальну локальну базу даних з версіями файлів.

Типовим представником такої системи контролю версій є система Revision Control System – RCS (<https://www.gnu.org/software/rcs/>)

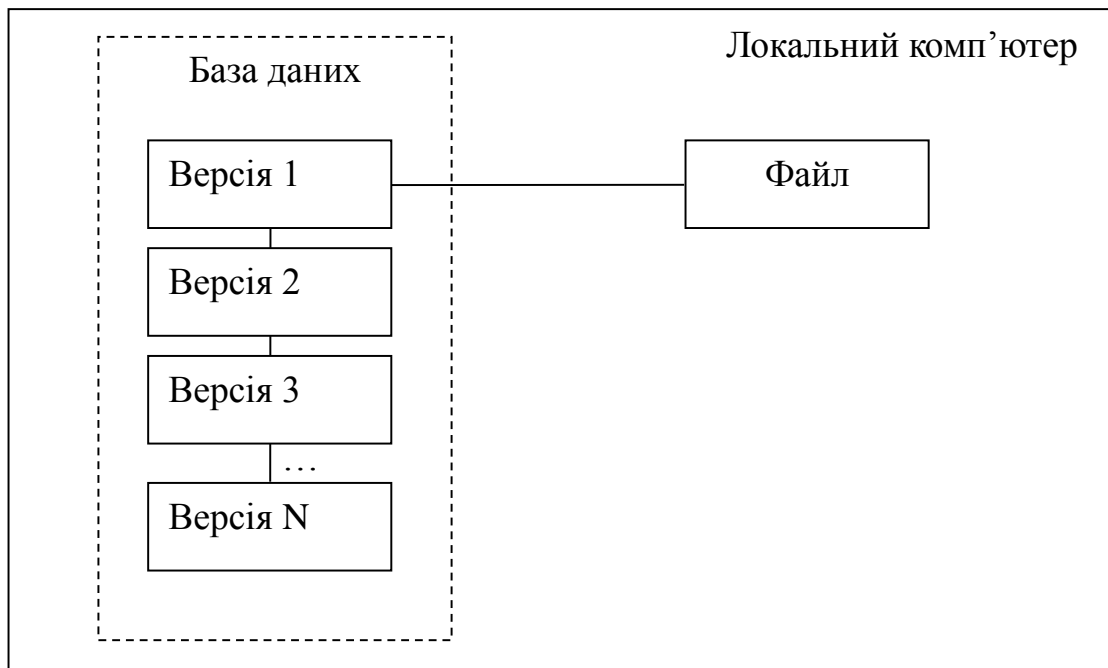


Рисунок 1.2 – Локальна система контролю версій

Локальні системи контролю версій вирішують питання управління версіями проекту, однак вони не вирішують проблему організації командної розробки, а саме – проблему одночасної роботи над проектом декількох розробників. Тому наступною віхою розвитку систем контролю версій стали централізовані системи управління версіями (CVCS).



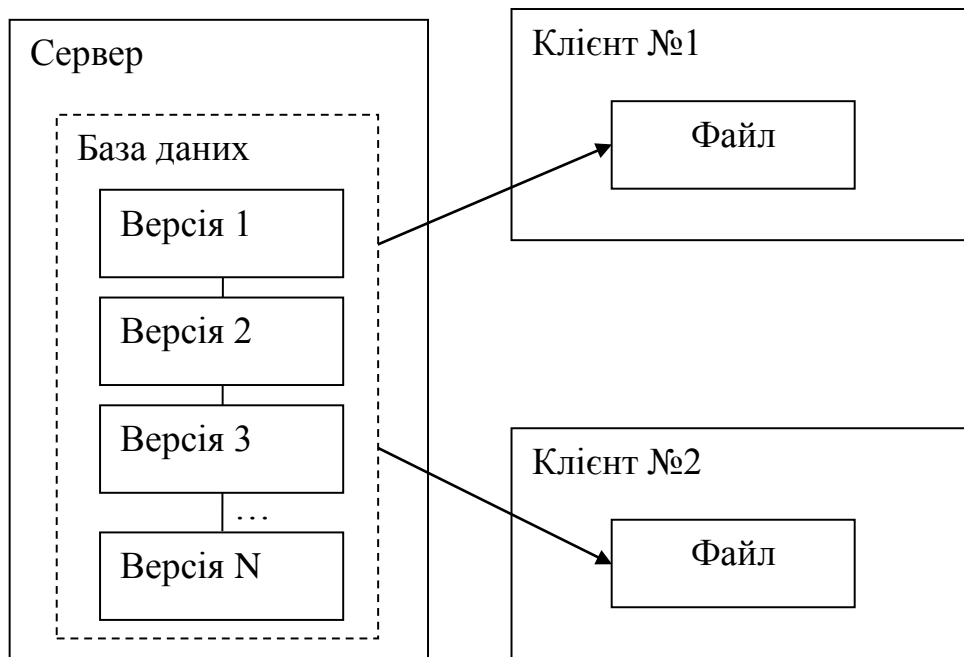


Рисунок 1.3 – Централізована система контролю версій

Централізовані системи контролю версій мають єдиний сервер з файлами проекту та декілька клієнтів, які можуть завантажувати файли з серверу і працювати над ними паралельно. Протягом багатьох років CVCS були де-факто стандартом розробки програмного забезпечення. Прикладом централізованої систем контролю версій є Apache Subversion (<https://subversion.apache.org/>).

Централізовані системи контролю версій мають переваги над локальними системами контролю версій:

- всі учасники розробки знають про зміни, які вносять інші учасники;
- адміністратори системи мають широкі можливості налаштування системи, починаючи від способу внесення змін, закінчуючи безпекою та правами доступу.

Недоліком централізованої системи контролю версій є наявність очевидної точки відмови – централізованого серверу. Якщо по тій чи іншій причині сервер не буде працювати, або щось станеться з носіями/базою даних на сервері, то виникає час простою, коли жоден учасник системи не може зберегти зміни в проект. Більш того, у разі відсутності резервних копій, є вірогідність повної втрати даних.

Даних недоліків позбавлені децентралізовані системи контролю версій (DVCS). Основна відмінність таких систем у тому, що клієнти завантажують з серверу не файли проекту, а віддзеркалюють всю базу даних серверу. По-суті кожен клієнт може замінити собою центральний сервер у разі потреби.

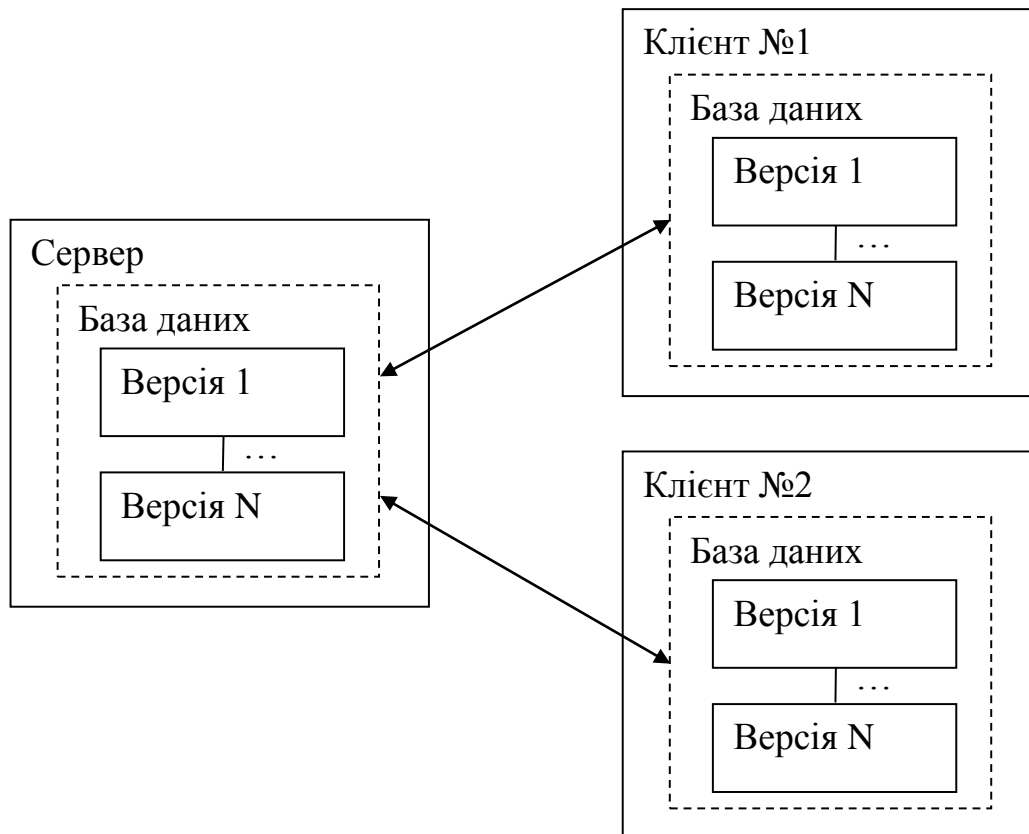


Рисунок 1.4 – Децентралізована система контролю версій

### Система контролю версій Git

Git (<https://git-scm.com/>) – децентралізована система контролю версій. На даний час одна з найбільш розповсюджених. Популярності даній системі також додають зручні хостинг-сервіси, такі як GitHub, Bitbucket.

Git вміє працювати з чотирма протоколами передачі даних: локальний, SSH, Git та HTTP(S).

1) При використанні локального протоколу репозиторієм виступає інший каталог на диску.

2) SSH – при використанні даного протоколу, репозиторій зберігається на віддаленому сервері, до якого є SSH-доступ.

3) HTTP(S) – дозволяє розміщувати репозиторії проектів на звичайних HTTP(S) веб-серверах.

4) Git-протокол – спеціально розроблений протокол, найшвидший з доступних протоколів. Недоліком Git-протоколу є відсутність аутентифікації. Зазвичай він використовується в парі з SSH-протоколом. Крім того, git-протокол складніший у налаштуванні, у порівнянні з іншими протоколами.

### Встановлення та налаштування Git-клієнта

Команди наведені далі для операційної системи Ubuntu 18.04 LTS x64.

Найпростіший спосіб встановлення Git – зі стандартного репозиторію:

```
$ sudo apt-get install git
```

Починаючи з Ubuntu версії 16.04 замість apt-get можна використовувати команду apt, тобто:

```
$ sudo apt install git
```

Далі необхідно вказати глобальні налаштування, а саме – дані автора, від імені якого будуть вноситись зміни в проекти за замовчуванням.

Конфігурація відбувається за допомогою інструменту `git config`. У якості даних конфігурації можуть виступати:

- 1) Глобальний конфігураційний файл `/etc/gitconfig`
- 2) Конфігураційний файл поточного користувача `~/.gitconfig`
- 3) Локальний конфігураційний файл `.git/config`

Отже, по-перше необхідно встановити дані поточного користувача, від імені якого будуть вноситись зміни в бази даних Git:

```
$ git config --global user.name "John Smith"
$ git config --global user.email john@stu.cn.ua
```

Наступним важливим налаштуванням є вибір текстового редактору за замовчуванням. Текстовий редактор буде викликатись кожного разу, коли системі Git необхідно отримати від користувача інформацію – наприклад, коментар до коміту, перегляд даних і т.д.

```
git config --global core.editor nano
```

Тут `nano` – ім'я текстового редактору, який можна викликати з командного рядка за допомогою команди:

```
$ nano
```

Замість `nano` (<https://www.nano-editor.org/>) можна використовувати будь-який інший текстовий редактор, наприклад `vim`.

Також рекомендується вказати інструмент порівняння файлів (diff tool), який буде використовуватись для перегляду різниці між різними версіями файлів:

```
$ git config --global merge.tool meld
```

Тут `meld` – назва інструменту порівняння (<https://meldmerge.org/>)

Всі встановлені налаштування можна переглянути за допомогою команди:

```
git config -list
```

Якщо необхідно отримати більш детальну інформацію про вище наведені або будь-які інші команди Git, то це можна зробити за допомогою команд:

```
$ git help  
$ git help <command>  
$ man git-<command>
```

### Робота з Git-клієнтом

Внесення змін до проекту в Git відбувається в декілька етапів:

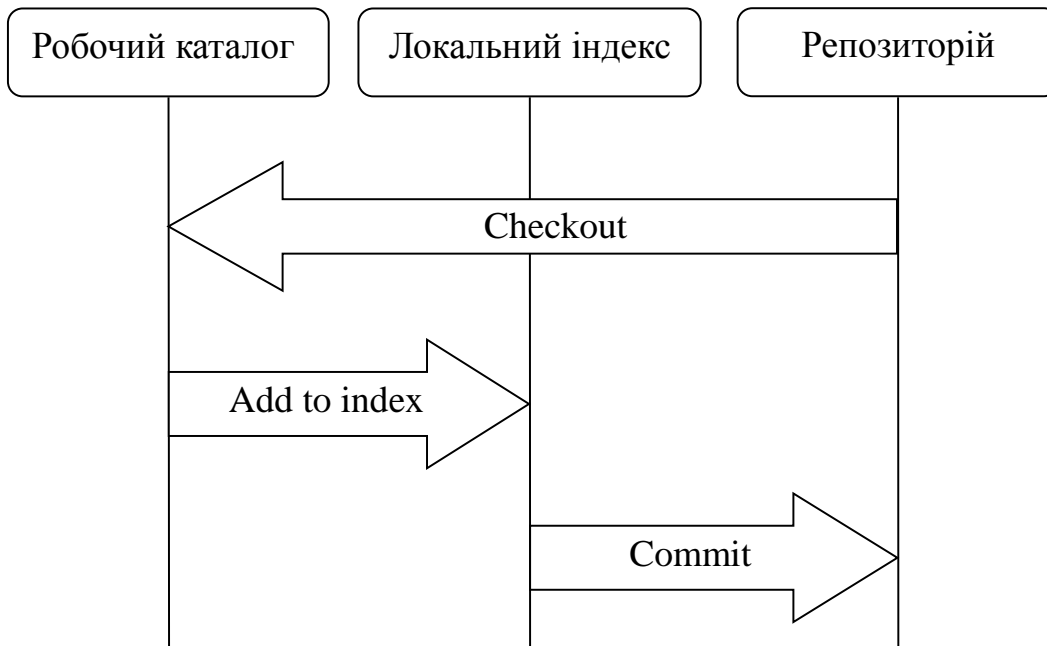


Рисунок 1.5 – Порядок внесення змін до проекту в системі Git

Для прикладу візьмемо готовий Git-репозиторій з GitHub: Google Robots.txt Parser and Matcher Library (<https://github.com/google/robotstxt>).

Для завантаження репозиторію необхідно виконати команду `clone`:

```
$ git clone https://github.com/google/robotstxt.git
```

В результаті виконання даної команди буде створено папку `robotstxt`, в якій будуть міститись вихідні коди проекту.

Альтернативою команди `clone` є створення чистого репозиторію та підключення до нього існуючого віддаленого репозиторію:

```
$ mkdir robotstxt-alt
$ cd robots-alt
$ git init
Initialized empty Git repository in
/home/rom/robotstxt-alt/.git/
$ git remote add origin \
  https://github.com/google/robotstxt.git
$ git fetch origin
From https://github.com/google/robotstxt
* [new branch]      master      -> origin/master
$ git checkout master
Branch 'master' set up to track remote branch 'master'
from 'origin'.
```

Розглянемо кожну команду більш детально:

1) **git init** – ініціалізація порожнього локального репозиторію.

2) **git remote add <remote-name> <path>** – додавання віддаленого репозиторію для відстеження змін. Тут ім'я **origin** – коротка назва віддаленого репозиторію. Ім'я `origin` – системне і зазвичай використовується для позначення будь-якого віддаленого репозиторію. Можна вказати будь-яке інше ім'я, але інші імена зазвичай використовують лише для позначення альтернативних віддалених репозиторіїв (дзеркал).

3) **git fetch <remote-name>** – завантажити вказаний віддалений репозиторій. Дані завантажуються, але не відображаються при цьому в локальному каталозі.

4) **git checkout <branch>** – перейти до вказаної гілки, після виконання даної команди, остання версія репозиторію зі вказаної гілки буде відображена в каталозі і тепер можна працювати з файлами проекту.

Гілки використовуються для ведення декількох паралельних версій, або для незалежного внесення змін декількома користувачами. За замовчуванням в репозиторії є гілка `master`, яка містить робочу актуальну останню версію проекту. В будь-який момент можна створити нову гілку від поточної:

```
$ git branch <new_branch_name>
$ git checkout <new_branch_name>
```

В результаті першої команди буде створена нова гілка, в результаті виконання другої команди, новостворена гілка стане поточною, тобто всі наступні коміти будуть вноситися саме в нову гілку. Переглянути назву поточної гілки можна за допомогою команд:

```
$ git branch
$ git status
```

Для збереження змін в git використовуються команди `git add` та `git commit`. Команда `git add` додає файл в індекс, таким чином всі зміни, внесені в файл можуть бути збережені в коміт. Можна вважати команду `git add` підготовкою до коміту. Тобто, команда `git add` – це заява про намір внести зміни. А команда `git commit` – виконати цю заяву та зберегти внесені до файлу зміни в базу даних git. Поряд внесення змін такий:

1) Редагуємо потрібні файли проекту та зберігаємо їх (на цьому зміні збережені в файлах, але не збережені в базі даних git)

2) Додаємо всі файли, які були відредаговані, в індекс за допомогою команди `git add` (можна додавати не окремі файли, а відразу каталоги)

```
$ git add file1.txt file2.txt catalog1
```

3) Створюємо коміт за допомогою команди `git commit`.

```
$ git commit -m "Added file1, file2 and catalog1"
```

Коміт зберігається у поточній гілці та відповідає певним змінам в проекті. В одній гілці може бути безліч комітів. Все це дозволяє вести одночасно декілька паралельних версій проекту, а також в будь-який момент часу повернутись до потрібного коміту.

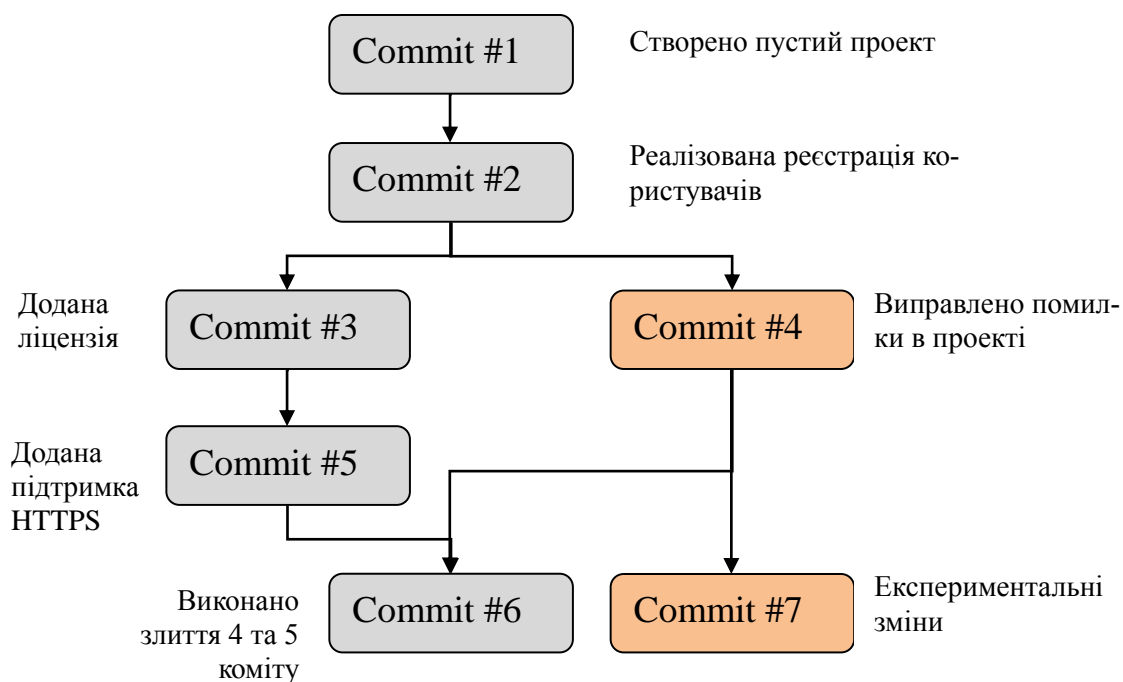


Рисунок 1.6 – Приклад історії комітів

Різними кольорами позначено різні гілки проекту. Коміти з різних гілок можуть бути включені до іншої гілки шляхом злиття (merge).

Щоб відправити гілку з комітами на віддалений репозиторій, необхідно виконати команду `git push` (необхідні права на запис):

```
$ git push <remote_repository_name> <branch_name>
```

Наприклад:

```
$ git push origin master
```

Перед виконанням команди `push` рекомендується спочатку завантажити останню версію гілки за допомогою `git fetch` + `git merge` або за допомогою `git pull` (яка є аналогом виконання відразу двох команд – `fetch` та `merge`). Тобто алгоритм відправки комітів з гілки `master` на віддалений репозиторій має бути наступним:

```
$ git fetch origin master  
$ git merge origin/master  
$ git push origin master
```

Тут: `git fetch` – завантажити з віддаленого репозиторію останню версію гілок, `git merge` – виконати злиття завантаженої гілки з локальною гілкою, `git push` – відправити локальну гілку на віддалений репозиторій.

### **Налаштування віддаленого SSH-доступу до репозиторію**

Див. <https://git-scm.com/book/en/v2/Git-on-the-Server-Setting-Up-the-Server>.

Gitosis – набір скриптів для управління файлом `authorized_keys` та організації контролю доступу до репозиторію. Для визначення налаштувань доступу до репозиторію та збереження ключей доступу користувачів використовується окремий Git-репозиторій.

Для встановлення Gitosis потрібна наявність встановлених `python-setuptools`:

```
$ aptitude install python-setuptools
```

Клонуємо та встановлюємо Gitosis з головного сайту проекту:

```
$ git clone git://eagain.net/gitosis.git  
$ cd gitosis  
$ sudo python setup.py install
```

Gitosis створює сховище репозиторіїв в директорії `/home/git`.

Gitosis автоматично керує публічними ключами для доступу до репозиторію, тому перемістимо файл `authorized_keys`, ключі розробників додамо пізніше:

```
$ mv /home/git/.ssh/authorized_keys /home/git/.ssh/ak.bak
```

Далі необхідно ініціалізувати GitoSis публічним ключем основного користувача:

```
$ sudo -H -u git gitosis-init < /tmp/id_dsa.pub
Initialized empty Git repository in /home/git/gitosis-admin.git/
Reinitialized existing Git repository in /home/git/gitosis-admin.git/
```

Це дозволить користувачу (із зазначеним публічним ключем) вносити зміни до головного Git-репозиторію для налаштування GitoSis. Необхідно налаштувати можливість виконання post-update скрипту:

```
$ sudo chmod 755 /home/git/gitosis-admin.git/hooks/post-update
```

Якщо все налаштовано вірно, можна отримати репозиторій управління доступом до Git-сервера, використовуючи ssh-доступ користувача, для якого був доданий публічний ключ при ініціалізації GitoSis.

```
# on your local computer
$ git clone git@gitserver:gitosis-admin.git
```

Після копіювання репозиторію з'явиться каталог gitosis-admin. Файл gitosis.conf використовується для визначення користувачів, репозиторіїв проектів та прав доступу до них. Директорія keydir використовується для зберігання публічних ключів всіх користувачів, які мають доступ до репозиторію.

Спочатку файл gitosis.conf містить інформацію лише для проекту gitosis-admin:

```
$ cat gitosis.conf
[gitosis]

[group gitosis-admin]
writable = gitosis-admin
members = git
```

Тобто git – єдиний користувач, який має доступ до репозиторію проекту gitosis-admin.

Наприклад, додамо новий проект. Створимо новий розділ mobile, в якому визначимо розробників команди та проекти, до яких розробники повинні мати доступ. Створимо новий проект iphone\_project. Змінимо вміст файлу gitosis.conf:

```
[group mobile]
writable = iphone_project
members = git
```

Після внесення змін до проекту gitosis-admin, необхідно зафіксувати зміни та надіслати зміни на сервер:

```
$ git commit -am 'add iphone_project and mobile group'
[master]: created 8962da8: "changed name"
```



```

1 files changed, 4 insertions(+), 0 deletions(-)
$ git push
Counting objects: 5, done.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 272 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
To git@gitserver:/home/git/gitosis-admin.git
fb27aec..8962da8 master -> master

```

Можна розпочати роботу з новим проектом `iphone_project`, вказавши його як віддалений репозиторій для локальної копії проекту. Немає необхідності у створенні "чистої" копії репозиторію для нових проектів на сервері – Gitosis створює їх автоматично при першому відправленні змін до проекту (push).

```

$ git remote add origin
git@gitserver:iphone_project.git
$ git push origin master
Initialized empty Git repository in
/home/git/iphone_project.git/
Counting objects: 3, done.
Writing objects: 100% (3/3), 230 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@gitserver:iphone_project.git
* [new branch] master -> master

```

Щоб надати доступ іншим користувачам до проекту, необхідно додати їхні публічні ключі до каталогу `keydir`.

```

$ cp /tmp/id_rsa.vasya.pub keydir/vasya.pub
$ cp /tmp/id_rsa.petya.pub keydir/petya.pub

```

Після цього можна додати нових користувачів до групи `mobile` для надання доступу (read/write) до проекту `iphone_project`:

```

[group mobile]
writable = iphone_project
members = git vasya petya

```

Для надання користувачеві `vasya` доступу до проекту тільки на читання, необхідно змінити `gitosis.conf`:

```

[group mobile]
writable = iphone_project
members = git petya

```

```

[group mobile_ro]
readonly = iphone_project
members = vasya

```

Створимо групу `mobile`, наслідуючи всіх учасників групи `mobile_committers`:

```

[group mobile_committers]
members = git vasya petya

```

```

[group mobile]
writable = iphone_project

```

```
members = @mobile_committers
```

Бажано також додати `loglevel=DEBUG` у розділі `[gitosis]`.

### **Налаштування HTTP- та Smart HTTP-доступу до репозиторію**

Розглянемо налаштування доступу до Git-репозиторію через протокол HTTP для Apache та Nginx.

1) Створюємо чистий (bare) репозиторій. Є два способи: або створити повністю пустий чистий репозиторій, або створити чистий репозиторій на базі існуючого. В першому випадку створюємо каталог та ініціюємо репозиторій наступним чином:

```
$ mkdir robotstxt && cd $_  
$ git init --bare
```

В другому випадку клонуємо потрібний репозиторій з додатковим аргументом:

```
$ git clone --bare https://github.com/google/robotstxt
```

Каталог в результаті виконання даної команди буде створено автоматично та матиме назву «<ім'я репозиторію>.git».

2) Встановлюємо сервер Apache:

```
$ sudo apt-get install apache2 apache2-utils
```

3) Після успішного виконання команди з п.п.2 сервер має бути автоматично запущений (якщо не було конфліктів з іншими серверними за стосунками). Статус серверу можна перевірити за допомогою команди:

```
$ sudo systemctl status apache2
```

Щоб знову повернутися до терміналу після виконання даної команди, треба натиснути Q.

Також Apache прослуховує порт 80 та віддає у відповідь на HTTP GET-запит веб-сторінку за замовчуванням. Для перегляду достатньо в будь-якому веб-браузері перейти за адресою «127.0.0.1».

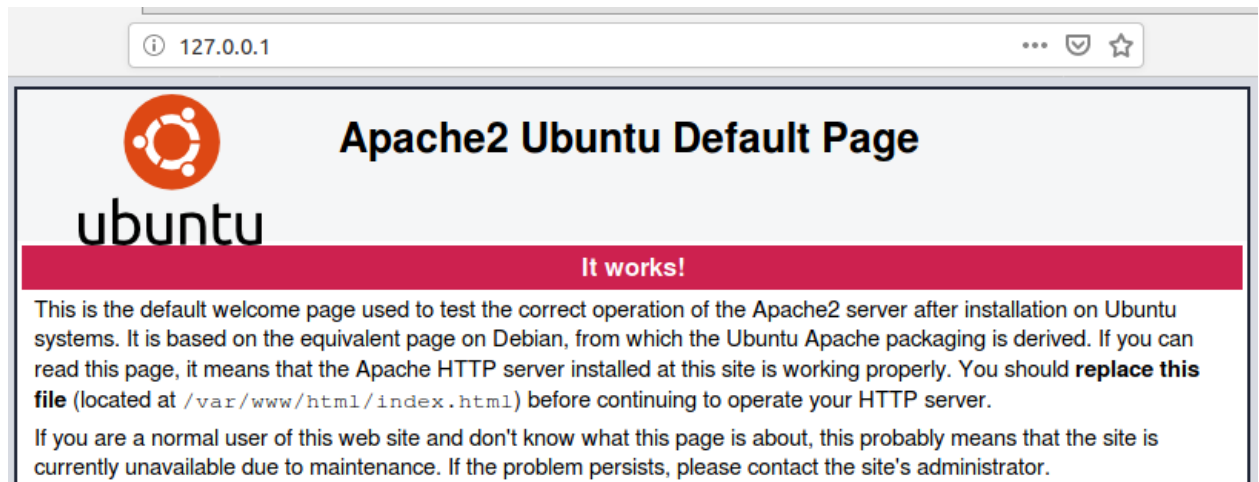


Рисунок 1.7 – Стандартна веб-сторінка веб-серверу Apache безпосередньо після встановлення пакету apache2

4) Вмикаємо необхідні додаткові модулі:

```
$ sudo a2enmod cgi alias env
$ sudo systemctl restart apache2
```

3) Налаштовуємо доступ до каталогу з репозиторіями для серверу Apache:

```
$ sudo chgrp -R www-data <path_to_root_repository_dir>
```

4) Створюємо віртуальний хост. Наприклад, нехай всі git-репозиторії будуть розташовуватись на субдоміні git.myserver.local:

- Редагуємо файл hosts:

```
$ sudo nano /etc/hosts
```

- Додаємо в кінець файлу рядки:

```
127.0.0.1 myserver.local
127.0.0.1 git.myserver.local
```

- Зберігаємо за допомогою комбінації клавіш CTRL+X, потім

Y

- Створюємо файл конфігурації віртуального хосту:

```
$ sudo nano /etc/apache2/sites-available/001-git.conf
```

- Додаємо до файлу наступну конфігурацію:

```
<VirtualHost *:80>
    ServerName git.myserver.local
```

## Методичні вказівки до виконання лабораторних робіт

```
DocumentRoot <path_to_repository_root_dir>
ErrorLog    ${APACHE_LOG_DIR}/error.log
CustomLog   ${APACHE_LOG_DIR}/access.log
combined

</VirtualHost>
```

- Вмикаємо конфігурацію:

```
$ sudo a2ensite 001-git.conf
$ sudo systemctl restart apache2
```

- Додаємо в каталог репозиторіїв файл index.html з будь-яким вмістом та перевіряємо роботу, перейшовши в браузері за адресою git.myserver.local:

```
<html>
  <body>
    <h1>Hello, world!</h1>
  </body>
</html>
```

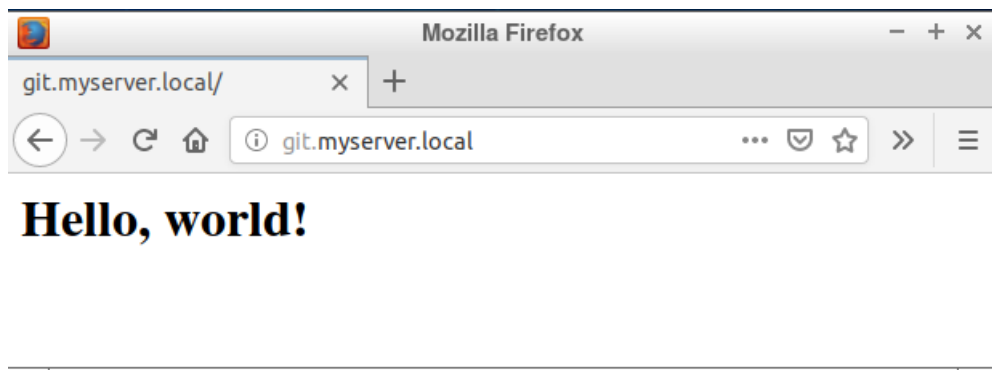


Рисунок 1.8 –Налаштований віртуальний хост

- 5) Знаходимо шлях до каталогу з бінарними файлами Git:

```
git --exec-path
```

- 6) Налаштовуємо протокол Smart HTTP шляхом редагування файлу конфігурації віртуального хосту:

```
$ cd /etc/apache2
$ sudo touch git.conf
```

До конфігурації додаємо налаштування змінних та задаємо пере направлення на CGI-скрипт:

```
SetEnv GIT_PROJECT_ROOT <path_to_repository_root_dir>
SetEnv GIT_HTTP_EXPORT_ALL
```

```
ScriptAlias /projects/ <exec_path>
```

Тут: `path_to_repository_root_dir` – шлях до кореневого каталогу з Git-репозиторіями; `exec_path` – шлях, повернутий командою з п.п. 5.

7) Налаштовуємо стандартну HTTP-аутентифікацію для доступу до Git-репозиторіїв:

- Створюємо файл `htaccess` в кореновому каталозі з репозиторіями:

```
$ htpasswd -c <path_to_repository_root>/.htpasswd  
<username>
```

Потрібно буде ввести пароль для користувача `username`

- Далі редагуємо конфігурацію віртуального хосту:

```
<Files "git-http-backend">  
    AuthType Basic  
    AuthName "Git Access"  
    AuthUserFile  
<path_to_repository_root_dir>/.htpasswd  
    Require expr !({QUERY_STRING} -strmatch  
'*service=git-receive-pack*' || {%REQUEST_URI} =~ m#/git-  
receive-pack$#)  
    Require valid-user  
</Files>
```

8) Перезавантажуємо Apache:

```
$ sudo systemctl restart apache2
```

9) Тепер до репозиторіїв є доступ через протокол HTTP та стандартну HTTP-аутентифікацію. Репозиторій можна клонувати, можна відправити локальні коміти на сервер і т.д. за допомогою звичних клієнтських команд: `git clone`, `git push` і т.д.

**Налаштування веб-доступу на перегляд вмісту репозиторіїв за допомогою Gitweb**

Налаштуємо веб-інтерфейс для відображення вмісту репозиторію з використанням GitWeb. Для цього необхідно отримати вихідний код GitWeb та згенерувати CGI-скрипт:

```
$ git clone git://git.kernel.org/pub/scm/git/git.git  
$ cd git/  
$ make GITWEB_PROJECTROOT="/home/git" \  
    prefix=/usr gitweb/gitweb.cgi  
$ sudo cp -Rf gitweb /var/www/
```

Змінна `GITWEB_PROJECTROOT` повинна вказувати на директорію, де потрібно шукати Git-репозиторії проектів.

Необхідно, щоб Apache використовував CGI для цього скрипту. Додамо VirtualHost до конфігураційного файлу Apache:

```
<VirtualHost *:80>
    ServerName gitserver
    DocumentRoot /var/www/gitweb
    <Directory /var/www/gitweb>
        Options          ExecCGI          +FollowSymLinks
+SymLinksIfOwnerMatch
        AllowOverride All
        order allow,deny
        Allow from all
        AddHandler cgi-script cgi
        DirectoryIndex gitweb.cgi
    </Directory>
</VirtualHost>
```

За адресою <http://gitserver/> можна переглянути репозиторій проєкту, а вказуючи <http://git.gitserver> в якості шляху до репозиторію, можна копіювати репозиторій по протоколу HTTP.

Додаткова інформація щодо налаштування GitWeb наведена <https://git-scm.com/book/en/v2/Git-on-the-Server-GitWeb>.

### **1.3 Порядок виконання роботи**

#### **Вивчення клієнтської частини системи git.**

- 1) Створити пустий git-репозиторій (створити локальний репозиторій проєкту на власній машині або створити репозиторій проєкту на публічному сервісі <https://github.com/>).
- 2) Налаштувати права доступу до репозиторію.
- 3) Виконати клонування репозиторію на клієнтську машину;
- 4) Додати файли будь-якого проєкту (наприклад, вихідний код будь-якого курсового проєкту) до локального репозиторію проєкту.
- 5) Виконати коміт, який включає всі додані файли проєкту. За замовчуванням буде створена гілка master.
- 6) Відправити коміт з внесеними змінами на віддалений репозиторій.
- 7) Створити дві локальні гілки проєкту та створити по одному додатковому коміту в цих гілках.
- 8) Виконати злиття змін, створених в гілках, в гілку master.
- 9) Переконайтесь в наявності змін на гілці master.
- 10) Внести на гілках проєкту конфлікуючі зміни (таких, щоб при злитті змін в гілку master утворився конфлікт).
- 11) Виконати злиття змін, створених в гілках, в гілку master.

- 12) Виправити конфлікт. Виконати додатковий коміт після виправлення конфлікту.
- 13) Відправити коміт з внесеними змінами на віддалений репозиторій.
- 14) Вивчити принцип роботи та перевірити виконання інших команд клієнту git.

### **Налаштування серверної частини git.**

- 1) Встановити git.
- 2) Налаштувати read/write-доступ до репозиторіїв git для різних користувачів за допомогою протоколів ssh або http.
- 3) Налаштувати веб-доступ для перегляду вмісту репозиторію за допомогою Gitweb.
- 4) Перевірити виконання сесії клієнта (п. Вивчення клієнтської частини системи git) для створеного репозиторію.

### **1.4 Завдання для самостійної роботи**

Ознайомитись із додатковим синтаксисом команд git (<https://git-scm.com/book/en/v2>).

### **1.5 Структура звіту**

- навести результати виконання команд процесу створення git-репозиторія та надання доступу до нього;
- навести вміст основних конфігураційних файлів;
- навести результати виконання команд сесії роботи з git-репозиторієм.

## **2 Лабораторна робота №2**

### **Вивчення принципів модульного тестування з використанням бібліотеки JUnit**

#### **2.1 Мета роботи**

Навчитися розробляти Unit-тести з використанням інструментальної бібліотеки JUnit. Розробити тести для одного зі стандартних класів JDK.

#### **2.2 Теоретичні відомості**

Тестування програмного забезпечення – процес виявлення помилок в програмному забезпеченні. Виділяють види тестування згідно таких критеріїв:

- ступень ізольованості компонентів: модульне, інтеграційне, системне тестування;
- об'єкт тестування: функціональне, тестування навантаження, тестування інтерфейсу; тестування безпеки;
- наявність інформації про структуру та функціонування системи: тестування «чорного», «білого» ящика;
- час проведення: альфа-, бета-тестування, приймальне тестування, регрес тестування;
- ступінь автоматизації: ручне, автоматизоване, напівавтоматизоване;
- виконання кода: статичне та динамічне тестування.

Кінцевою метою будь-якого процесу тестування є забезпечення якості програмного забезпечення. При цьому жоден вид тестування не надає 100% гарантії працездатності програми.

Планування тестових сценаріїв має бути виконано таким чином, щоб перевірялися різні варіанти використання програми.

Модульне тестування передбачає перевірку працездатності найменших складових програми: зазвичай це окремий клас і його методи. У випадках, якщо при виконанні методів тестованого класу відбувається виклик методів стороннього класу, необхідно відокремити тестовані класи один від одного, замінити їх за допомогою mock-об'єктів, заглушок, імітацій. Наприклад, якщо тестується метод класу, який приймає ім'я і пароль користувача, а потім звертається до бази даних для перевірки правильності введених користувачем даних аутентифікації (інший клас), то потрібно замінити клас, який працює з реальними даними «імітацією», яка не виконує звернення до бази даних, а зберігає дані користувачів всередині самого класу. Імітація повинна бути досить проста, щоб не містити помилок - в іншому випадку говорити про модульному тестуванні не можна.



JUnit - популярний в даний час інструмент модульного тестування java-додатків. JUnit 4 заснований на використанні анотацій.

Наприклад, для тестування правильності виконання методів класу Calculator, необхідно створити наступний тестовий клас TestCalculator:

```
import org.junit.Test;
import static org.junit.Assert.*;

public class TestCalculator {

    public Calculator calculator;

    @Before
    public void prepareTestData() {
        calculator = new Calculator ();
        ...
    }

    @Before
    public void setupMocks() { ... }

    @After
    public void cleanupTestData() { ... }

    @BeforeClass
    public static void setupDatabaseConnection() { ... }

    @AfterClass
    public static void teardownDatabaseConnection() { ... }

    @Test
    public void testAdd() {
        assertEquals(4, calculator.add(1, 3) );
        assertEquals(2, calculator.add( -3, 5 ) );
        assertEquals(0, calculator.add(0, 0) );
        assertEquals(-10, calculator.add(-4, -6) );
        ...
    }

    @Test
    public void testDivide() {
        ....
    }

    @Test(expected=ArithmeticException.class)
    public void testDivisionByZero() {
        calculator.divide(4, 0);
    }
    ...
}
```

Рекомендується (але необов'язково) використання стандарту іменування тестового класу і його методів (префікс test).

Для позначення тестового методу використовується анотація @Test. Метод assertEquals перевіряє рівність очікуваного і отриманого значення. Інші подібні методи - assertTrue, assertFalse, assertNull, assertNotNull, assertEquals.

Анотацією @Before позначають методи, які повинні бути запущені перед запуском кожного тестового методу, анотацією @After - після запуску кожного тестового методу відповідно. Якщо в тестовому класі визначено кілька методів з анотацією @Before, порядок запуску методів може бути довільним.

Анотаціями @BeforeClass і @AfterClass позначають методи, які повинні бути виконані до і відповідно після виконання всього набору тестів класу. Ці методи повинні бути оголошені як статичні.

Плани тестів повинні включати і «неправильні» варіанти використання системи, наприклад, неприпустимі вхідні параметри методу. Тестування того, що код коректно працює в виняткових ситуаціях, також важливо, як і тестування функціональності коду. Для тестування коректності роботи методу в виняткової ситуації необхідно оголосити очікуване виключення в анотації @Test.

Параметр timeout анотації @Test дозволяє вказати тимчасові обмеження виконання тестового методу:

```
@Test(timeout=5000)
public void testOperation() {    ...    }.
```

У деяких випадках потрібно відключення тестового методу класу. Для цього використовується анотація @Ignore. Як параметр бажано вказати текст повідомлення, яке буде виведено при запуску тестових методів класу.

```
@Ignore("Not running because <fill in a good reason here>")
```

```
@Test
public void testOperation(){    ...    }.
```

Набір тестів можна визначити таким чином:

```
@RunWith(value=Suite.class)
@SuiteClasses(value={CalculatorTest.class,
AnotherTest.class})
public class AllTests {
    ...
}
```

JUnit 4 вводить можливість визначення різних класів для запуску тестів. Анотація @RunWith визначає використання класу Suite для за-

пуску тестів. У свою чергу, анотація `@SuiteClasses` використовується класом `Suite` для визначення тестів, які входять в даний набір. При визначенні набору тестів допускається можливість використання методів, анотованих за допомогою `@BeforeClass` і `@AfterClass`, які будуть викликатися перед початком запуску всіх тестів і після виконання останнього тесту з набору відповідно. Таким чином, вона може змінювати необхідних параметрів відразу для цілого набору тестів.

Клас `Parameterized` бібліотеки `JUnit 4` дозволяє запускати одні і ті ж тести, але з різними наборами даних. Розглянемо це на прикладі. Нижче наведено приклад тесту для методу, який обчислює факторіал заданого числа `n`:

```
@RunWith(value=Parameterized.class)
public class TestFactorial {
    private long expected;
    private int value;
    @Parameters
    public static Collection data() {
        return Arrays.asList( new Object[][] {
                                { 1, 0 },    // expected, value
                                { 1, 1 },
                                { 2, 2 },
                                { 24, 4 },
                                { 5040, 7 },
                            });
    }
    public TestFactorial(long expected, int value) {
        this.expected = expected;
        this.value = value;
    }
    @Test
    public void factorial() {
        Calculator calculator = new Calculator();
        assertEquals(expected,
            calculator.factorial(value));
    }
}
```

Клас `Parameterized` запускає всі тести класу `FactorialTest` (в прикладі він тільки один), використовуючи при цьому дані методів, промаркованих анотацією `@Parameters`. В даному випадку маємо список з п'ятьма елементами. Кожен елемент містить масив, який буде використаний в якості аргументів конструктора класу `FactorialTest`. Тест `factorial()` використовує ці дані при виклику методу `assertEquals()`. Тест буде виконаний 5 разів з використанням наступних даних:

```
factorial#0: assertEquals(1,calculator.factorial(0));
```

```
factorial#1: assertEquals(1, calculator.factorial(1));
factorial#2: assertEquals(2, calculator.factorial(2));
factorial#3: assertEquals(24, calculator.factorial(4));
factorial#4: assertEquals(5040, calculator.factorial(7));
```

## 2.3 Порядок виконання роботи

Створити тестовий клас і тести для перевірки належного функціонування методів класу. Перевірці підлягають не менше 10 методів класу на вибір.

Класи для тестування вибираються відповідно таблиці 1.1.

Таблиця 1.1 – Вибір класу для тестування

Група 1		Група 2	
№ п/п	Клас	№ п/п	Клас
1	java.util.ArrayList	1	java.lang.Enum
2	java.lang.Byte	2	java.util.HashMap
3	java.io.BufferedReader	3	java.util.Scanner
4	java.math.BigInteger	4	java.util.Vector
5	java.text.BreakIterator	5	java.lang.Long
6	java.util.Arrays	6	java.io.FileOutputStream
7	java.lang.Boolean	7	java.util.LinkedList
8	java.io.BufferedWriter	8	java.io.ObjectInputStream
9	java.text.DecimalFormat	9	java.io.ObjectOutputStream
10	java.util.BitSet	10	java.util.LinkedHashSet
11	java.lang.Character	11	java.util.HashTable
12	java.io.DataInputStream	12	java.util.jar.Manifest
13	java.math.BigDecimal	13	java.lang.reflect.Constructor
14	java.text.SimpleDateFormat	14	java.util.jar.Attributes
15	java.util.Calendar	15	java.io.FileInputStream
16	java.util.StringTokenizer	16	java.lang.reflect.Method
17	java.lang.Double	17	java.math.BigDecimal
18	java.io.DataOutputStream	18	java.io.File
19	java.io.File	19	java.text.BreakIterator
20	java.util.HashSet	20	java.io.StringReader
21	java.lang.StringBuffer	21	java.lang.Float
22	java.util.Collections	22	java.util.PriorityQueue
Група 3		Група 4	
1	java.text.StringCharacterIterator	1	java.lang.Math
2	java.lang.String	2	java.io.DataOutputStream
3	java.lang.Math	3	java.io.StringReader
4	java.io.LineNumberReader	4	java.util.ArrayList
5	java.io.StringReader	5	java.util.Calendar

6	java.util.PriorityQueue	6	java.util.BitSet
7	java.util.Stack	7	java.util.Collections
8	java.lang.Integer	8	java.util.Stack
9	java.util.LinkedHashMap	9	java.lang.Double
10	java.lang.Float	10	java.io.BufferedWriter
11	java.lang.Short	11	java.util.Arrays
12	java.lang.reflect.Field	12	java.lang.Short
13	java.util.jar.Attributes	13	java.lang.String
14	java.util.Stack	14	java.lang.Short
15	java.util.PriorityQueue	15	java.lang.reflect.Field
16	java.util.LinkedHashMap	16	java.lang.Math
17	java.util.HashSet	17	java.lang.Integer
18	java.util.Collections	18	java.lang.reflect.Constructor
19	java.util.Calendar	19	java.lang.Double
20	java.util.BitSet	20	java.util.jar.Manifest
21	java.util.Arrays	21	java.lang.Byte
22	java.util.ArrayList	22	java.lang.Boolean
<b>Extra</b>			
1	java.text.StringCharacterIterator	8	java.io.StringReader
2	java.text.SimpleDateFormat	9	java.io.LineNumberReader
3	java.text.DecimalFormat	10	java.io.File
4	java.text.BreakIterator	11	java.io.DataOutputStream
5	java.lang.reflect.Method	12	java.io.DataInputStream
6	java.math.BigDecimal	13	java.io.BufferedWriter
7	java.lang.StringBuffer	14	java.io.BufferedReader

## 2.4 Структура звіту

- конфігурування середовища тестування;
- перелік та API методів класу, що підлягає тестуванню;
- листинги вихідного коду тестів (або посилання на тести);
- результати виконання тестів (скріншоти, консольний вивід);
- висновки.

### **3 Лабораторная работа №3**

#### **Системи автоматизованого збирання проектів (Apache Maven)**

##### **3.1 Мета роботи**

Вивчення циклу збирання проектів. Практичне застосування інструменту Maven.

##### **3.2 Теоретичні відомості**

Процес компіляції, тестування, збирання та розгортання проекту здійснюються неодноразово протягом усього життєвого циклу програмного забезпечення. Здійснювати цей процес вручну при великому числі вихідних файлів, з огляду на залежність між модулями проекту, а також залежно від сторонніх бібліотек, необхідних для компіляції, практично неможливо. Крім того, процес збирання проекту повинен відбуватися незалежно від середовища розробки. Цикл збирання також повинен включати етап тестування з генерацією відповідних звітів.

Процес створення готового дистрибутива програмного забезпечення також вимагає автоматизації, тому що існує величезна кількість форматів створюваних пакетів в залежності від операційної системи, платформи, дистрибутива і т.д.

Системи збирання проектів якраз і призначені для автоматизації вищевказаних процесів. Приклади подібних систем – Make, Autotools, Qmake, Cmake, Ant, Maven, Gradle.

Maven – не тільки інструмент збирання проектів, а й інструмент декларативного опису проекту. Maven вводить поняття об'єктної моделі проекту (Project Object Model – POM). За допомогою цієї моделі можна задати опис проекту, його унікальні ідентифікатори, визначити перелік залежностей та вказати деякі інші атрибути.

Використання подібної моделі має такі переваги.

1. Управління залежностями. Оскільки кожному проекту ставляться у відповідність унікальні координати (groupId:artifactId:version), то ці ж координати використовуються для визначення залежностей.

Можливість автоматичного пошуку та завантаження потрібних бібліотек-артефактів із віддалених репозиторіїв реалізовано завдяки унікальним координатам артефактів. Існує можливість організації власного репозиторію артефактів та відповідного налаштування репозиторію проекту.

2. Дистанційні репозиторії. Координати, що визначаються в POM-моделі, використовуються для індексування і пошуку потрібних артефактів в maven-репозиторіях.

3. Стандартизація циклу збирання проекту. Модулі maven (плагіни) реалізують послідовність дій для різних фаз циклу збирання проектів і організують цей процес відповідно до опису проекту. Налаштування плагінів здійснюється також за допомогою декларативного конфігурування в POM-моделі проекту.

4. Інтеграція з різними середовищами розробки.

Середовища розробки (IntelliJ, Eclipse) створюють свої власні файли проектів, в яких зберігають інформацію про проект, причому для різних IDE ці файли різні. Maven стандартизує опис проекту і дозволяє згенерувати файли проекту для того чи іншого середовища розробки на основі моделі проекту.

Плагін eclipse дозволяє виконати генерацію проекту для середовища розробки eclipse. Налаштування плагіну можна додати в головний файл проекту pom.xml.

```
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-eclipse-plugin</artifactId>
<configuration>
    <downloadSources>true</downloadSources>
    <dependenciesAsLibraries>true</dependencies
AsLibraries>
    <useFullNames>false</useFullNames>
</configuration>
</plugin>
```

Після виконання команди `mvn eclipse:eclipse`, maven створює файли проекту, які необхідні для роботи в середовищі розробки eclipse. Для повної підтримки maven в Eclipse необхідно завантажити відповідний плагін, але для виконання даної лабораторної роботи це обов'язково.

5. Простий пошук і завантаження необхідних артефактів. Можливість автоматичного пошуку і завантаження потрібних бібліотек-артефактів з віддалених репозиторіїв завдяки унікальним ідентифікаторам, а також існує можливість організації власного сховища артефактів і налаштування репозиторіїв для проекту.

Розробникам не потрібно згадувати кожну деталь конфігурації. Maven забезпечує стандарту поведінку за замовчуванням для проектів; коли створюється проект. Maven ініціалізує відповідну стандартизовану структуру проекту, а розробникам необхідно лише правильно розмістити файли вихідних кодів та ресурсів по каталогах.

В таблиці нижче показані значення каталогів за замовчуванням:

Таблиця 2.1

<b>Вихідні коди проекту</b>	<b>./src/main/java</b>
-----------------------------	------------------------

<b>Тести</b>	<b>./src/tests</b>
<b>Ресурси</b>	<b>./src/main/resources</b>
<b>Зкомпільований байт-код</b>	<b>./target</b>

Maven можна встановити з бінарних пакетів, які поставляються з стандартного репозиторію Debian, Ubuntu. Для встановлення найактуальнішої версії необхідно зайти на сайт <https://maven.apache.org/> та дотримуючись інструкцій завантажити необхідні архіви.

Перевірка коректності встановлення:

```
$ mvn --version
Apache Maven 3.6.0
Maven home: /usr/share/maven
Java version: 11.0.4, vendor: Ubuntu, runtime:
/usr/lib/jvm/java-11-openjdk-amd64
Default locale: en_US, platform encoding: UTF-8
OS name: "linux", version: "5.0.0-27-generic",
arch: "amd64", family: "unix"
```

Створити новий проект можна двома шляхами:

- 1) Вручну
- 2) За допомогою готових шаблонів (archetypes)

У першому випадку необхідно створити конфігураційний POM-файл. Мінімальний вміст файлу наступний:

```
<project
  xmlns = "http://maven.apache.org/POM/4.0.0"
  xmlns:xsi
"http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation
"http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-
4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>ua.cn.stu.lab2</groupId>
  <artifactId>lab2-maven</artifactId>
  <version>1.0</version>
</project>
```

Теги groupId, artifactId, version задають унікальний ідентифікатор проекту, який ще записують у наступному форматі:

```
groupId:artifactId:version
```



Після задання конфігурації необхідно створити структуру каталогів проекту згідно таблиці 2.1.

У другому випадку Maven автоматизує процес створення нового проекту. Наприклад, використаємо шаблон `maven-archetype-quickstart` (необхідний доступ до мережі Інтернет; також перший запуск даної команди може виконуватись тривалий час, оскільки Maven завантажує необхідні залежності до локального кешу):

```
$ mvn archetype:generate \
    -DgroupId=ua.cn.stu.lab2 -
DartifactId=lab2-maven \
    -DarchetypeArtifactId=maven-archetype-
quickstart \
    -DarchetypeVersion=1.4 -
DinteractiveMode=false
```

У разі успіху буде створено каталог `lab2-maven`, а в консолі буде виведено результат виконання команди: `build success`. Поточну конфігурацію можна переглянути, відкривши файл `pom.xml`:

```
$ cd lab2-maven && nano pom.xml
```

Поточна конфігурація:

```
<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0
.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>ua.cn.stu.lab2</groupId>
  <artifactId>lab2-maven</artifactId>
  <version>1.0-SNAPSHOT</version>
  <name>lab2-maven</name>
  <url>http://www.example.com</url>
  <properties>
    <project.build.sourceEncoding>UTF-
8</project.build.sourceEncoding>
```

```

    <ma-
ven.compiler.source>1.7</maven.compiler.source>
    <ma-
ven.compiler.target>1.7</maven.compiler.target>
  </properties>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <build>
    <pluginManagement>
      <plugins>
        <plugin>
          <artifactId>maven-clean-
plugin</artifactId>
          <version>3.1.0</version>
        </plugin>
        <plugin>
          <artifactId>maven-resources-
plugin</artifactId>
          <version>3.0.2</version>
        </plugin>
        <plugin>
          <artifactId>maven-compiler-
plugin</artifactId>
          <version>3.8.0</version>
        </plugin>
        <plugin>
          <artifactId>maven-jar-plugin</artifactId>
          <version>3.0.2</version>
        </plugin>
        <plugin>
          <artifactId>maven-install-
plugin</artifactId>
          <version>2.5.2</version>
        </plugin>
        <plugin>
          <artifactId>maven-deploy-
plugin</artifactId>
          <version>2.8.2</version>

```

```

        </plugin>
        <plugin>
            <artifactId>maven-site-
plugin</artifactId>
            <version>3.7.1</version>
        </plugin>
        <plugin>
            <artifactId>maven-project-info-reports-
plugin</artifactId>
            <version>3.0.0</version>
        </plugin>
    </plugins>
</pluginManagement>
</build>
</project>

```

Всі конфігурації Maven успадковуються від базової конфігурації Super POM. Конкретні конфігурації конкретних проектів наслідують базову конфігурацію і, таким чином, містять лише мінімально необхідну інформацію для збирання проекту, яка лише уточнює базову конфігурацію Super POM.

Для перегляду конфігурації Super POM необхідно виконати в каталозі проекту команду:

```
$ mvn help:effective-pom
```

### Життєвий цикл збирання проекту

Життєвий цикл – чітко визначена послідовність **фаз** (phases, див. таблицю 2.2), яка визначає порядок виконання **цілей** (goals), тобто певних задач, необхідних для збирання проекту. Фаза – це етап життєвого циклу. Наприклад, типовий життєвий цикл збирання проектів складається з наступної послідовності фаз.

Таблиця 2.2

Фаза	Пояснення
<b>prepare-resources</b>	Підготовка ресурсів проекту
<b>validate</b>	Валідація проекту на відповідність заданим правилам
<b>compile</b>	Компіляція вихідних кодів проекту в байт-код
<b>test</b>	Запуск тестів та генерація звіту за результатами тестування
<b>package</b>	Створення JAR/WAR файлів
<b>install</b>	Встановлення створеного на етапі package файлу на

	локальний або віддалений репозиторій
<b>deploy</b>	Встановлення фінального виконуваного JAR/WAR файлу на віддалений репозиторій

Повний список фаз та життєвих циклів можна переглянути за посиланням: <http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html>.

Перед кожною фазою і після кожної фази існує можливість реєстрації цілей. Під час збирання проекту, процес проходить певну послідовність фаз і виконує цілі, які зареєстровані на кожній фазі. Наступні три стандартні життєві цикли в Maven існують за замовчуванням:

- clean
- default (or build)
- site

Ціль – це конкретна задача, пов’язана з процесом збирання проекту. Ціль може бути зареєстрована до певної фази (або навіть до декількох фаз). Однак, реєстрація цілі до певної фази не є обов’язковою. Ціль, яка не зв’язана з жодною фазою, може бути виконана поза життєвим циклом шляхом прямого виклику за назвою цілі. Порядок виконання залежить від порядку, в якому викликаються цілі та фази.

Далі розглянемо типові теги, які використовуються для конфігурації Maven.

Версія Java встановлюється за допомогою тегів properties:

```
<properties>
  <maven.compiler.source>1.7</maven.compiler.source>
  <maven.compiler.target>1.7</maven.compiler.target>
</properties>
```

Для того, щоб мати можливість вказати версію Java вище 7-ї, необхідно сконфігурувати версію плагіна maven-compiler-plugin:

```
<build>
  <pluginManagement>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
```

```

        <artifactId>maven-compiler-
plugin</artifactId>
        <version>3.8.1</version>
    </plugin>
</plugins>
</pluginManagement>
</build>
<properties>

<maven.compiler.release>11</maven.compiler.release>
</properties>

```

Залежності додаються до проекту за допомогою тегів `dependencies`:

```

<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>test</scope>
    </dependency>
</dependencies>

```

Додавання додаткових репозиторіїв, де Maven буде шукати залежності, бібліотеки та плагіни:

```

<repositories>
    <repository>
        <id>domain.name</id>

<url>http://some.url.com/maven2/repository</url>
    </repository>
</repositories>

```

Maven виконує пошук залежностей у наступній послідовності:

- пошук в локальному сховищі; якщо залежність не знайдено, то відбувається перехід до наступного кроку;
- пошук у центральній репозиторії (maven central); якщо залежність не знайдено та додатково вказані інші віддалені репозиторії – то відбувається перехід до останнього кроку, інакше – пошук завершується помилкою;
- пошук у віддалених репозиторіях.

### Плагіни

Майже всі команди Maven є плагінами, в тому числі і типові цілі, які реєструються до певних фаз життєвого циклу збирання проекту. Всі задачі виконуються за допомогою плагінів, навіть найтипівіші з них, такі як build, package і т.д.

Отже, плагіни використовуються для будь-яких задач в рамках процесу збирання проектів за допомогою Maven:

- компіляція коду проекту
- запуск юніт-тестів
- створення документації
- створення JAR-файлів
- створення WAR-файлів
- генерація звітів

Кожен плагін зазвичай надає набір цілей, кожен з яких можна запустити, виконавши наступну команду:

```
$ mvn <назва_плагіну>:<назва_цілі>
```

Наприклад:

```
$ mvn compiler:compile
```

Список типових плагінів:

- clean
- compiler
- jar
- war
- javadoc

### 3.3 Порядок виконання роботи

1. Встановити maven:

```
$ sudo apt-get install maven
```

Перевірити налаштування maven:

```
$ mvn -version
```

2. Збірка веб-застосування.

Сконфігурувати файл зборки веб-застосування (існуючого або створити новий проект – архетип maven-archetype-webapp (A simple Java web application) з включенням етапу автоматичного розгортання застосування на веб-сервері.

Для запуску створеного веб-застосування необхідно встановити tomcat:

```
$> aptitude install tomcat6 tomcat6-admin  
tomcat6-docs tomcat6-examples tomcat6-user
```

У файлі /etc/tomcat6/tomcat-users.xml задаємо пароль для користувача admin.

Після цього веб-застосування можна розгорнути на веб-сервері вручну. Для автоматизації розгортання, необхідно налаштувати плагін tomcat-maven-plugin:

```
<plugin>  
  <groupId>org.codehaus.mojo</groupId>  
  <artifactId>tomcat-maven-  
plugin</artifactId>  
  <configuration>  
  
    <url>http://localhost:8080/manager/html</url>  
    <server>mainId</server>  
  </configuration>  
</plugin>
```

Вказуємо адресу веб-сервера, де необхідно розгорнути веб-застосування.

Дані для авторизації на вказаному сервері необхідно записати у файлі /etc/maven2/settins.xml у розділі < servers >:

```
<server>  
  <id>mainId</id>  
  <username>admin</username>  
  <password>пароль</password>  
</server>
```

Для автоматичного розгортання необхідно виконати mvn tomcat:deploy, після чого веб-застосування буде доступно за відповідною адресою: [http://localhost:8080/назва\\_проекту](http://localhost:8080/назва_проекту).

3. Створення executable JAR-файл з використанням власних бібліотек-залежностей.

- 1) Зібрати проект з моделювання (або BuldoExample). Для цього потрібно зібрати та інсталювати в локальний репозиторій бібліотеку Simulation.
- 2) Для власного проекту встановити залежність від артефакту Simulation.
- 3) Для конфігурування файлу Manifest вказуємо налаштування пагіну maven-jar-plugin (визначаємо головний клас та задаємо динамічне формування змінної classpath)

```
<plugin>
```

```
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-jar-plugin</artifactId>
<configuration>
    <archive>
        <manifest>

<addClasspath>true</addClasspath>
        <classpathPrefix>lib/
</classpathPrefix>
        <mainClass>buldo0.Buldo0App
</mainClass>
    </manifest>
    </archive>
</configuration>
</plugin>
```

Для автоматичного копіювання всієї бібліотек-залежностей проекту з локального репозиторію артефактів у заданий каталог (у даному випадку /lib) додаємо відповідні налаштування плагіну maven-dependency-plugin:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-dependency-
plugin</artifactId>
    <executions>
        <execution>
            <id>copy-dependencies</id>
            <phase>package</phase>
            <goals><goal>copy-
dependencies</goal></goals>
            <configuration>
                <outputDirectory>target/lib
</outputDirectory>
            </configuration>
        </execution>
    </executions>
</plugin>
```

4) Збираємо та запускаємо на виконання застосування:

```
mvn package
java -jar BuldoExample.jar
```



Замість ручного конфігурування для створення executable JAR-файл можна використовувати спеціальні плагіни `maven-assembly-plugin` / `maven-shade-plugin`.

Приклад використання плагіну `maven-assembly-plugin`:

```
<plugin>
  <artifactId>maven-assembly-
plugin</artifactId>
  <version>3.1.1</version>
  <configuration>
    <archive>
      <manifest>

<mainClass>{MAIN_CLASS}</mainClass>
      </manifest>
    </archive>
    <descriptorRefs>
      <descriptorRef>jar-with-
dependencies</descriptorRef>
    </descriptorRefs>
  </configuration>
</plugin>
```

Створення єдиного JAR-файлу з усіма залежностями в такому випадку:

```
$ mvn clean compile assembly:single
```

### **3.4 Завдання для самостійної роботи**

1) Визначення області дії залежності (scope), транзитивних залежностей.

2) Додати документацію до проекту: описати у форматі Javadoc (<https://docs.oracle.com/javase/9/javadoc/javadoc.htm>) всі класи, публічні методи в проекті та їх аргументи. Згенерувати документацію у форматі HTML за допомогою Maven.

Для генерації документації використовується плагін `javadoc` (деталі конфігурації тут: <https://maven.apache.org/plugins/maven-javadoc-plugin/>):

```
<plugin>
```

```
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-javadoc-
plugin</artifactId>
<version>3.1.1</version>
</plugin>
```

Згенерувати документацію можна за допомогою однієї з наступних команд (залежить від конфігурації: перша команда, якщо генерація звітів була зконфігурована для життєвого циклу build, друга команда – для життєвого циклу site):

```
$ mvn javadoc:javadoc
$ mvn site
```

Результати у браузері:



Рисунок 2.1 – Результат генерації документації до проекту

3) Розгорнути власний maven-репозиторій артефактів (на базі Apache Archiva <https://archiva.apache.org/docs/2.2.5/adminguide/index.html> або ін. інструменту).

Налаштувати локальний репозиторій артефактів maven. Глобальні налаштування репозиторію в файлі /путь/к/maven2/settings.xml або локальні в pom.xml-файлі певного проекту. Наприклад, в розділі <profiles></profiles> додано репозиторій:

```
<profile>
<id>stuRepo</id>
<repositories>
<repository>
  <id>main</id>
  <name>StuRepository</name>
```

```

        <url>http://devel.stu.cn.ua:8081/artifactor
y/repo/</url>
        <layout>default</layout>
    </repository>
    <repository>
        <id>snapshots</id>
        <url>http://devel.stu.cn.ua:8081/artifactor
y/repo/</url>
        <releases>
            <enabled>false</enabled>
        </releases>
    </repository>
</repositories>
<pluginRepositories>
<pluginRepository>
    <id>central</id>
    <url>http://devel.stu.cn.ua:8081/artifactor
y/plugins-releases/</url>
    <snapshots>
        <enabled>false</enabled>
    </snapshots>
</pluginRepository>
<pluginRepository>
    <id>snapshots</id>
    <url>http://devel.stu.cn.ua:8081/artifactor
y/plugins-snapshots/</url>
    <releases>
        <enabled>false</enabled>
    </releases>
</pluginRepository>
</pluginRepositories>
</profile>
Активізуємо профайл:
<activeProfiles>
    <activeProfile>stuRepo</activeProfile>
</activeProfiles>

```

### 3.5 Структура звіту

- зміст файлу глобальних налаштувань maven;
- зміст файлів pom.xml для кожного модуля web-застосування;
- результати виконання зборки проектів.

## **4 Лабораторна робота №4**

### **Тестування продуктивності веб-застосування**

#### **4.1 Мета роботи**

Дослідження принципів тестування продуктивності програмного забезпечення. Вивчення особливостей та практичне застосування кросплатформного інструменту тестування навантаження Apache Jmeter.

#### **4.2 Теоретичні відомості**

Тестування навантаження застосовується для аналізу параметрів продуктивності інформаційних систем для різних рівнів навантаження. Тестування навантаження – це автоматизоване тестування, що імітує роботу певної кількості користувачів на якомусь загальному (розподіленому) ресурсі. Моделювання навантаження відбувається за допомогою спеціальних продуктів та інструментів.

Для моделювання навантаження використовується поняття віртуального користувача. Керуючи кількістю віртуальних користувачів, тестувальник керує навантаженням на систему. Віртуальний користувач виконує типові операції в системі шляхом відтворення трафіку, який відправляється клієнтським додатком на сервер, в саме виконує скрипти, які посилають на сервер пакети у форматі протоколу, що використовується на стороні серверу застосування, наприклад, http, odbc, NCA та ін.

Основні показники продуктивності інформаційної системи, що вимірюються в ході тестування навантаження:

- Час відгуку (час виконання операції);
- Число операцій, що виконуються в одиницю часу (TPS - transactions per second).

Одним з термінів тестування навантаження є "крива деградації" – графік, що показує залежність продуктивності системи (наприклад, в одиницях часу відгуку) від робочого навантаження (наприклад, від числа віртуальних користувачів).

Основним результатом тестування навантаження є вимірювання продуктивності інформаційної системи, які можуть бути використані для локалізації вузьких місць і оптимізації продуктивності.

Тестування навантаження тісно пов'язане з тестуванням можливості обробки великих обсягів даних. Зростання бази даних сайту через кілька місяців, тим більше років, зазвичай виявляється значним, що потребує оцінки відповідних параметрів продуктивності системи.

Тестування масштабованості також пов'язане з тестуванням продуктивності та передбачає оцінку зростання продуктивності застосування при додаванні апаратних ресурсів (пам'ять, процесор і т.д.).

При тестуванні навантаження для кожної сторінки (для кожного сервісу, який надає додаток) потрібно визначити нижню межу часу відгуку. Необхідно враховувати розподіл параметру часу відгуку відповідно доби, а також частоти використання сервісів.

Критеріями успішності тестування навантаження є нефункціональні вимоги до продуктивності системи, які формулюються та документуються до початку розробки системи.

Одним із оптимальних підходів у використанні навантажувального тестування для оцінки продуктивності системи є тестування на ранніх стадіях розробки. Тестування застосування на ранніх стадіях готовності з метою визначення ефективності архітектурного рішення називається 'Proof-of-Concept' тестуванням.

### 4.3 Порядок виконання роботи

1. Розгортання веб-застосування на веб-сервері tomcat. Можливим є тестування веб-застосування, що розроблялося в межах курсового проекту з дисципліни «Технології прикладного програмування» за минулий семестр.

Для розгортання веб-застосування встановити tomcat:

```
aptitude install tomcat6 tomcat6-admin tomcat6-docs  
tomcat6-examples tomcat6-user
```

В файлі /etc/tomcat6/tomcat-users.xml створюємо роль для менеджера та користувача, у якого та сама роль:

```
<role rolename="manager"/>  
<user username="manager" password="qwerty"  
roles="manager"/>
```

Перезапускаємо веб-сервер tomcat:

```
sudo /etc/init.d/tomcat6 restart
```

Для запуску менеджера веб-застосувань tomcat необхідно за посиланням <http://127.0.1.1:8080/manager/html> авторизуватись під відповідним користувачем, що був створений раніше (user: «manager», pass : «qwerty»). Після чого з використанням графічного інтерфейсу менеджера веб-застосувань tomcat (рисунок 4.1) можна розгорнути відповідний war-файл застосування на сервері.

The screenshot shows the Tomcat Manager web interface. The 'Deploy' tab is active, displaying fields for 'Context Path (required)', 'XML Configuration file URL', and 'WAR or Directory URL'. Below these fields is a 'Deploy' button. The 'WAR file to deploy' section shows a 'Select WAR file to upload' button and a 'Deploy' button. The 'Diagnostics' section has a 'Find leaks' button. The 'Server Information' section displays a table with server details.

Tomcat Version	JVM Version	JVM Vendor	OS Name	OS Version	OS Architecture
Apache Tomcat/6.0.28	1.6.0_20-b20	Sun Microsystems Inc.	Linux	2.6.35-22-generic	i386

Copyright © 1999-2010, Apache Software Foundation

Рисунок 4.1 – Розгортання веб-застосування

## 1. Встановлення Apache Jmeter.

Додаткову інформацію щодо встановлення Apache Jmeter див.  
<https://jmeter.apache.org/usermanual/get-started.html>.

Завантажити пакет `jakarta-jmeter-2.4.tar.gz` можна з офіційного сайту [http://jakarta.apache.org/site/downloads/downloads\\_jmeter.cgi](http://jakarta.apache.org/site/downloads/downloads_jmeter.cgi). Розпакувати завантажений пакет та запустити Apache Jmeter за допомогою скрипта з каталогу `bin`:

```
$ ~/jakarta-jmeter-2.4$ ./bin/jmeter.sh
```

Інтерфейс Jmeter наведено на рисунку 4.2.

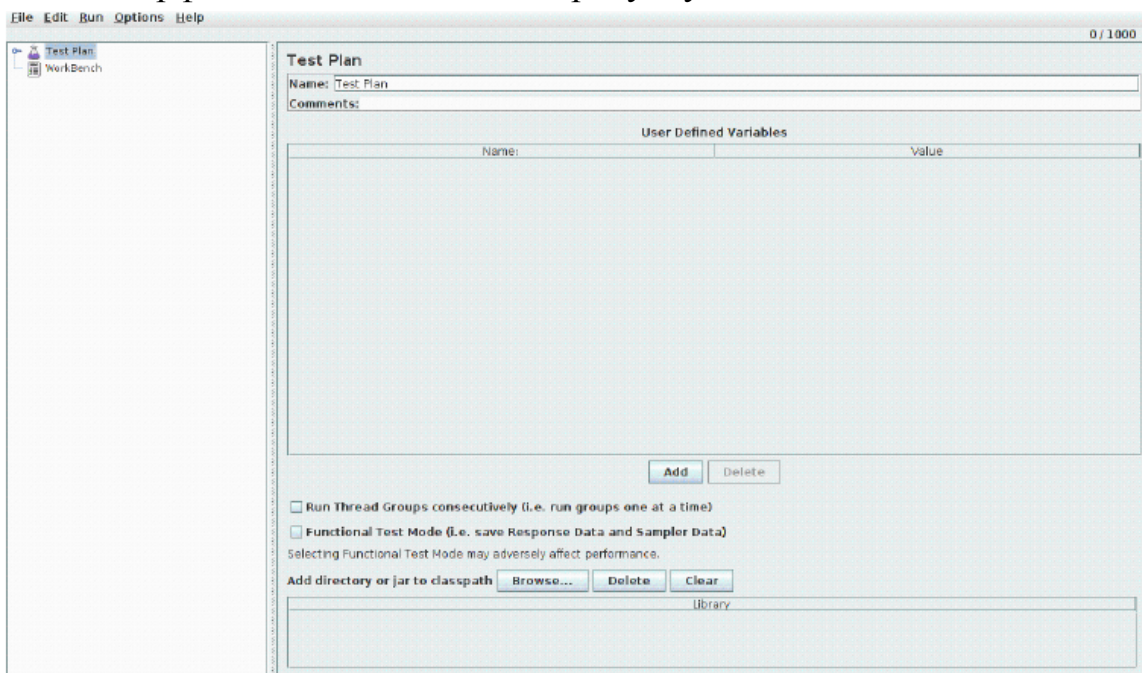


Рисунок 4.2 — Графічний інтерфейс Apache Jmeter

## 2. Тестування веб-застосування.

Додати в тестовий план елемент Thread Group. В Thread Group визначити кількість користувачів, які надсилають запити до відповідного веб-застосування, за який період часу вони надсилають запити та кількість запитів, які надсилає кожен користувач (рисунок 4.3). Для того, щоб додати Thread Group необхідно на Test Plan клікнути правою кнопкою миші та обрати пункт Add --> ThreadGroup.

**Thread Group**

Name: Thread Group

Comments:

Action to be taken after a Sampler error

☒ Continue ☐ Stop Thread ☐ Stop Test ☐ Stop Test Now

Thread Properties

Number of Threads (users): 1

Ramp-Up Period (in seconds): 1

Loop Count: ☐ Forever 1

☐ Scheduler

Рисунок 4.3 — Значення за замовчуванням параметрів Thread Group

У полі Number of threads вказуємо кількість користувачів, які будуть звертатися на сервер. У наступному полі вказуємо затримки між початком кожного потоку. Наприклад, якщо ввести Ramp-Up period 5 секунд, JMeter завершить виконання запитів від усіх користувачів за час 5 секунд. Так, якщо ми маємо 5 користувачів і 5 секунд період, то затримка між початком нового потоку буде 1 секунда ( $5 \text{ користувачів} / 5 \text{ секунд} = 1 \text{ користувач в секунду}$ ). Якщо встановити значення 0, то JMeter відразу ж запустить усі потоки. Поле Loop Count вказує, скільки разів повторюватиметься тест.

Після визначення параметрів необхідно вказати запити, які виконуватимуть віртуальні користувачі. У Test Plan додаємо HTTP requests зі значеннями за замовчуванням Add->ConfigElement->HTTPRequestDefaults (рисунок 4.4).

**HTTP Request Defaults**

Name: HTTP Request Defaults

Comments:

**Web Server**

Server Name or IP: volosek-lisap.com.ua Port Number: 80

**Timeouts (milliseconds)**

Connect: Response:

**HTTP Request**

Protocol (default http): Content encoding:

Path: index.php

**Send Parameters With the Request:**

Name:	Value	Encode?	Include Equals?
-------	-------	---------	-----------------

Add Delete

**Proxy Server**

Server Name or IP: Port Number: Username: Password:

☐ Retrieve All Embedded Resources from HTML Files

Рисунок 4.4 — HTTP Request Defaults

Далі додаємо HTTP Request (Add --> Sampler --> HTTP Request) і в полі HTTP Request Path вказуємо шлях до сторінки, на яку буде надіслано запит типу get. У полі Server name вказувати адресу сервера не потрібно, тому що сервер вказано на сторінці за замовчуванням (рисунки 4.5).



**HTTP Request**

Name: Home page

Comments:

**Web Server**

Server Name or IP: Port Number: Connect: Response:

**HTTP Request**

Protocol (default http): Method: GET Content encoding:

Path: /index.php

☐ Redirect Automatically ☒ Follow Redirects ☒ Use KeepAlive ☐ Use multipart/form-data for HTTP POST

**Send Parameters With the Request:**

Name:	Value	Encode?	Include Equals?

Add Delete

**Send Files With the Request:**

File Path:	Parameter Name:	MIME Type:

Add Browse... Delete

**Proxy Server**

Server Name or IP: Port Number: Username: Password:

**Optional Tasks**

☐ Retrieve All Embedded Resources from HTML Files ☐ Use as Monitor ☐ Save response as MD5 hash?

Embedded URLs must match:

Рисунок 4.5 — HTTP Request головної сторінки

Аналогічно додаємо інший HTTP Request, в полі Path вказуємо шлях до іншої сторінки.

Останнім кроком, який необхідно зробити, щоб розпочати тестування, є додавання слухачів (Listener) для обробки інформації від HTTP Request. Для цього додамо Graph Results (Add --> Listener --> Graph Results) та View Results Tree (Add --> Listener --> View Results Tree) (рисунок 4.6 та 4.7).

Після завершення тестування можна переглянути результат. На малюнку 4.8 наведено приклад тесту, на якому видно, що система не витримала навантаження, яке ми відправили. Можна детально подивитись вміст кожного пакета, а також приклад html сторінки, яку отримали у відповідь на запит та визначити причину негативної відповіді.

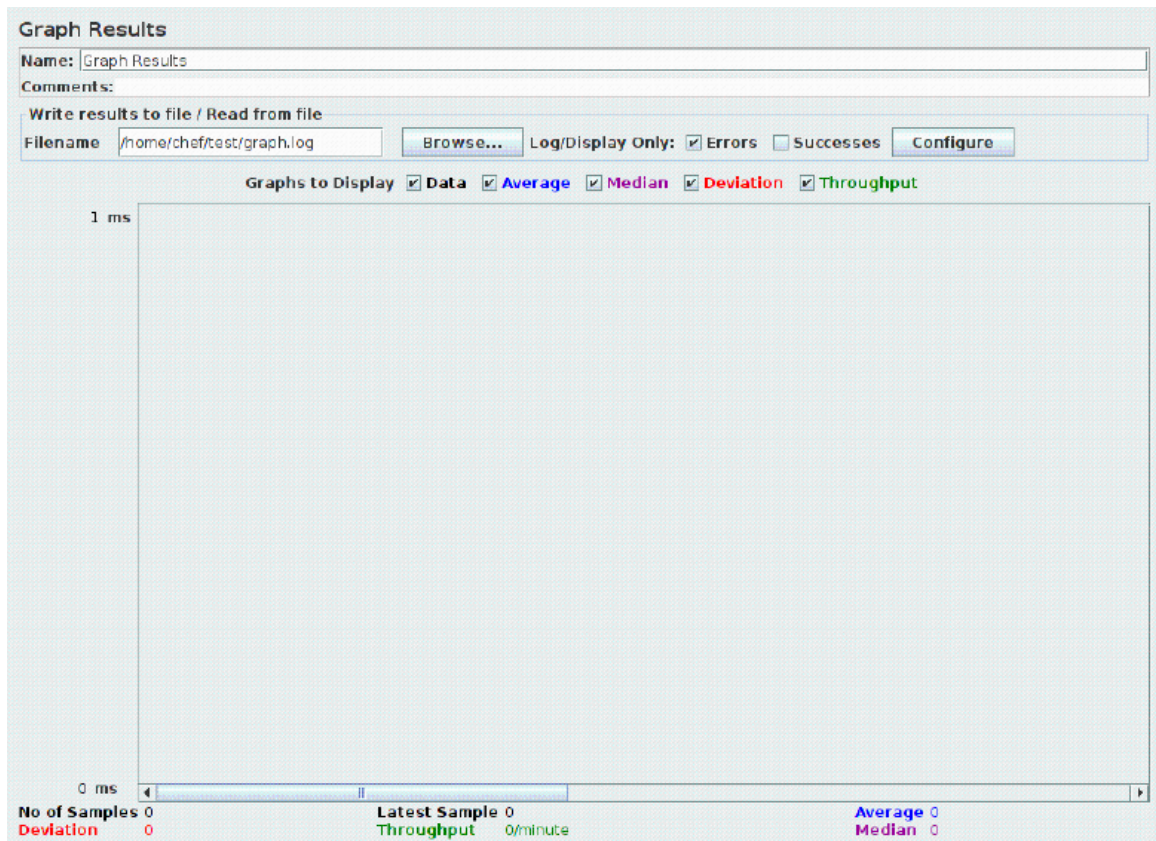


Рисунок 4.6 — Graph Results

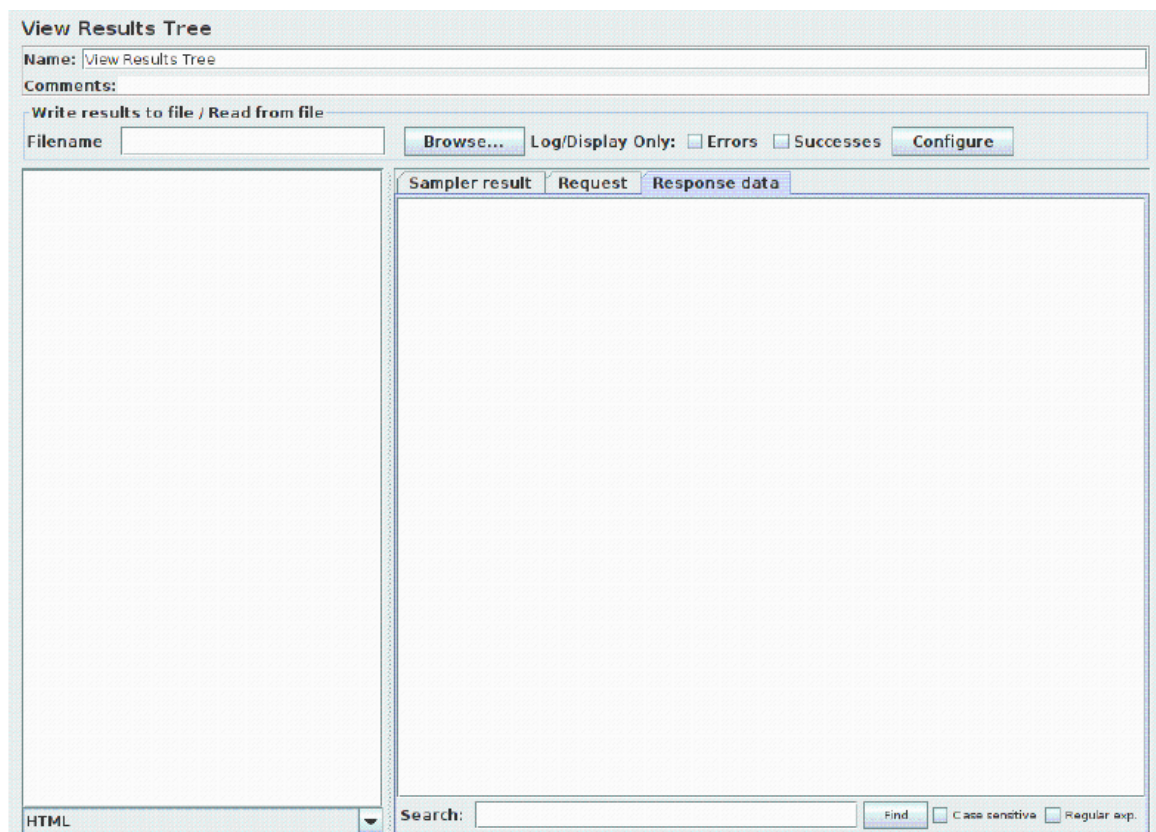


Рисунок 4.7 — View Results Tree

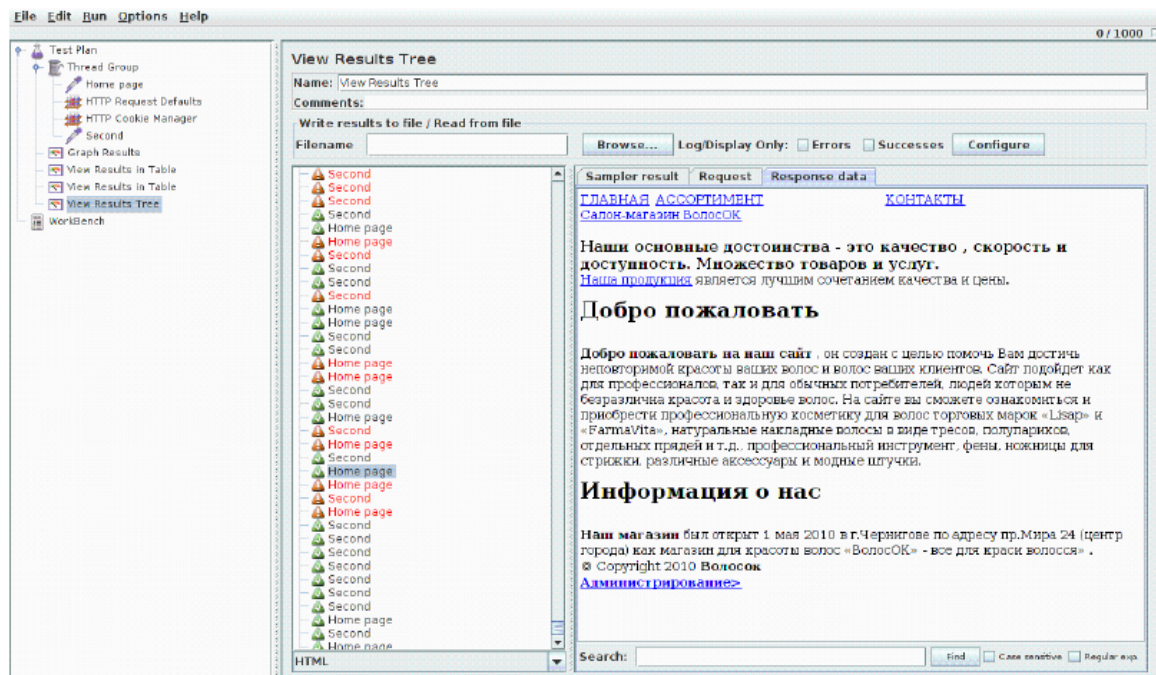


Рисунок 4.8 — Результати тестування

Бажано, щоб додаток було розгорнуто на іншому сервері, оскільки JMeter вимогливий до ресурсів та результати тестування можуть бути некоректними.

#### 4.4 Завдання для самостійної роботи

3. Вивчити можливості організації database testing (тестування роботи з БД) з використанням JMeter.
4. Вивчити аспекти інтеграції Apache JMeter з JUnit.

#### 4.5 Структура звіту

- результати тестування у вигляді графіків;
- результати тестування у вигляді дерева;
- проаналізувати результати тестування. Встановити причини, через які система не витримала навантаження.

## **5 Лабораторна робота №5. Застосування серверів неперервної інтеграції (Continuous Integration/ Continuous Delivery Servers)**

### **5.1 Мета роботи**

Вивчити особливості застосування інструментів неперервної інтеграції. Отримати навички встановлення, налаштування та адміністрування системи неперервної інтеграції Jenkins.

### **5.2 Теоретичні відомості**

Неперервна інтеграція (Continuous Integration, Continuous Delivery – CI/ CD) полягає в частих періодичних збірках проєкту в процесі розробки для швидкого виявлення та вирішення проблем інтеграції. Перехід до неперервної інтеграції дозволяє знизити трудомісткість інтеграції, що при виконанні на завершальних етапах є зазвичай складною та витратною.

Принцип використання систем неперервної інтеграції.

На окремому сервері розгортається служба, що виконує такі задачі:

- отримання вихідного коду з репозиторію;
- збірка проєкту;
- запуск та виконання автоматизованих тестів;
- розгортання (встановлення) проєкту;
- формування та відправлення звітів про результати зборки проєкту.

Збірка проєкту може виконуватися:

- за зовнішнім запитом;
- за розкладом;
- при внесенні змін до репозиторію;
- за іншими критеріями.

#### **Збірка за розкладом**

Оскільки для великих проєктів процес зборки є зазвичай довгим та витратним, то використовують практику автоматизованих нічних зборок. Вводиться система нумерації зборок, кожна збірка нумерується натуральним числом, що збільшується з кожною новою збіркою. Вихідні тексти та інші вихідні дані під час завантаження їх із репозиторію системи контролю версій позначаються номером. Завдяки цьому, певна збірка може бути точно відтворена в майбутньому — достатньо завантажити вихідні дані відповідно потрібного номеру та запустити процес знову. Це дозволяє повторно створювати попередні версії програмного проєкту з внесенням потрібних змін.

Відомі такі засоби неперервної інтеграції (CI/CD Tools):



Jenkins (<https://www.jenkins.io/>);  
TeamCity (<https://www.jetbrains.com/teamcity/>);  
Bamboo (<https://www.atlassian.com/software/bamboo>);  
Travis CI (<https://www.travis-ci.com/>).

### 5.3 Порядок виконання роботи

#### 1. Встановити Jenkins

<https://www.jenkins.io/doc/book/installing/linux/#debianubuntu>

2. Налаштувати автоматизовану збірку для проекту з лабораторної роботи №3, а саме: 1) налаштувати шлях до репозиторію проекту (локальний, github або ін.); 2) налаштувати систему збірки та цілі збірки проекту (використовувати систему збірки л. р. №3) (рисунок 5.1).

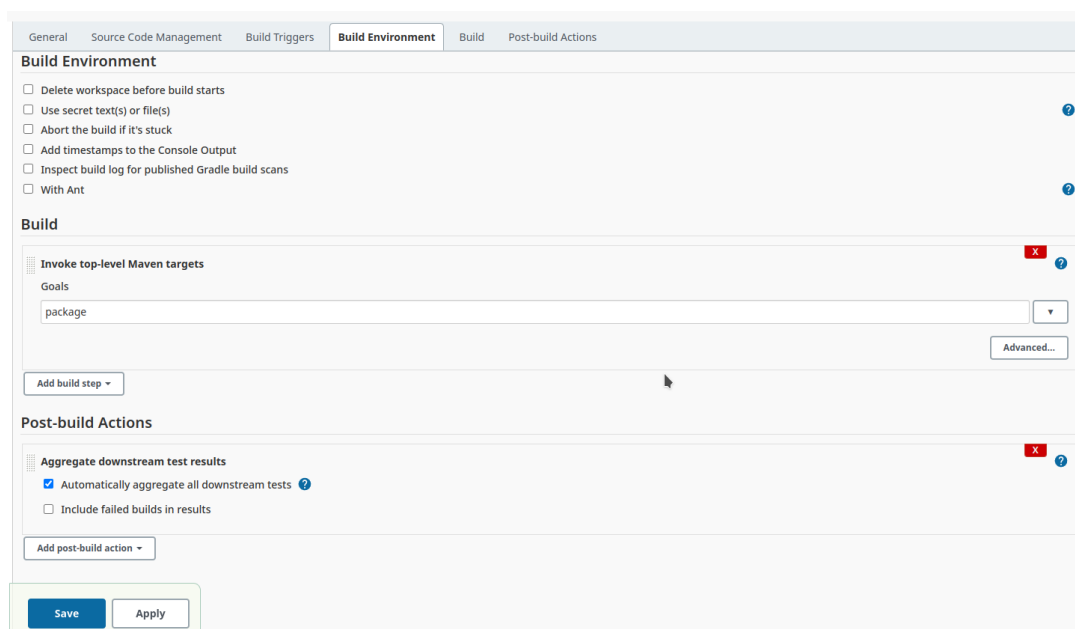


Рисунок 5.1 – Налаштування системи збірки та цілей збірки проекту

#### 3. Налаштувати розклад автоматизованої збірки проекту.

Періодичність збірки може бути налаштована з використанням регулярних виразів (рис. 5.2).

## Методичні вказівки до виконання лабораторних робіт

General **Build Triggers** Advanced Project Options Pipeline

- \* specifies all valid values
- M-N specifies a range of values
- M-N/X or \*/X steps by intervals of X through the specified range or whole valid range
- A, B, ..., Z enumerates multiple values

To allow periodically scheduled tasks to produce even load on the system, the symbol H (for "hash") should be used wherever possible. For example, using `0 0 * * *` for a dozen daily jobs will cause a large spike at midnight. In contrast, using `H H * * *` would still execute each job once a day, but not all at the same time, better using limited resources.

The H symbol can be used with a range. For example, `H H(0-7) * * *` means some time between 12:00 AM (midnight) to 7:59 AM. You can also use step intervals with H, with or without ranges.

The H symbol can be thought of as a random value over a range, but it actually is a hash of the job name, not a random function, so that the value remains stable for any given project.

Beware that for the day of month field, short cycles such as `*/3` or `H/3` will not work consistently near the end of most months, due to variable month lengths. For example, `*/3` will run on the 1st, 4th, ...31st days of a long month, then again the next day of the next month. Hashes are always chosen in the 1-28 range, so `H/3` will produce a gap between runs of between 3 and 6 days at the end of a month. (Longer cycles will also have inconsistent lengths but the effect may be relatively less noticeable.)

Empty lines and lines that start with # will be ignored as comments.

In addition, @yearly, @annually, @monthly, @weekly, @daily, @midnight, and @hourly are supported as convenient aliases. These use the hash system for automatic balancing. For example, @hourly is the same as `H * * * * *` and could mean at any time during the hour. @midnight actually means some time between 12:00 AM and 2:59 AM.

Examples:

```
# Every fifteen minutes (perhaps at :07, :22, :37, :52):
H/15 * * * *
# Every ten minutes in the first half of every hour (three times, perhaps at :04, :14, :24):
H(0-29)/10 * * * *
# Once every two hours at 45 minutes past the hour starting at 9:45 AM and finishing at 3:45 PM every weekday:
45 9-16/2 * * 1-5
# Once in every two hour slot between 8 AM and 4 PM every weekday (perhaps at 9:38 AM, 11:38 AM, 1:38 PM, 3:38 PM):
H H(8-15)/2 * * 1-5
# Once a day on the 1st and 15th of every month except December:
H H 1,15 1-11 *
```

**Time zone specification**

Periodic tasks are normally executed at the scheduled time in the time zone of the Jenkins master JVM (currently **Europe/Kiev**). This behavior can optionally be changed by specifying an alternative time zone in the first line of the field. Time zone specification starts with TZ=, followed by the ID of a time zone.

Complete example of a schedule with a time zone specification:

```
TZ=Europe/London
# This job needs to be run in the morning, London time
H * * * * *
```

Сохранить Применить

Рисунок 5.2 – Регулярні вирази для налаштування періодичності зборки

Також може бути налаштована автоматизована зборка при відправленні змін до репозиторію проекту (рисунок 5.3).

General **Build Triggers** Advanced Project Options Pipeline

**Build Triggers**

☐ Build after other projects are built

☒ Запускать периодически

Расписание

Нет запланированных запусков

☐ GitHub hook trigger for GITScm polling

When Jenkins receives a GitHub push hook, GitHub Plugin checks to see whether the hook came from a GitHub repository which matches the Git repository defined in SCM/Git section of this job. If they match and this option is enabled, GitHub Plugin triggers a one-time polling on GITScm. When GITScm polls GitHub, it finds that there is a change and initiates a build. The last sentence describes the behavior of Git plugin, thus the polling and initiating the build is not a part of GitHub plugin. (from [GitHub plugin](#))

☒ Опрашивать SCM об изменениях

Расписание

No schedules so will only run due to SCM changes if triggered by a post-commit hook

This field follows the syntax of cron (with minor differences). Specifically, each line consists of 5 fields separated by TAB or whitespace:

MINUTE	HOUR	DOM	MONTH	DOW
MINUTE	Minutes within the hour (0-59)			
HOUR	The hour of the day (0-23)			
DOM	The day of the month (1-31)			
MONTH	The month (1-12)			
DOW	The day of the week (0-7) where 0 and 7 are Sunday.			

field, the following operators are available. In the order of precedence,

Сохранить Применить

Рисунок 5.3 – Налаштування автоматизованої зборки при змінах в репозиторію проекту

4. Запустити збірку вручну, приклад журналу успішного виконання збірки на рисунку 5.4.



Рисунок 5.4 – Приклад журналу успішного виконання збірки

5. Ознайомитися з принципами декларативного визначення сценарію (pipeline) збірки проекту. Згенерувати декларативний опис pipeline.

## 5.4 Завдання для самостійної роботи

Додати етап тестування навантаження з використанням Apache JMeter (див. лабораторну роботу №4) в сценарій інтеграції проєкту, створеного в п.2. <https://www.jenkins.io/doc/book/using/using-jmeter-with-jenkins/>.

## 5.5 Структура звіту

- процес та результати встановлення та налаштування системи постійної інтеграції Jenkins;
- налаштування правил збірки проєкту;
- результат автоматизованої збірки проєкту за розкладом.

## **6 Лабораторна робота № 6**

### **Системи керування версіями баз даних**

#### **6.1 Мета роботи**

Пригадання принципів роботи з базами даних. Ознайомлення з поняттям «система керування версіями баз даних». Отримання практичних навичок відслідковувати зміни у базі даних за допомогою інструменту Liquibase і ресурсу Liquibase Hub.

#### **6.2 Теоретичні відомості**

##### **Системи керування версіями баз даних**

Як відомо, для відслідковування змін у файлах існують системи контролю версій. Крім того, за допомогою систем контролю версіями можна не тільки переглянути, коли і ким було змінено документ, а і повернутися до змін, унесених на певному етапі розробки файлу або проєкту.

Для баз даних також існує подібне програмне забезпечення, яке називають системами керування версіями баз даних. Узагалі, опис процесу зміни бази даних із однієї версії на іншу також відомий як міграція бази даних (**Database Migration**). Сам же процес може полягати як у оновленні на новішу версію, так і повернення на одну з попередніх.

Застосування версіонування баз даних дозволяє вирішити такі задачі:

- ініціалізація бази даних;
- заповнення бази даних інформацією;
- перетворення уже наявної інформації в базі даних;
- модифікація схеми (структури) бази даних.

Залежно від мови програмування та/або використовуваного фреймворку існують різні методи ініціалізації та міграції баз даних. Так, наприклад, для фреймворку Spring Boot притаманні 4 такі методи:

- технології JPA/Hibernate;
- ініціалізація Spring Boot;
- ручне виконання запитів для створення і модифікації бази даних;
- використання високорівневих інструментів (Flyway чи Liquibase).

У цій лабораторній роботі буде розглянутий кросплатформенний інструмент для керування версіями баз даних - бібліотека Liquibase, розроблена у 2006 році на мові Java.

#### **Ключові поняття Liquibase**



Усі зміни бази даних зберігаються у вигляді текстових файлів (допустимі розширення: XML, YAML, JSON та SQL) та ідентифікуються атрибутами «id» і «author». Такі файли зазвичай називаються **changelog**. Список усіх застосованих змін зберігається в кожній базі даних, яка переглядається під час усіх оновлень бази даних, щоб визначити, які нові зміни потрібно застосувати. Як результат, номер версії бази даних відсутній, проте такий підхід дає змогу працювати у середовищах з кількома розробниками та вітками коду. Liquibase автоматично створює відповідні компоненти **DatabaseChangeLog** і **DatabaseChangeLogLock** при першому виконанні файлу змін **changelog**.

Кожна модифікація бази даних має назву **changeset**.

Якщо усі зміни зберігаються у файлі з розширенням XML, то він зазвичай називається **changelog.xml**, містить тег **databaseChangelog** та вкладені у нього зміни, позначені тегами **changeSet**. Конкретне оновлення **changeSet**, у свою чергу, складається з атрибутів **author** та **id**, які позначають ім'я автора зміни й її ідентифікатора.

Якщо історія змін бази даних зберігаються у файлі з іншим розширенням, то вміст **changelog**-файлу буде містити свої особливості виходячи із особливостей представлення інформації у заданому форматі.

При необхідності, можна повернутися до певної зміни, виконавши повернення до потрібної версії – операцію **rollback**.

Слід зауважити, що перед тим, як поєднувати **changelog** та досліджувану базу даних, необхідно створити файл налаштувань **liquibase.properties**, який буде зберігати параметри, потрібні для з'єднання з базою даних (**url**, розташування драйвера тощо).

### Інсталяція та налаштування Liquibase

Для демонстрації роботи Liquibase буде застосовуватися операційна система Ubuntu версії 20.04.

Перш за все нам слід запустити термінал і прописати у ньому наступну команду:

```
$ sudo snap install liquibase
```

Протягом наступних кількох секунд після виконання відбуватиметься інсталяція системи керування версіями баз даних Liquibase.

Проте, можна завантажити це програмне забезпечення і з офіційного сайту (версія для Linux: [https://www.liquibase.com/download?\\_ga=2.115268615.1414755213.1675348269-1096829722.1675240386](https://www.liquibase.com/download?_ga=2.115268615.1414755213.1675348269-1096829722.1675240386)).

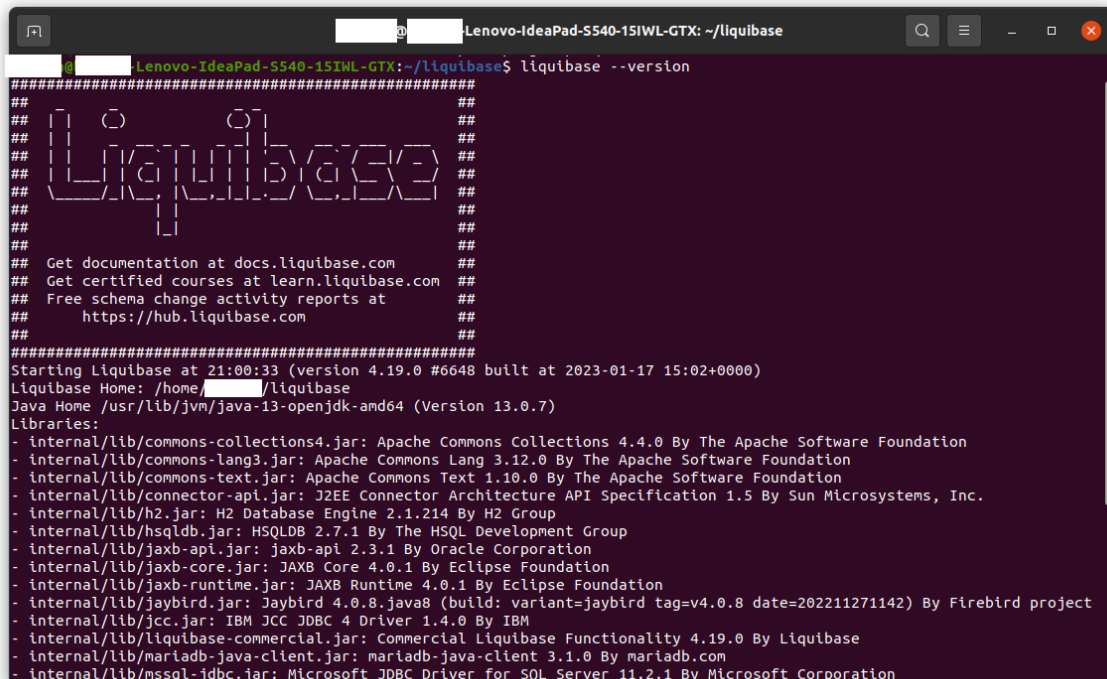
Після цього завантажений архів потрібно розпакувати у будь-якому каталозі файлової системи.

Слід зауважити, що для роботи Liquibase також потрібно встановити Java.

Якщо ж усе було виконано правильно, то команда визначення версії цього програмного забезпечення покаже відповідний результат (див. рисунок 5.1).

Команда для визначення версії liquibase:

```
$ liquibase --version
```



```
Lenovo-IdeaPad-S540-15IWL-GTX: ~/liquibase
Lenovo-IdeaPad-S540-15IWL-GTX: ~/liquibase$ liquibase --version
#####
##                               ##
##  L I Q U I B A S E           ##
##  [ L I N E ]                 ##
##                               ##
##                               ##
##  Get documentation at docs.liquibase.com  ##
##  Get certified courses at learn.liquibase.com  ##
##  Free schema change activity reports at  ##
##    https://hub.liquibase.com              ##
##                               ##
#####
Starting Liquibase at 21:00:33 (version 4.19.0 #6648 built at 2023-01-17 15:02+0000)
Liquibase Home: /home/~/liquibase
Java Home /usr/lib/jvm/java-13-openjdk-amd64 (Version 13.0.7)
Libraries:
- internal/lib/commons-collections4.jar: Apache Commons Collections 4.4.0 By The Apache Software Foundation
- internal/lib/commons-lang3.jar: Apache Commons Lang 3.12.0 By The Apache Software Foundation
- internal/lib/commons-text.jar: Apache Commons Text 1.10.0 By The Apache Software Foundation
- internal/lib/connector-apl.jar: J2EE Connector Architecture API Specification 1.5 By Sun Microsystems, Inc.
- internal/lib/h2.jar: H2 Database Engine 2.1.214 By H2 Group
- internal/lib/hsqldb.jar: HSQLDB 2.7.1 By The HSQL Development Group
- internal/lib/jaxb-api.jar: jaxb-api 2.3.1 By Oracle Corporation
- internal/lib/jaxb-core.jar: JAXB Core 4.0.1 By Eclipse Foundation
- internal/lib/jaxb-runtime.jar: JAXB Runtime 4.0.1 By Eclipse Foundation
- internal/lib/jaybird.jar: Jaybird 4.0.8.java8 (build: variant=jaybird tag=v4.0.8 date=202211271142) By Firebird project
- internal/lib/jcc.jar: IBM JCC JDBC 4 Driver 1.4.0 By IBM
- internal/lib/liquibase-commercial.jar: Commercial Liquibase Functionality 4.19.0 By Liquibase
- internal/lib/mariadb-java-client.jar: mariadb-java-client 3.1.0 By mariadb.com
- internal/lib/mssql-jdbc.jar: Microsoft JDBC Driver for SQL Server 11.2.1 By Microsoft Corporation
```

Рисунок 5.1 – Визначення версії Liquibase

Після цього слід визначитися із базою даних, яку ми будемо відслідковувати. Визначившись, треба завантажити необхідні файли для виикористовуваної бази даних за посиланням <https://docs.liquibase.com/start/install/tutorials/home.html>, вибравши потрібний тип бази даних. Для MongoDB, наприклад, таких файлів 2. Далі ці файли треба розмістити у каталозі liquibase/lib. Варто зауважити, що у каталозі liquibase/internal/lib уже наявні драйвери до деяких баз даних. Однак, якщо завантажувється драйвер новішої версії, то відповідний драйвер старішої версії необхідно вилучити з вищезгаданого каталогу.

Наступним кроком буде налаштування файлу liquibase.properties, який розташовується у каталозі програми Liquibase. Для бази даних MongoDB файл може мати такий вміст:

```
changelogFile: changelog.xml
driver: oracle.jdbc.OracleDriver
url: mongodb://localhost:27017/databaseName
classpath:../path/to/file/ojdbc6-11.2.0.3.0.jar
```

- 1) **changeLogFile** використовується для визначення файлу, що буде фіксувати зміни у досліджуваній базі даних.
- 2) **driver** описує драйвер, який застосовується для взаємодії Liquibase та бази даних.
- 3) **url** — це адреса бази даних у мережі. Так, у попередньому прикладі база даних розташовується на цьому ж хості (localhost) порту 27017, а база даних має назву databaseName.
- 4) **classpath** — шлях до драйвера (конкретно до відповідного jar-файлу).

Швидко створити файл liquibase.properties (без заповнення) можна завдяки наступній команді:

```
$ liquibase init project
```

Після цього слід створити файл для відслідковування змін — changelog. Він може мати розширення SQL, JSON, XML або YAML. Нижче наведено приклад такого файлу з розширенням XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<databaseChangeLog
  xmlns=
    http://www.liquibase.org/xml/ns/dbchangelog
  xmlns:xsi=
    "http://www.w3.org/2001/XMLSchema-instance"
  xmlns:ext=
    "http://www.liquibase.org/xml/ns/dbchangelog-
ext"
  xmlns:pro=
    "http://www.liquibase.org/xml/ns/pro"
  xsi:schemaLocation=
    "http://www.liquibase.org/xml/ns/dbchangelog
    http://www.liquibase.org/xml/ns/dbchangelog/dbc
hangelog-latest.xsd
    http://www.liquibase.org/xml/ns/dbchangelog-ext
    http://www.liquibase.org/xml/ns/dbchangelog/dbc
hangelog-ext.xsd
    http://www.liquibase.org/xml/ns/pro
    http://www.liquibase.org/xml/ns/pro/liquibase-
pro-latest.xsd">
  </databaseChangeLog>
```

Зміни, внесені у базу даних, будуть позначені тегами `changeSet` у вищенаведеному файлі.

Для перевірки з'єднання між Liquibase та базою даних існує команда **status**:

```
$ liquibase status --username=user --password=test --changelog-file=<changelog.xml>
```

Параметри `--username` і `--password` у цьому випадку використовуються для підключення до бази даних з користувачем `user` і паролем `test` (ці параметри можна вказати заздалегідь у файлі `liquibase.properties`). Параметр `--changelog-file` зберігає назву файлу, який відслідковує базу даних.

Для синхронізації між базою даних і liquibase використовується команда **update**:

```
$ liquibase update --changelog-file=<changelog.xml>
```

Щоб повернутися до попередньої версії бази даних, потрібна команда **rollback**:

```
$ liquibase rollback --changelog-file=dbchangelog.xml --tag=version1
```

Параметр `--tag` містить значення тегу, який відповідає певній версії бази даних (додається однойменною командою).

Варто зазначити, що окрім самої бібліотеки Liquibase, існує ресурс Liquibase Hub (<https://hub.liquibase.com>) для зручного відслідковування та управління версіями баз даних.

### **6.3 Порядок виконання роботи**

1. Завантажити Liquibase для Linux.
2. Перевірити правильність інсталяції, навівши результат виконання команди `liquibase --version`.
3. Вибрати базу даних, для якої слід фіксувати зміни, і, при потребі, встановити додаткові файли.
4. Створити файл `liquibase.properties` і заповнити його відповідно до вибраної бази даних.
5. Створити файл з розширенням XML, метою якого буде фіксація змін у базі даних (`changelog.xml`).
6. Перевірити з'єднання між базою даних і Liquibase командою `status` і продемонструвати результат її роботи.
7. Оновити зміни командою `update`, навести результати виконання.

8. Внести зміни до бази даних, переглянути їх наявність у файлі changelog.xml. Навести додані рядки файлу в звіті.
9. Виконати повернення до однієї із попередніх версій бази даних командою rollback.

#### **6.4 Завдання для самостійної роботи**

Продемонструвати зміни у одній зі своїх баз даних за допомогою ресурсу Liquibase Hub.

#### **6.5 Структура звіту**

- скріншот результату завантаження Liquibase (виконання команди `liquibase --properties`);
- назва і опис вмісту вибраної бази даних;
- текст файлу `liquibase.properties`;
- текст створеного файлу `changelog.xml`;
- скріншот результату перевірки з'єднання між базою даних та Liquibase (виконання команди `status`);
- скріншот результату оновлення бази даних (виконання команди `update`);
- короткий опис самостійно внесених змін до бази даних;
- виділення змін попереднього пункту у файлі `changelog.xml`;
- скріншот результату повернення до однієї з попередніх версій бази даних (виконання команди `rollback`);
- скріншот сторінки ресурсу Liquibase Hub із під'єднаною базою даних.

## Рекомендована література

1. Brooks, Frederick P. The Mythical Man-Month: Essays on Software Engineering: 20th Anniversary Edition. — 2nd ed. — Addison–Wesley Professional, 1995. — 336 p.
2. Humble Jez, Farley David. Continuous Delivery: A Handbook for Building, Deploying, Testing and Releasing Software. — Addison-Wesley Professional, 2010. — 512 p.
3. Henrik Kniberg. Scrum and XP from the trenches, 2nd Edition, 2015.

## Інформаційні ресурси

1. Git . — [Електронний ресурс]. — Режим доступу : <https://git-scm.com/>.
2. Scott Chacon and Ben Straub. Pro Git, 2019. — [Електронний ресурс]. — Режим доступу: <https://git-scm.com/book/en/v2>.
3. Cătălin Tudose, Petar Tahchiev, Felipe Leme, Vincent Massol, and Gary Gregory. JUnit In Action, Third Edition, 2020. — [Електронний ресурс]. — Режим доступу: <https://www.manning.com/books/junit-in-action-third-edition>.
4. Mockito framework site. — [Електронний ресурс]. — Режим доступу: <https://site.mockito.org/>.
5. Jenkins CI Documentation. — [Електронний ресурс]. — Режим доступу: <https://www.jenkins.io/>.
6. Apache JMeter Documentation. — [Електронний ресурс]. — Режим доступу: <http://jakarta.apache.org/jmeter/>.
7. Noмерpage — Flyway. — [Електронний ресурс]. — Режим доступу: <https://flywaydb.org/>.
8. The Liquibase Community | The Database DevOps Liquibase Community. — [Електронний ресурс]. — Режим доступу: [www.liquibase.org](http://www.liquibase.org).